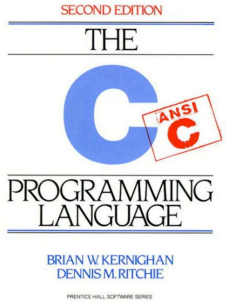
CSC 411

Computer Organization (Fall 2024)
Lecture 4: Integers (signed, unsigned)

Prof. Marco Alvarez, University of Rhode Island

The C Language

- Developed by Dennis Ritchie at Bell Labs in the early 1970s
- Many operating systems, including Unix and its variants (Linux), are written in C
- Allows low-level access to memory, making it efficient for system programming
- C programs are generally portable across different platforms with minimal modification
- C follows a traditional compilation process, where the source code is translated into machine code by a compiler



TIOBE Index for January 2024

- Indicator of the popularity of programming languages
  - popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings.

Jan 2024	Jan 2023	Change	Programming Language	Ratings	Change
1	1		Python	13.97%	-2.39%
2	2		C	11.44%	-4.81%
3	3		C++	9.96%	-2.95%
4	4		Java	7.87%	-4.34%
5	5		C#	7.16%	+1.43%
6	7	↑	JavaScript	2.77%	-0.11%
7	**10**	↑	**PHP**	**1.79%**	**+0.40%**
8	6	↓	Visual Basic	1.60%	-3.04%
9	8	↓	SQL	1.46%	-1.04%
10	20	⇈	Scratch	1.44%	+0.86%

https://www.tiobe.com/tiobe-index/

Representing data

Representing data

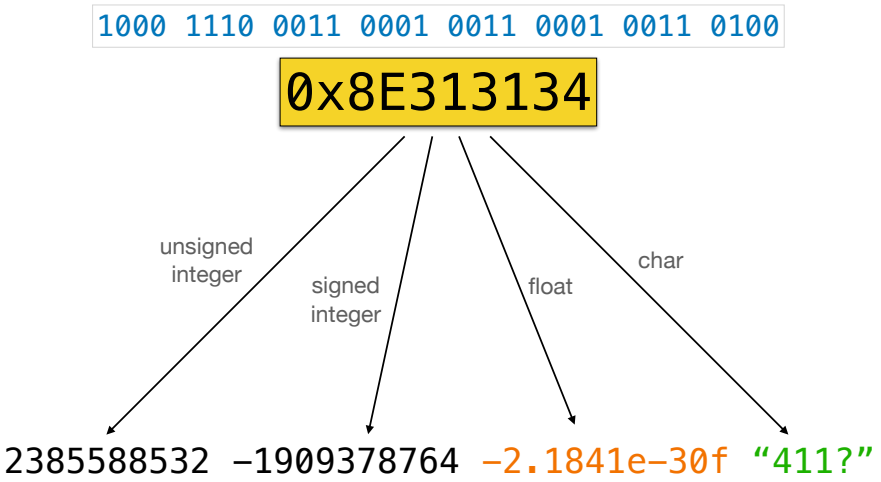
- In memory, **all values** are stored as "bit-vectors"
  - data types are used to interpret the bits (provide meaning)
  - each possible bit-vector assigned exclusively to one meaning
- In a bit sequence of n bits, we can represent 2^n different values
  - number of permutations with repetition (given n digits, there are two ways to choose each digit)
  - example: how many different sequences can be represented in 4 bits?

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

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

$$2^4 = 16$$

Same bits can represent many data types



Show me the code

```c
#include <stdio.h>

int main() {
    unsigned hex = 0x8e313134;

    float *f = (float *) &hex;
    char *s = (char *) &hex;

    printf("%u %d %.4ef %c%c%c%c",
            hex, hex, *f, s[0], s[1], s[2], s[3]);

    return 0;
}
```

```
2385588532  -1909378764  -2.1841e-30f  411?
```

Unsigned integers

Unsigned integers

- Bits represent the number directly
  - same as binary-to-decimal conversion

$$\sum_{i=0}^{n-1} b_i 2^i$$

0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7

1 0 0 0	8
1 0 0 1	9
1 0 1 0	10
1 0 1 1	11
1 1 0 0	12
1 1 0 1	13
1 1 1 0	14
1 1 1 1	15

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

Advantages
- simple representation
- full positive range utilization
- straightforward arithmetic operations

Drawbacks
- cannot represent negative numbers
- overflow not easily detected

Practice

- Provide the range for the following data types:
  - unsigned char (8 bits)
  - unsigned short int (16 bits)
  - unsigned int (32 bits)

