Bits, Bytes, Ints

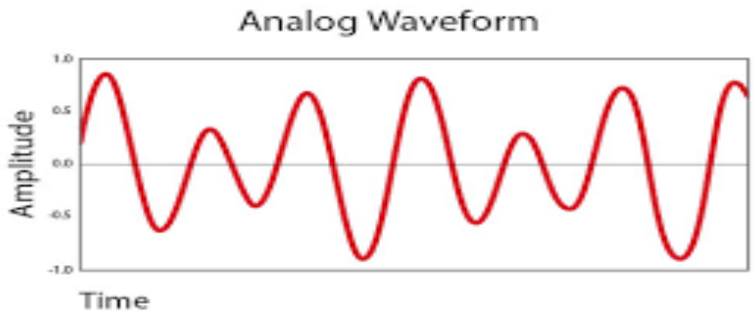
Jinyang Li

Some slides are due to Tiger Wang

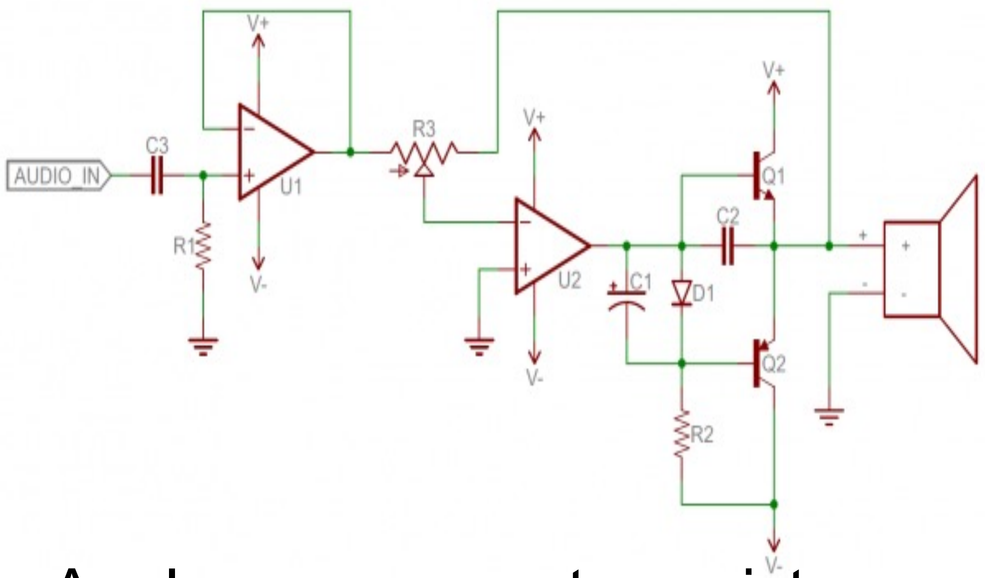
Lesson plan

- How computers represent integers in binary formats
  - Bit, Byte
- How to make binary formats readable to humans
  - Hex notation
- How computers add/subtract integers
- Unsigned vs. signed integer representation

The language of technology has evolved from analog signals…



Analog signals: smooth and continuous

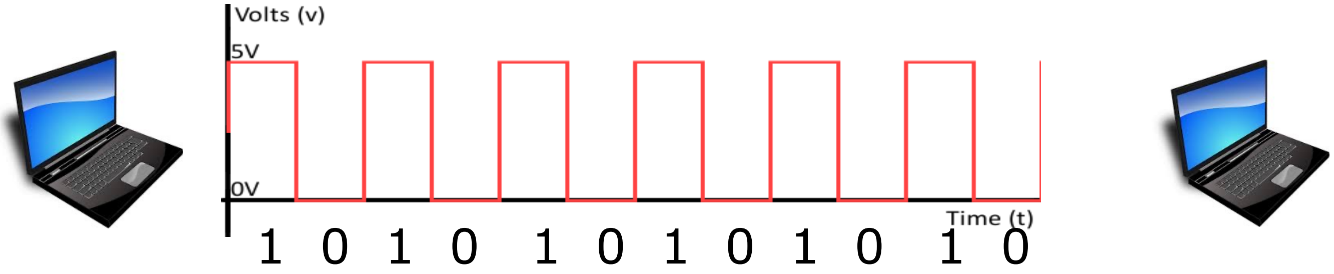


Hard

1. **Difficult to design**
2. **Susceptible to noise**

Analog components: resistors, capacitors, inductors, diodes, etc.

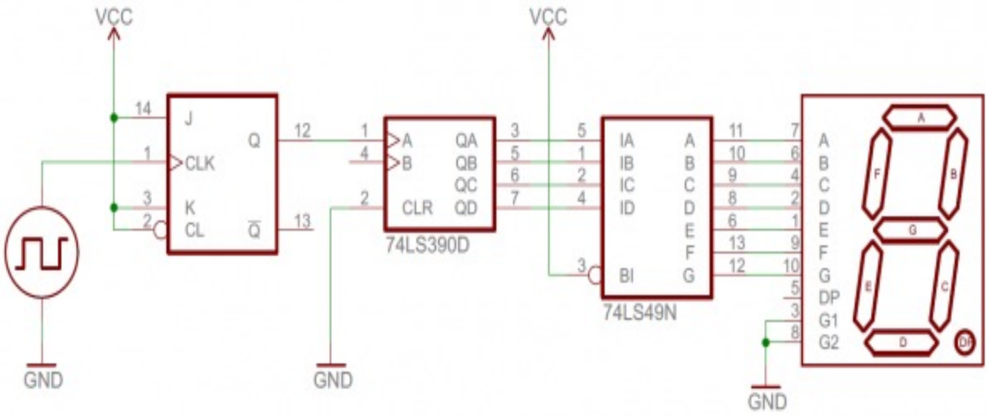
… to digital



Digital signals: discrete (0 or 1)

Easier

1. **Easier to design**
2. **Robust to noise**



Digital components: transistors, logic gates …

Using bits to represent everything

Bit = Binary digit, 0 or 1

- A bit is too small to be useful
  - A bit has 2 values; the English alphabet has 26 values (characters)
- Idea: use a group of bits
  - different bit patterns represent different "values"

Question

- How many bit patterns can a group of 2 bits have?

