# COMP1917: Computing 1

# 7. Number Storage and Accuracy

Reading: Moffat, Section 13.2

# Outline

- Binary Arithmetic
- Negative Numbers
- Overflow
- Floating Point
- Roundoff Errors
- Type Conversion

# Decimal Arithmetic – Addition

$$\begin{array}{lccccccl}
\text{carry} & & & 1 & & 1 & & \\
 & 2 & 0 & & 9 & & 7 & \text{Addend} \\
+ & & & & & & 5 & \text{Augend} \\
\hline
 & 2 & 1 & & 0 & & 2 & \text{Sum}
\end{array}$$

- Important principle of "sum" and "carry"

# Binary Arithmetic – Addition

- Similar idea: “sum” and “carry”
- Four cases to consider:

| | | | | |
|---|---|---|---|---|
| Addend | 0 | 0 | 1 | 1 |
| Augend | 0 | 1 | 0 | 1 |
| | — | — | — | — |
| Sum | 0 | 1 | 1 | 0 |
| Carry | 0 | 0 | 0 | 1 |

# Binary Arithmetic – Addition

$10_{10} + 12_{10} = 22_{10}$

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

# Unsigned Data Types in C

| **type** | **bytes** | **bits** | **range** |
|---|---|---|---|
| `unsigned char` | 1 | 8 | 0 ... 255 |
| `unsigned short` | 2 | 16 | 0 ... 65535 |
| `unsigned int` | 4 | 32 | 0 ... 4294967295 |
| | | $n$ | 0 ... $2^n - 1$ |

Note: these sizes are machine dependent. Some machines also provide an "`unsigned long`" type using a larger number of bytes. You can use the `sizeof()` function to determine the sizes on your machine.

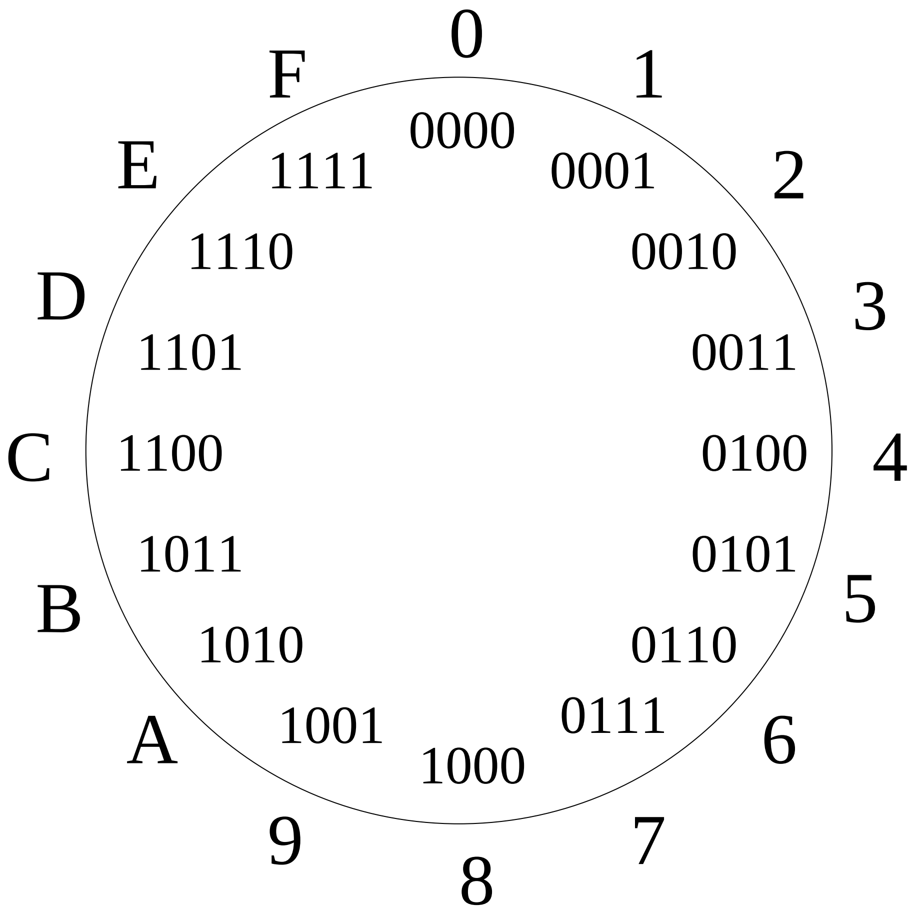
# Overflow

**Question**: What will happen when this code is executed ?

```
unsigned char c = 250;
int i;

for( i=0; i < 10; i++ ) {
   printf("c = %3d\n", c );
   c++;
}
```