Unsigned integer arithmetic

- Addition
  - align numbers and add digits from right to left
  - carry over when sum is greater or equal than the base (2 for binary)

```
  1 1 1
  0 0 1 1 1 0 0 1 +
  0 1 1 1 0 1 1 0
---------------------
  1 0 1 0 1 1 1 1
```

```
    1       1 1 1
  1 1 0 0 0 1 1 1 +
  0 1 0 1 0 1 1 1
---------------------
1 0 0 0 1 1 1 1 0
```

Unsigned integer arithmetic

- Multiplication
  - multiply each digit of multiplier with multiplicand
  - shift partial products left based on multiplier digit position
  - sum all partial products

```
        1 0 0 1 x
        0 1 1 0
-----------------
        0 0 0 0
      1 0 0 1
    1 0 0 1
0 0 0 0
-----------------
0 1 1 0 1 1 0
```

```
        1 1 1 1 x
        1 1 1 1
-----------------
        1 1 1 1
      1 1 1 1
    1 1 1 1
  1 1 1 1
-----------------
1 1 1 0 0 0 0 1
```

Tricky? perform the addition row-by-row

Overflow

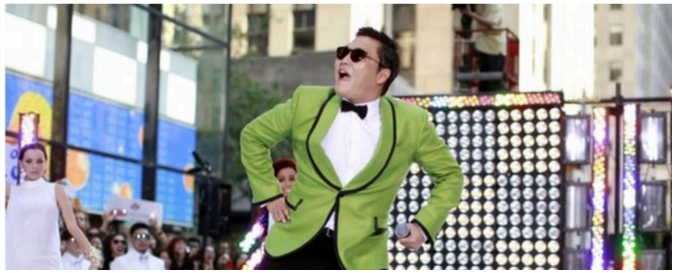
- Definition
  - occurs when the result of an arithmetic operation exceeds the maximum representable value for the bit-width
- Behavior
  - when it occurs, the arithmetic "wraps around", equivalent to performing arithmetic modulo 2^n
    - basically taking the result $\mod 2^n$ (truncating the bits and retaining the n least significant bits)
    - e.g., adding 1 to 255 in an 8-bit system results in 0
  - in C, the runtime does not produce errors, values just **"wrap"**
  - this wrapping around behavior can be useful in certain situations

Overflow

- Can have consequences if not handled properly
  - incorrect calculations
  - program crashes due to unexpected behavior
  - security vulnerabilities
- To prevent overflow
  - choose appropriate data types with sufficient range
  - implement checks and validations within the code

Gangnam Style music video 'broke' YouTube view limit

4 December 2014



YouTube said the video - its most watched ever - has been viewed more than **2,147,483,647** times. It has now changed the maximum view limit to **9,223,372,036,854,775,807**, or more than nine quintillion.

Zero-Day Alert: Google Chrome Under Active Attack, Exploiting New Vulnerability

Nov 29, 2023 Newsroom

Google has rolled out security updates to fix seven security issues in its Chrome browser, including a zero-day that has come under active exploitation in the wild.

Tracked as **CVE-2023-6345**, the high-severity vulnerability has been described as an integer overflow bug in Skia, an open source 2D graphics library.

Signed integers

Sign magnitude

- Trivial approach (**not used**)
  - use MSB as the sign bit, 0 for positive, 1 for negative
  - remaining bits represent magnitude
- Example
  - e.g. all possibilities using $w = 3$ bits

0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	−0
1	0	1	−1
1	1	0	−2
1	1	1	−3

Range: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$

Advantages
- intuitive representation
- easy negation (flip sign bit)

Drawbacks
- two representations of zero (+0 and -0)
- complicates arithmetic circuits (try adding 001 and 110)
- wastes one pattern

One's complement

- Positive numbers
  - same representation as unsigned integers
- Negative numbers
  - bitwise NOT of the positive counterpart
- Called complement because $x + -x = 11\ldots11$
- Example
  - e.g. all possibilities using $w = 3$ bits

0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	−3
1	0	1	−2
1	1	0	−1
1	1	1	−0

Range: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$

Advantages
- easy negation (bitwise NOT)
- symmetric range

Drawbacks
- two representations of zero (+0 and -0)
- complex addition (end-around carry)
- not widely used in modern systems

Two's complement

- Positive numbers
  - same representation as unsigned integers
- Negative numbers
  - bitwise NOT of the positive counterpart plus 1
- Example
  - e.g. all possibilities using $w = 3$ bits

$$-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	−4
1	0	1	−3
1	1	0	−2
1	1	1	−1