Can be
either 0 or 1

$b_1\ b_0$

All patterns of 2-bits: 00, 01, 10, 11

- How many bit patterns does a group of n bits have?

$$\underbrace{b_{n-1}b_{n-2}\ldots b_1b_0}_{\text{n bits}}$$

# of patterns of n-bits: 2^n

Digression: Any self-respecting CS person must memorize powers of 2

2^0 = 1

2^1 = 2

2^2 = 4

2^3 = 8

2^4 = 16

2^5 = 32

2^6 = 64

2^7 = 128

2^8 = 256

2^9 = 512

2^{10} = 1024

2^5

2^8

Approximations of powers of 2

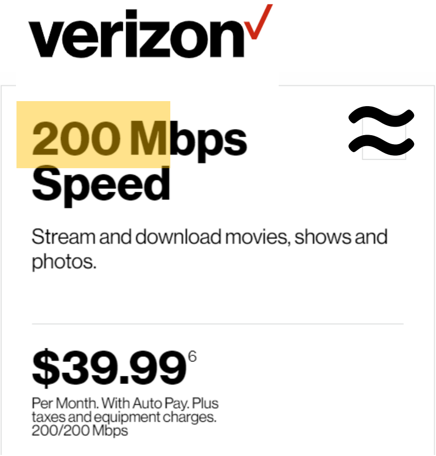
$2^{10} = 1024 \approx 10^{3}$ (Kilo)

$2^{20} \approx 10^{3*2} = 10^{6}$ (Mega)

$2^{30} \approx 10^{3*3} = 10^{9}$ (Giga)

$2^{40} \approx 10^{3*4} = 10^{12}$ (Tera)

$2^{50} \approx 10^{3*5} = 10^{15}$ (Peta)



verizon

200 Mbps Speed

Stream and download movies, shows and photos.

$39.99[6]

Per Month. With Auto Pay. Plus taxes and equipment charges. 200/200 Mbps

$\approx 2^{??}$

Represent ~~non-negative~~ unsigned integer

bits: b_1b_0

Goal: map each bit pattern to an integer

4 patterns

00 → 0

01 → 1

10 → 2

11 → 3

Range: [0, 3]

Represent unsigned integer

Bit pattern: $b_{n-1}b_{n-2}\ldots b_2b_1b_0$

Range: $[0, 2^n -1]$

Base-2 representation:

$$b_{n-1}b_{n-2}\ldots b_2b_1b_0 = \sum_{i=0}^{n-1} b_i * 2^i$$

b_i is bit at i-th position (from right to left, starting at i=0)

Examples

Bit pattern: 00000110

Value: $0*2^7+0*2^6+0*2^5+0*2^4+0*2^3+1*2^2+1*2^1+0*2^0 = 6$

Bit pattern: 10000110

Value:

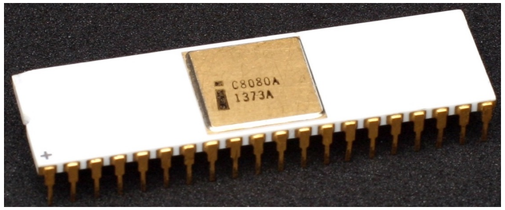
?

Byte

- Byte: a fixed size group of bits
  - The term is coined by Werner Buchholz (IBM).
  - Long long ago, different vendors use different byte sizes
- Now: Byte is 8-bit

IBM System/360, 1964

Introduced 8-bit byte



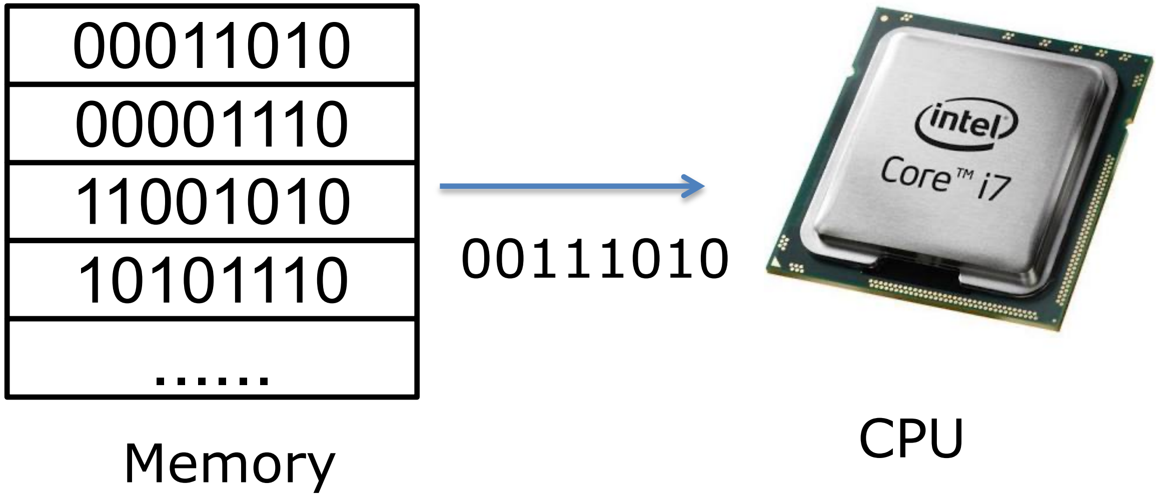
Intel 8080, 1974

Widely adopted

Modern processors

Standardized

Byte



00011010
00001110
11001010
10101110
……

Memory

00111010

CPU

Byte is the smallest unit of information storage, computation and transfer

Integers are represented by 1,2,4, or 8 bytes.

Range of 1-byte non-negative integers: [0, ??]

Bit-pattern of the largest integer?

Range of 4-byte non-negative integers: [0, ??]

Bit-pattern of the largest integer?