# "Clock" Arithmetic



# Representations for Negative Numbers

| Hex | Binary | Unsigned | Sign-Mag | Excess-7 | 2's Complement |
|---|---|---|---|---|---|
| F | 1111 | 15 | −7 | +8 | −1 |
| E | 1110 | 14 | −6 | +7 | −2 |
| D | 1101 | 13 | −5 | +6 | −3 |
| C | 1100 | 12 | −4 | +5 | −4 |
| B | 1011 | 11 | −3 | +4 | −5 |
| A | 1010 | 10 | −2 | +3 | −6 |
| 9 | 1001 | 9 | −1 | +2 | −7 |
| 8 | 1000 | 8 | −0 | +1 | −8 |
| 7 | 0111 | 7 | +7 | 0 | +7 |
| 6 | 0110 | 6 | +6 | −1 | +6 |
| 5 | 0101 | 5 | +5 | −2 | +5 |
| 4 | 0100 | 4 | +4 | −3 | +4 |
| 3 | 0011 | 3 | +3 | −4 | +3 |
| 2 | 0010 | 2 | +2 | −5 | +2 |
| 1 | 0001 | 1 | +1 | −6 | +1 |
| 0 | 0000 | 0 | +0 | −7 | 0 |

# Comparison of Representations

- Signed-Magnitude is difficult to compute with, and has two zeros
- Excess-7 is useful for floating point exponents, but not for general integers
- 2’s Complement is the most convenient, and most widely used

# Justification for 2's Complement

**Motivation**:

We want to find a way of representing negative numbers which allows us to use the same hardware we already use for positive numbers.

**Question 1**:

What number should we use for "minus one" ?
i.e. what number, when one is added to it, becomes zero ?

# “Minus One”

**Answer 1**:

The binary number 11111111 should be used for “minus one”.

**Check**:

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

(Note: the final carry bit is ignored)

**Question 2**:

What number should be the negative of 10011100 ?

# Two's Complement

**Answer 2**:

The negative of 10011100 should be

$$01100011 + 1 = 01100100$$

**Check**:

$$\begin{array}{cccccccc|c} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & + \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & \\ \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & + \\ & & & & & & & 1 & \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \end{array}$$

# Computing Two's Complement

General rule for finding the negative of a binary number:

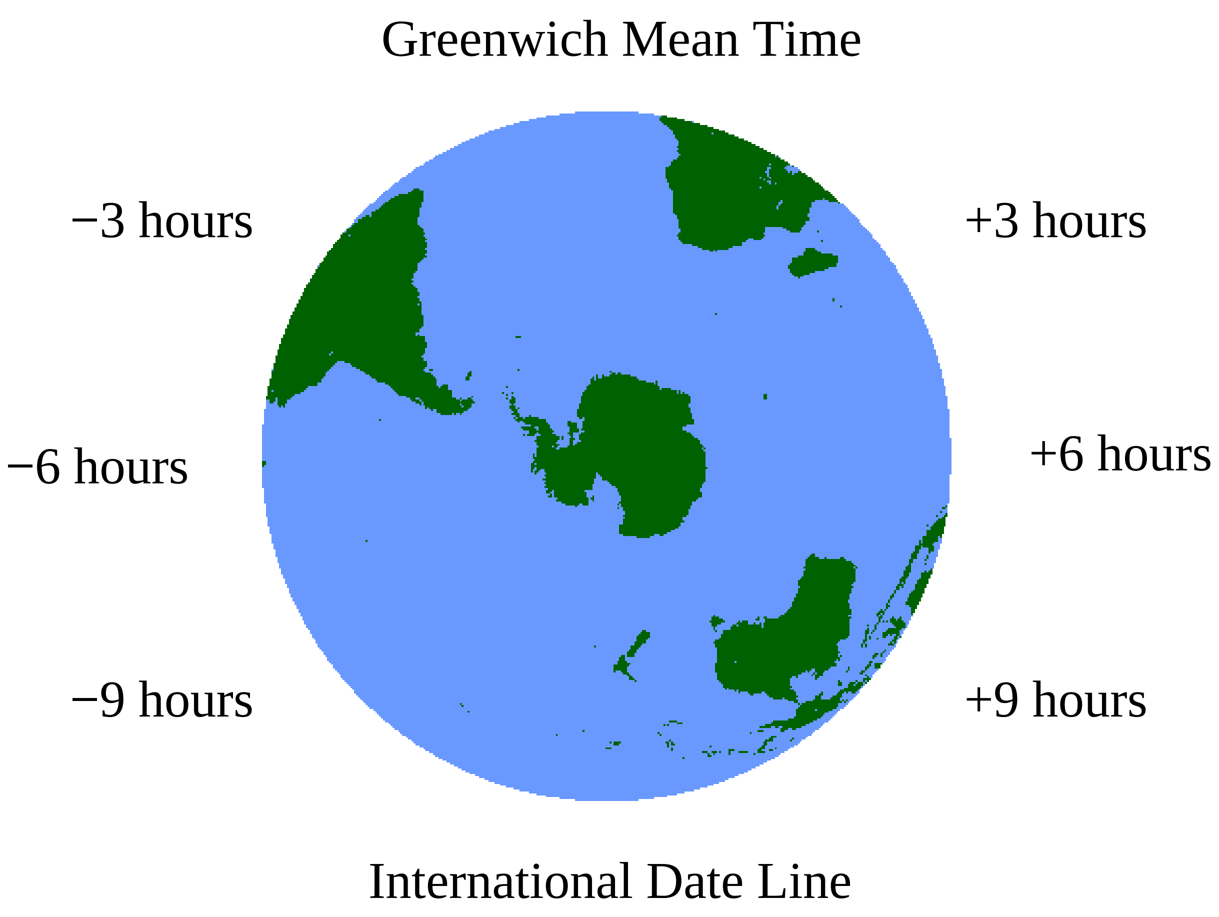
**Step 1**: replace every 1 with 0, every 0 with 1

**Step 2**: add 1

**Example**: find the 2's Complement representation for $-58_{10}$

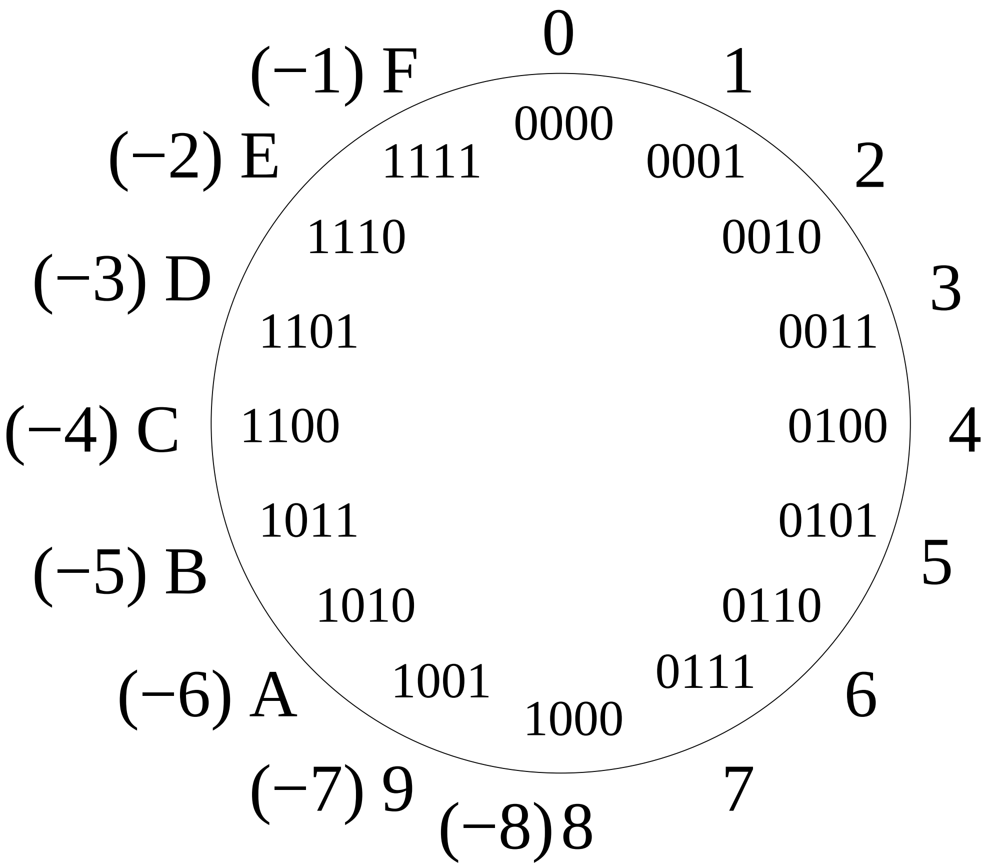
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | one's complement |
| | | | | | | | 1 | $+1$ |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | two's complement |