Range: $\left[-2^{n-1}, 2^{n-1}-1\right]$

Advantages
- single zero representation
- addition uses same hardware as unsigned representation, no need for end-around carry
- does not waste a pattern
- most widely used representation in modern computers

Drawbacks
- asymmetric range
- slightly more complex negation than one's complement

Practice

- Convert from two's complement to decimal

$-2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

0 0 1 1 1 0 =

1 1 0 0 0 1 =

1 0 0 0 0 0 =

1 1 1 1 1 1 =

1 1 1 1 1 0 =

0 0 0 0 0 0 =

0 0 0 0 0 1 =

0 1 1 1 1 1 =

Example using $n = 4$ bits

Binary	Unsigned	One's complement	Two's complement
0000	0	+0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	-7	-8
1001	9	-6	-7
1010	10	-5	-6
1011	11	-4	-5
1100	12	-3	-4
1101	13	-2	-3
1110	14	-1	-2
1111	15	-0	-1

Practice

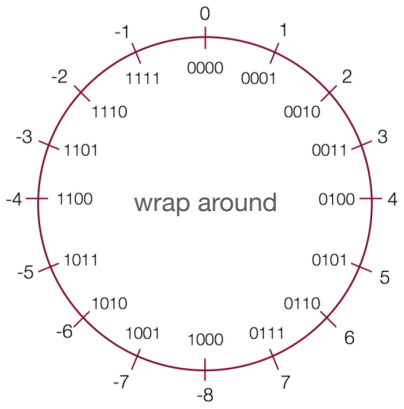
- Convert from x to -x using two's complement

1 1 1 1 =

0 0 1 1 =

0 0 0 0 =

1 0 0 0 =



Practice

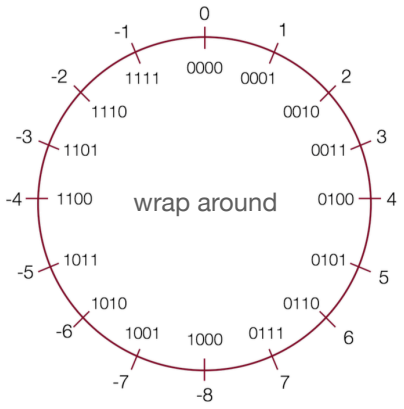
- Calculate addition using two's complement

1111 + 0111 =

0011 + 0110 =

1000 + 1000 =

1001 + 0110 =



Practice

- Provide the range in decimal and hexadecimal for the following data types:
  - unsigned char (8-bit)
  - char (8-bit two's complement)
  - unsigned short int (16-bit)
  - short int (16-bit two's complement)

Further considerations

Sign extension with two's complement

- Sign extension preserves the value of a number when increasing its bit-width (e.g. **casting**)
  - for positive values extend with 0s
  - for negative values extend with 1s

```
        8-bit to 16-bit

01011010   =>   0000000001011010
11011010   =>   1111111111011010
```

- Why it works?
  - maintains relative position of bits in the original number
  - preserves the sign bit
  - arithmetic operations produce correct results across different bit widths

The most negative number

- Exceptions to $\sim x + 1$
  - zero becomes zero (**overflow**)
  - **the most negative number does not have a positive counterpart** — impossible to represent (**overflow**)
- Can lead to unexpected programming bugs
  - in C these behaviors are undefined:

expression	eval
`-(-128)`	`-128`
`abs(-128)`	`-128`
`-128 * -1`	`-128`
`-128 / -1`	`-128`

assume values are signed chars

Basic data types in C

- The C standard does not define the size of “integer” types, except `char`
  - much safer to use `intN_t` and `uintN_t` for signed and unsigned integers of different sizes (stdint.h)
- The type of each variable tells the compiler how many bits are necessary in memory
  - necessary for translation of high level code into machine code

```c
#include <stdint.h>

int main() {
    // unsigned integer using 4 bytes ==> 0x000000C8
    // 0000 0000 0000 0000 0000 0000 1100 1000
    uint32_t myvar = 200;
    // ....
    return 0;
}
```

Range of values

Data type	Size	Format	Value range
character	8	signed	–128 to 127
		unsigned	0 to 255
integer	16	signed	–32768 to 32767
		unsigned	0 to 65535
	32	signed	–2,147,483,648 to 2,147,483,647
		unsigned	0 to 4,294,967,295
	64	signed	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
		unsigned	0 to 18,446,744,073,709,551,615

https://en.cppreference.com/w/cpp/language/types