Most and least significant bit

MSB: bit position with the largest positional value
LSB: bit position with the smallest positional value

1-byte unsigned int:

10011010

4-byte unsigned int:

01110011 10001101 01010011 11011010

Most significant byte

Least significant byte

Describing bit patterns in a human-readable way

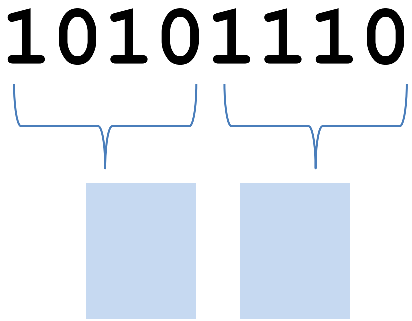
`1-byte int: 10101110`

C program:

```
unsigned int a = 0b10101110;
```

If I ask you to type a 4-byte int, …

Describing a bit pattern: hex notation



Use one (hex) symbol to represent a group of 4 bits

How many hex symbols are needed?

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Hex
1000	8
1001	9
1010	a
1011	b
1100	c
1101	d
1110	e
1111	f

C program:

```
unsigned int a = 0xae;
```

What have we learnt

- How computers represent integers
  - Bit, Byte

Q: What is 10001111 in decimal? A: 143

Q: What's the least significant bit of any even number?

- Hex notation

Q: What is 10001111 in hex? A: 0x8F

Lesson plan

- How computers represent integers
  - Bit, Byte
- Hex notation
- How computers add/subtract integers
- Signed integer representation
  - 2's complement
- A short history of processors:
  - from 8-bit to 64-bit machines
- Byte ordering: big vs. small endian

Unsigned int addition

  0 0 0 0 1 0 1 1

\+ 0 0 0 0 1 0 1 0

---

  0 0 0 1 0 1 0 1

Grade school method

  1 0 0 0 1 0 1 1

\+ 1 0 0 0 1 0 1 0

---

  0 0 0 1 0 1 0 1

😱

Overflow!

Unsigned int subtraction

$$\begin{array}{r} 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0 \\ -\quad 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\ \hline 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \end{array}$$

Grade school method

$$\begin{array}{r} 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0 \\ -\quad 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\ \hline ??? \end{array}$$

How to represent negative numbers?

Represent negative numbers: a strawman

Most significant bit (MSB) represents the sign

$$0\ \underline{0\ 0\ 0\ 0\ 0\ 0\ 1}_2 \longrightarrow 1$$

sign

magnitude

$$1\ 0\ 0\ 0\ 0\ 0\ 0\ 1_2 \longrightarrow -1$$

$$\begin{array}{rr} & 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\ + & 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\ \hline & 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \end{array}$$

-2 ???

☹Need different h/w for signed vs. unsigned computation

Two’s complement

Unsigned int

$00010110 = 0*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$

$10010110 = 1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$

Signed int

$00010110 = 0*(-2^7) +0*2^6 +0*2^5 +1*2^4 +0*2^3 +1*2^2 +1*2^1+0*2^0$

$10010110 = 1*(-2^7) +0*2^6 +0*2^5 +1*2^4 +0*2^3 +1*2^2 +1*2^1+0*2^0$

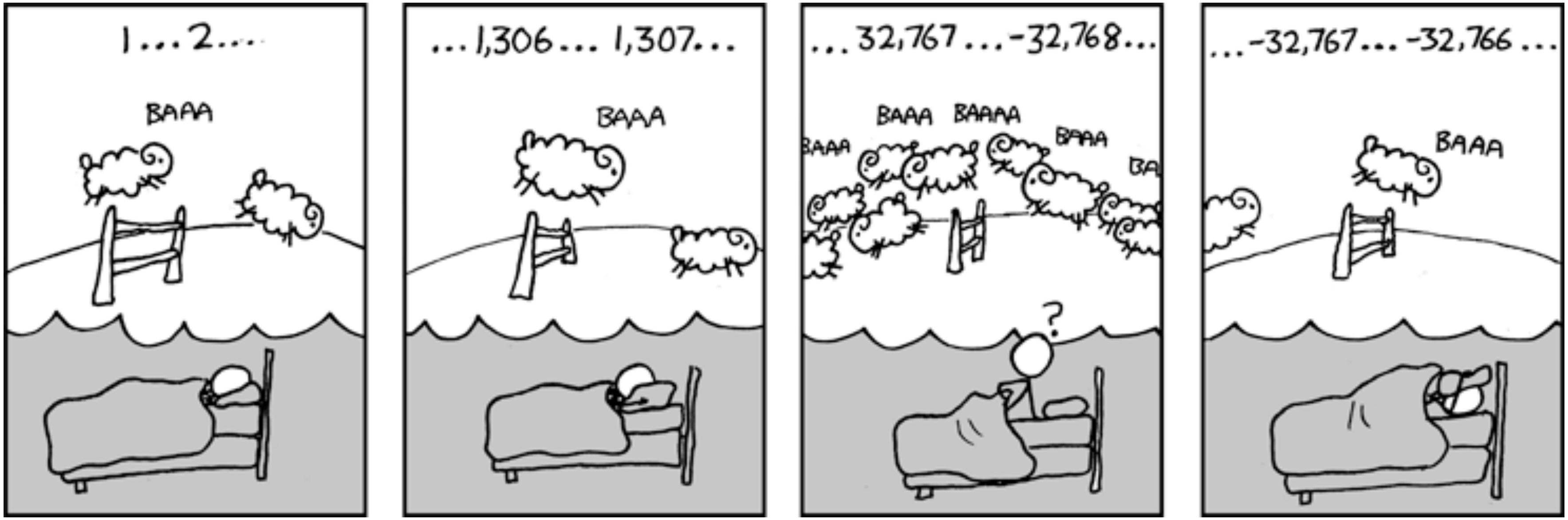
Two's complement

- 1-byte bit pattern → signed int

Bit pattern	value
00000000	0
00000001	1
…	…
01111111	$2^7-1 = 127$
10000000	$-2^7 = -128$
10000001	$-2^7+1= -127$
…	
11111111	$-2^7+(2^7-1) = -1$

Two’s complement

- ?-byte bit pattern → signed int



Source: xkcd.com

Basic facts of 2's complement

Signed int

Size (bytes)	Bit pattern of smallest	Bit pattern of largest	Range
1	`0x80`	`0x7f`	$[-2^{7}, 2^{7}-1]$
2	`0x8000`	`0x7fff`	$[-2^{15}, 2^{15}-1]$
4	`0x80000000`	`0x7fffffff`	$[-2^{31}, 2^{31}-1]$
8	`0x8000000000000000`	`0x7fffffffffffffff`	$[-2^{63}, 2^{63}-1]$

🤔 **Home exercise: make a similar table for unsigned int**

- Negative numbers ←→ MSB=1
- A sequence of 1's (e.g. `0xff`, `0xffffffff`) ←→ -1

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

`00101000` → ??