# Time Zones


Greenwich Mean Time
−3 hours
+3 hours
−6 hours
+6 hours
−9 hours
+9 hours
International Date Line

# Signed Clock Arithmetic



# Sign and MSB

You can tell the sign of the number from the Most Signifcant Bit (MSB)

- if MSB is `1`, number is negative
- if MSB is `0`, number is positive or zero

Note: this means there is always “one extra” negative number.

# Signed Data Types in C

| type | bytes | bits | range |
|---|---|---|---|
| `char` | 1 | 8 | $-128$ ... $+127$ |
| `short` | 2 | 16 | $-32768$ ... $+32767$ |
| `int` | 4 | 32 | $-2147483648$ ... $+2147483647$ |
| | | $n$ | $-2^{n-1}$ ... $+(2^{n-1}-1)$ |

Again, the exact sizes are machine dependent.

Some machines provide a "`long`" type using more bytes.

# Signed Overflow

**Question**: What will happen when this code is executed ?

```
int i=0;

while ( i >= 0 ) {
   i += 1024;
}

printf( "%d  %d\n", i-1, i );
```

# Overflow in Two's Complement

- In two's complement we can represent numbers in the range $-(2^{n-1}) \ldots +(2^{n-1}-1)$
- If we try to add two positive binary numbers $x$ and $y$ where $x+y > 2^{n-1}-1$, the sum will result in a negative number (MSB is 1)
  - positive overflow
- If we try to add two negative binary numbers $-x$ and $-y$ where $x+y > 2^{n-1}$, the sum will result in a positive number (the MSB is 0)
  - negative overflow
- Can use XOR gate in hardware to determine overflow condition

# Positive Overflow in Two's Complement

- Addition of positive numbers without overflow

$$\begin{array}{cccccc} & 0 & x & x & x & \\ & 0 & x & x & x & + \\ \hline & 0 & x & x & x & \end{array}$$

  - ▶ Carry into MSB must have been 0; carry out of MSB is 0

- Addition of positive numbers with overflow

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

  - ▶ Carry into MSB must have been 1; carry out of MSB is 0

- *carry in* $\neq$ *carry out* means overflow has occurred

# Negative Overflow in Two's Complement

- Addition of negative numbers without overflow

$$\begin{array}{ccccc} & 1 & x & x & x \\ & 1 & x & x & x \quad + \\ \hline 1 & 1 & x & x & x \end{array}$$

  - Carry into MSB must have been 1; carry out of MSB is 1

- Addition of negative numbers with overflow

$$\begin{array}{ccccc} & 1 & x & x & x \\ & 1 & x & x & x \quad + \\ \hline 1 & 0 & x & x & x \end{array}$$

  - Carry into MSB must have been 0; carry out of MSB is 1

- *carry in* $\neq$ *carry out* means overflow has occurred

# Decimal Floating Point Numbers

- We want to be able to represent very large and very small numbers, with adequate precision. This can be done with “scientific” or “exponential” notation
  - speed of light $= 1079252848.8 = 1.079 \times 10^9$ km/h
  - mass of proton $= 1.672 \times 10^{-27}$ kg
- This exponential form has 3 components:
  - sign ( ’+’ or ’-’ )
  - exponent (positive or negative integer)
  - fractional part

# Binary Floating Point Numbers

Floating point numbers in the computer

- are in an “exponential” binary form
- are stored in a limited number of bits.