$(40)_{10}$ → $(-40)_{10}$

A useful trick to do negation:

Step-1: flip all bits

Step-2: add 1

`00101000` $(40)_{10}$

↓ Step-1: flip bits

`11010111`

↓ 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

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

Using negation trick to find the bit-pattern of a negative number

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

Answer:

33 = $(00100001)_2$

Apply negation trick: $(11011110)_2+1=(11011111)_2$

Negation trick helps computers do subtraction

Instead of doing this:

$$\begin{array}{rl} 0\,0\,0\,0\,0\,1\,0\,0 & (4)_{10} \\ -\ 0\,0\,0\,0\,0\,0\,1\,1 & (3)_{10} \\ \hline \\ 0\,0\,0\,0\,0\,0\,0\,1 & (1)_{10} \end{array}$$

Do this instead:

$$0\,0\,0\,0\,0\,0\,1\,1\ (3)_{10} \Rightarrow \begin{array}{rl} 0\,0\,0\,0\,0\,1\,0\,0 & (4)_{10} \\ 1\,1\,1\,1\,1\,1\,0\,0 & \multirow{2}{*}{$\Big\} (-3)_{10}$} \\ +\ 0\,0\,0\,0\,0\,0\,0\,1 & \\ \hline 0\,0\,0\,0\,0\,0\,0\,1 & (1)_{10} \end{array}$$

Works for both unsigned and signed subtraction!

Unsigned addition

$$\begin{array}{r} 0\,0\,0\,0\,0\,0\,0\,1 \;(1)_{10} \\ +\,0\,0\,0\,0\,0\,0\,1\,1 \;(3)_{10} \\ \hline 0\,0\,0\,0\,0\,1\,0\,0 \;(4)_{10} \end{array}$$

$$\begin{array}{r} 0\,0\,0\,0\,0\,0\,0\,1 \;(1)_{10} \\ +\,1\,0\,0\,0\,0\,0\,0\,1 \;(129)_{10} \\ \hline 1\,0\,0\,0\,0\,0\,1\,0 \;(130)_{10} \end{array}$$

$$\begin{array}{r} 0\,1\,0\,0\,0\,0\,0\,1 \;(65)_{10} \\ +\,0\,1\,0\,0\,0\,0\,0\,0 \;(64)_{10} \\ \hline 1\,0\,0\,0\,0\,0\,0\,1 \;(129)_{10} \end{array}$$

$$\begin{array}{r} 1\,0\,0\,0\,0\,0\,0\,1 \;(129)_{10} \\ +\,1\,1\,1\,1\,1\,1\,1\,0 \;(254)_{10} \\ \hline 0\,1\,1\,1\,1\,1\,1\,1 \;(127)_{10} \end{array}$$

Overflow!

Signed addition

0 0 0 0 0 0 0 1 $(1)_{10}$

+0 0 0 0 0 0 1 1 $(3)_{10}$

0 0 0 0 0 1 0 0 $(4)_{10}$

0 0 0 0 0 0 0 1 $(1)_{10}$

+1 0 0 0 0 0 0 1 $(-127)_{10}$

1 0 0 0 0 0 1 0 $(-126)_{10}$

This is what 2's complement is designed to accomplish!

0 1 0 0 0 0 0 1 $(65)_{10}$

+0 1 0 0 0 0 0 0 $(64)_{10}$

1 0 0 0 0 0 0 1 $(-127)_{10}$

Overflow!

1 0 0 0 0 0 0 1 $(-127)_{10}$

+1 1 1 1 1 1 1 0 $(-2)_{10}$

0 1 1 1 1 1 1 1 $(127)_{10}$

Overflow!

Unsigned subtraction

  0 0 0 0 0 0 0 1 $(1)_{10}$

\- 0 0 0 0 0 0 1 1 $(3)_{10}$

  1 1 1 1 1 1 1 0 $(254)_{10}$

😱 Overflow!

  0 0 0 0 0 0 0 1 $(1)_{10}$

\- 1 1 1 1 1 1 1 1 $(255)_{10}$

  0 0 0 0 0 0 1 0 $(2)_{10}$

😱 Overflow!

  0 1 1 1 1 1 1 1 $(127)_{10}$

\- 1 1 1 1 1 1 1 0 $(254)_{10}$

  1 0 0 0 0 0 0 1 $(129)_{10}$

😱 Overflow!

  1 0 0 0 0 0 0 0 $(128)_{10}$

\- 0 0 0 0 0 0 0 1 $(1)_{10}$

  0 1 1 1 1 1 1 1 $(127)_{10}$

Signed subtraction

$$\begin{array}{rl} & 0\,0\,0\,0\,0\,0\,0\,1 \quad (1)_{10} \\ - & 0\,0\,0\,0\,0\,0\,1\,1 \quad (3)_{10} \\ \hline & 1\,1\,1\,1\,1\,1\,1\,0 \quad (-2)_{10} \end{array}$$

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

$$\begin{array}{rl} & 0\,1\,1\,1\,1\,1\,1\,1 \quad (127)_{10} \\ - & 1\,1\,1\,1\,1\,1\,1\,0 \quad (-2)_{10} \\ \hline & 1\,0\,0\,0\,0\,0\,0\,1 \quad (-127)_{10} \end{array}$$

😱

Overflow!

$$\begin{array}{rl} & 1\,0\,0\,0\,0\,0\,0\,0 \quad (-128)_{10} \\ - & 0\,0\,0\,0\,0\,0\,0\,1 \quad (1)_{10} \\ \hline & 0\,1\,1\,1\,1\,1\,1\,1 \quad (127)_{10} \end{array}$$

😱

Overflow!

Lesson plan

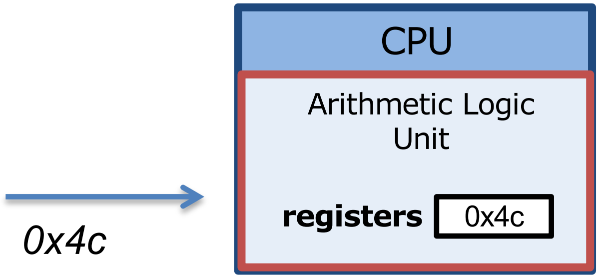
- How computers represent integers
  - Bit, Byte
- Hex notation
- How computers add/subtract integers
- Signed integer representation
  - 2’s complement
- A short history of processors:
  - from 8-bit to 64-bit machines
- Byte ordering: big vs. small endian

THE EVOLUTION OF INTEGER SIZES IN PROCESSORS

8-bit processors: Intel 8080 (1974)

0xb0	0x0f
0xa9	0xaf
0xa8	0x1e
0xa7	0x4c
0xa6	0xea
0xa5	0xcb
0xa4	0xba
0xa3	0x00
0xa2	0xff
0xa1	0x8c
…	……

Memory

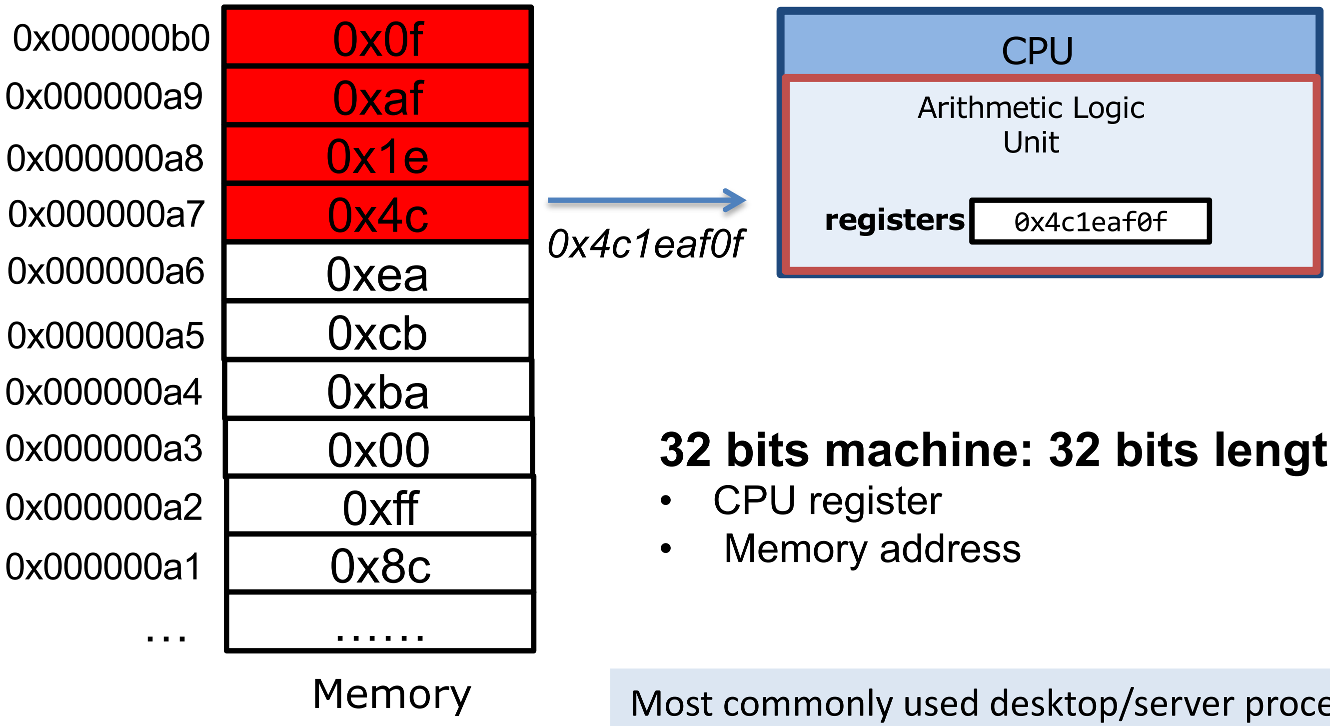


8 bits machine: 8 bits length of

- CPU register
- Memory address

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

32-bit processors: Intel 386 (1985)



0x000000b0	0x0f
0x000000a9	0xaf
0x000000a8	0x1e
0x000000a7	0x4c
0x000000a6	0xea
0x000000a5	0xcb
0x000000a4	0xba
0x000000a3	0x00
0x000000a2	0xff
0x000000a1	0x8c
…	……