  For example, if 16 bits are available, we might allocate:

  - 1 bit for the sign (0 = ’+’, 1 = ’-’)
  - 4 bits for the binary exponent (in Excess-7 form)
  - 11 bits for fractional part, in binary notation

# Floating Point Example

- How do we interpret this bit pattern as a floating-point number?

$$\underbrace{\texttt{1}}_{\texttt{Sign}}\ \underbrace{\texttt{1100}}_{\texttt{Exp}}\ \underbrace{\texttt{01010110000}}_{\texttt{Fraction}}$$

  - Because $\texttt{Sign} = 1$, the number is negative
  - Exponent is $\texttt{1100}$ in Excess-7, which is $12 - 7 = 5$
  - Binary number is:

$$-1.0101011 \times 2^5 = -101010.11$$

  - Decimal equivalent is $-42.75$

# Floating Point Details

- Highest exponent `1111` is reserved for `+infinity`, `-infinity` or `NaN`
- For exponents between `0001` (-6) and `1110` (+7), we assume a '1' in front of the fractional part (as in the previous example)
- Lowest exponent `0000` is treated as a special case, in order to represent very small numbers (including zero)
  - we assume '0' instead of '1' in front of the fractional part
  - to compensate, the exponent is increased by one (to -6)
  - for example:

$$\underbrace{0}_{\texttt{Sign}}\ \underbrace{0000}_{\texttt{Exp}}\underbrace{00101000000}_{\texttt{Fraction}} = 0.00101 \times 2^{-6}$$

# Floating Point Types in C

| type | bytes | bits | sign | Exponent | Fraction |
|---|---|---|---|---|---|
| `float` | 4 | 32 | 1 bit | 8 bits (Excess-127) | 23 bits |
| `double` | 8 | 64 | 1 bit | 11 bits (Excess-1023) | 52 bits |

There are Web sites where you can type a number and see its representation as a `float` or `double`

`http://babbage.cs.qc.cuny.edu/IEEE-754`

# Roundoff Errors

**Question**: What will happen when this code is executed?

```
float x = 0.0;

while( x < 1.0 ) {
   x = x + 0.02;
}

printf( "x = %1.10f\n", x );
```

# Roundoff Example

**Answer**:

```
x = 1.0199996233
```

**Why?**

- the Binary expansion of 0.02 does not terminate; so it instead gets truncated, producing a small error.
- these small errors accumulate, causing the loop to execute one time too many.
- this problem can often be avoided by using an `int` rather than a `float` to test the loop condition

# Big or Small Numbers First?

Which code will produce the more accurate result?

```
float x = 0.0;
int i;

for(i=0; i<100; i++) {
   x += 0.01;
}
x += 1000000.0;

printf("x=%1.2f\n",x);
```

```
float x = 0.0;
int i;

x += 1000000.0;
for(i=0; i<100; i++) {
   x += 0.01;
}

printf("x=%1.2f\n",x);
```

# Type Conversions

In an expression where you have operands of different types, they are automatically converted to a common type such that:

- the operand with the "narrower" type is converted into a "wider" type. This is done only if there is no loss of information.

**Warning:** Expressions that might lose information, *e.g.* assigning a float to an integer, are permissible. If you are lucky, the compiler may generate a warning.

The best defense against loss of information in automatic type conversion is to be explicit in your type conversion, i.e. when in doubt, make the type conversion explicit.

# Type Conversions *cont.*

What is the output of this code ?

```
float x = 22 / 7;

printf( "%1.2f\n",              x        );
printf( "%1.2f\n",           22 / 7.0 );
printf( "%1.2f\n", (float)   22 / 7   );
printf( "%1.2f\n", (float) ( 22 / 7 ) );
printf( "%1.2f\n",           22 / 7   );
```

Note: the output may be different from one machine to another!