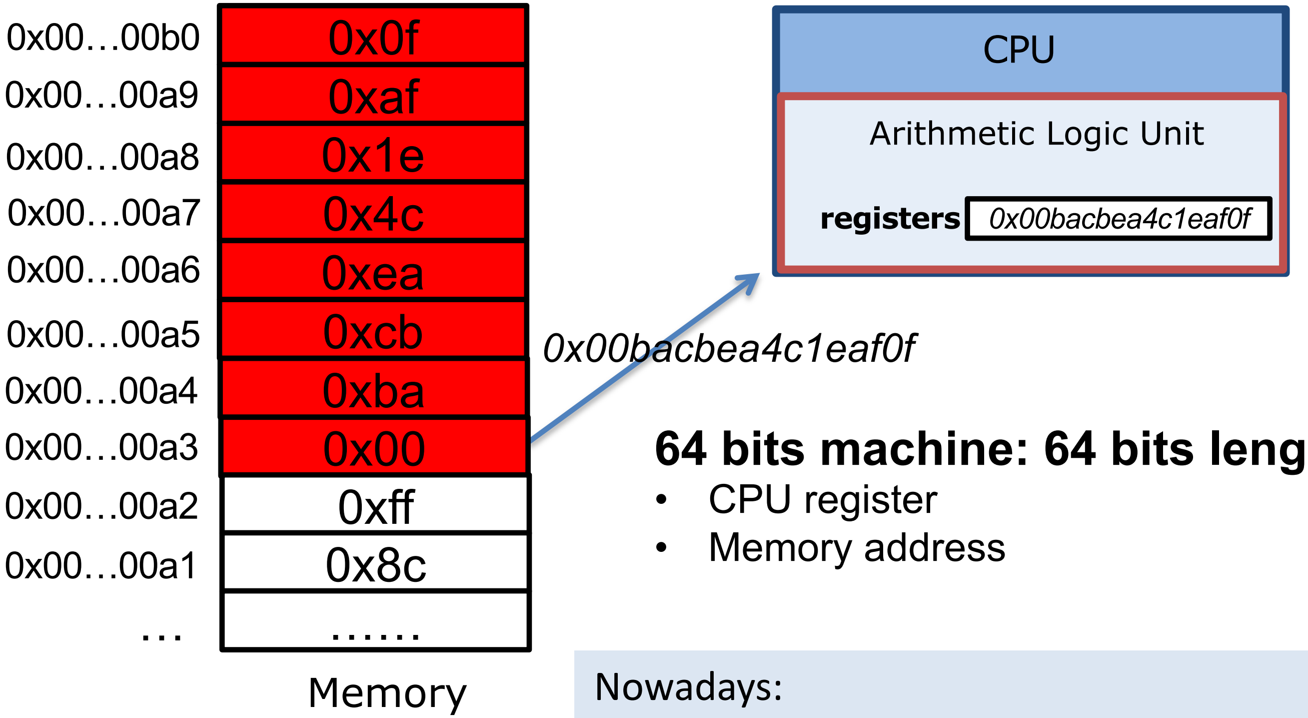
Memory

32 bits machine: 32 bits length of

- CPU register
- Memory address

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

64-bit processors: Intel Pentium 4 (2000)



0x00…00b0	0x0f
0x00…00a9	0xaf
0x00…00a8	0x1e
0x00…00a7	0x4c
0x00…00a6	0xea
0x00…00a5	0xcb
0x00…00a4	0xba
0x00…00a3	0x00
0x00…00a2	0xff
0x00…00a1	0x8c
…	……

Memory

0x00bacbea4c1eaf0f

64 bits machine: 64 bits length of

- CPU register
- Memory address

Nowadays:

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

C’s Integer data types on 64-bit 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

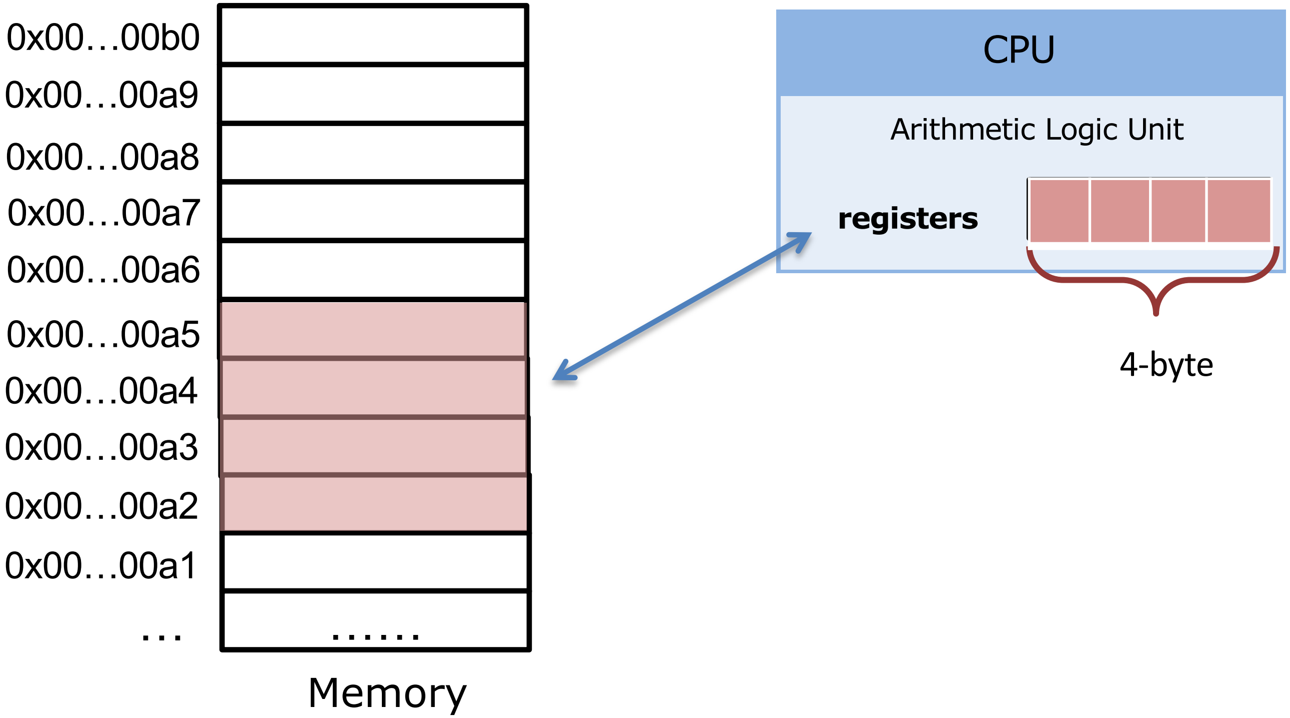
```
#include <stdio.h>

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
```

$$\begin{array}{rrr} & 1\,0\,0\,0\,0\,0\,0\,1 & -127 \\ + & 1\,0\,0\,0\,0\,0\,0\,1 & -127 \\ \hline \boxed{1} & 0\,0\,0\,0\,0\,0\,1\,0 & 2 \end{array}$$

Memory layout for multi-byte integers



0x00…00b0
0x00…00a9
0x00…00a8
0x00…00a7
0x00…00a6
0x00…00a5
0x00…00a4
0x00…00a3
0x00…00a2
0x00…00a1
…
……

Memory

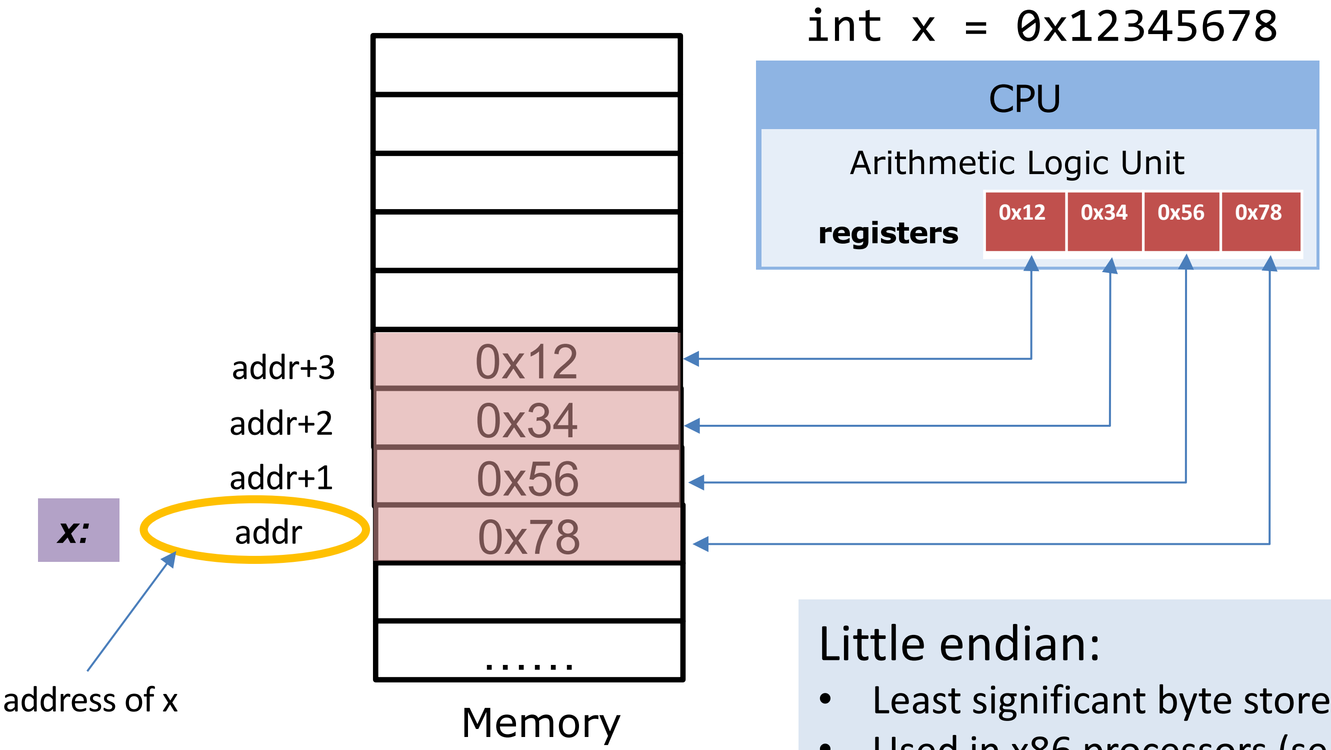
CPU

Arithmetic Logic Unit

registers

4-byte

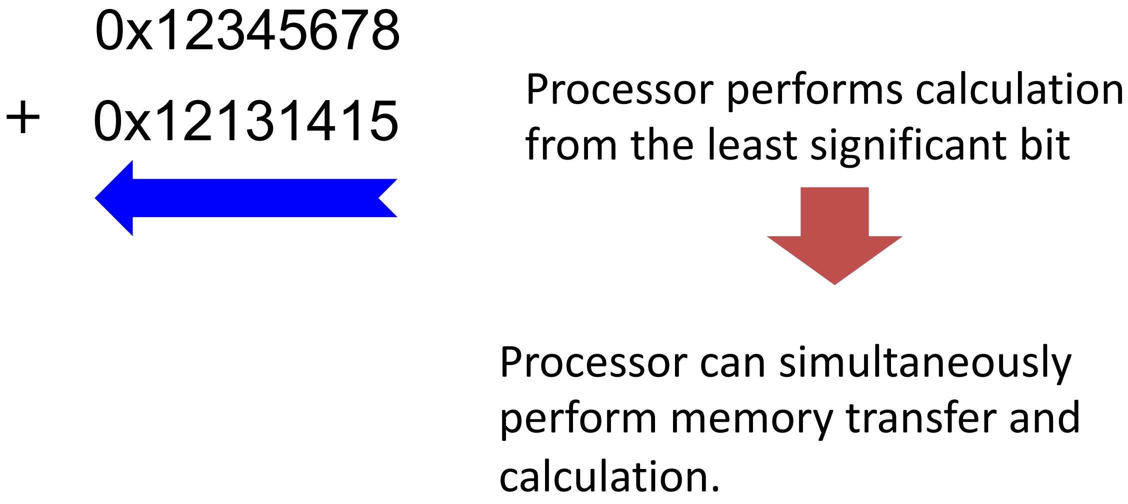
Memory layout: Little Endian



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

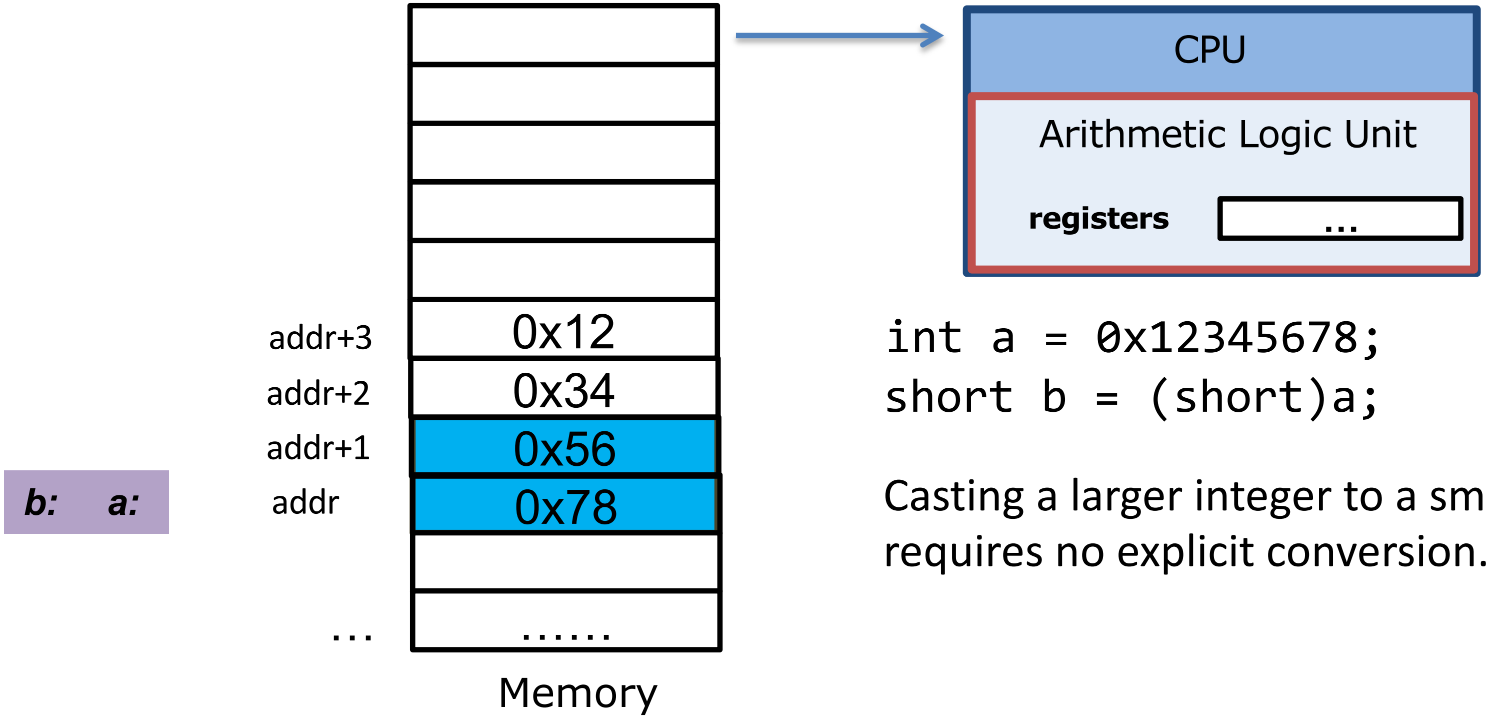


```
  0x12345678
+ 0x12131415
```

Processor performs calculation from the least significant bit

Processor can simultaneously perform memory transfer and calculation.

Another advantage of Little Endian

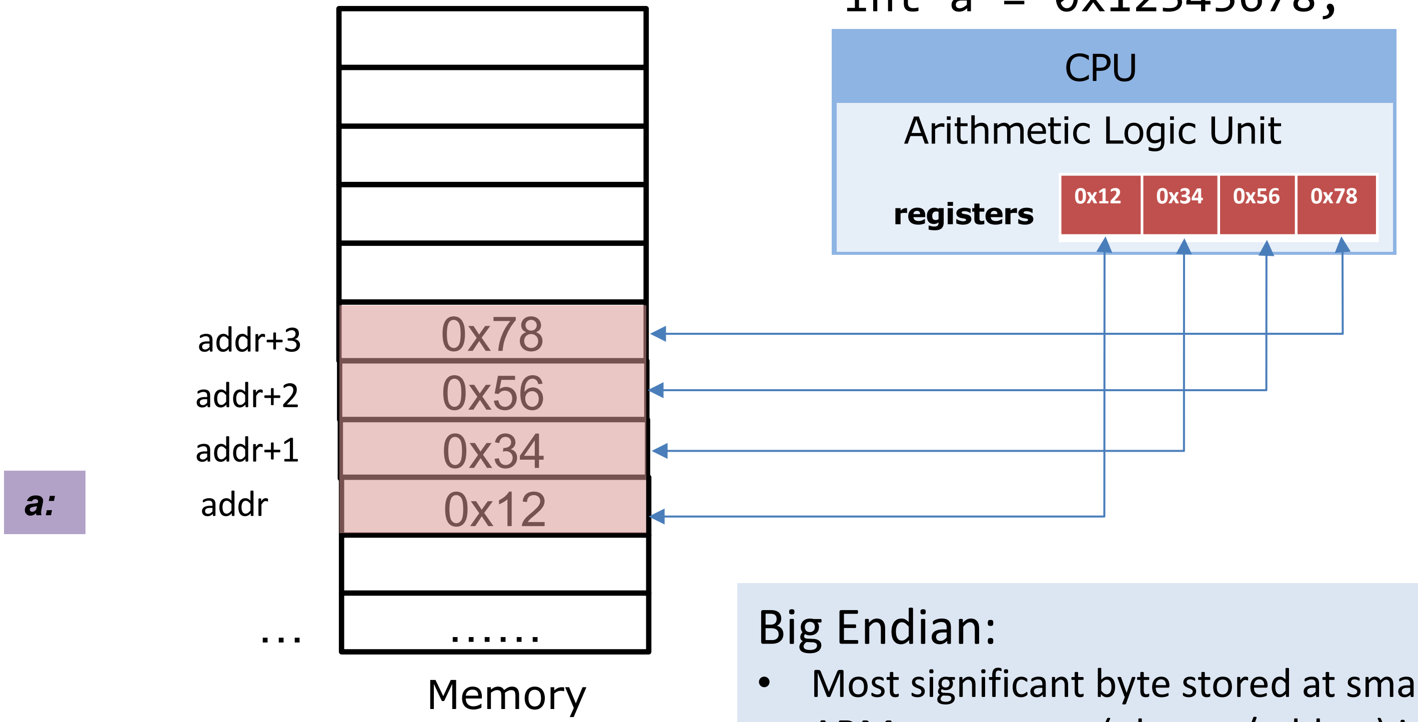


```
int a = 0x12345678;
short b = (short)a;
```

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

Memory layout: Big Endian

`int a = 0x12345678;`



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.

Big or Little Endian?

```
#include <stdio.h>

int main()
{
     int x = 0x05060708;
     int *p;
     p = &x; //p contains the address of x
     char y = *(char *)p; //y is the 1-byte value pointed to by p
     printf(“y=%d\n”, y);
}
```

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

Summary

- Integer representation
  - Unsigned (base-2)
  - Signed (2’s complement)
- Hex notation
- Operations (e.g. add,subtract) on fixed-width integers can cause overflow
- Big vs. little endian