BINARY SUBTRACTION

Kira Clements, University of Bristol

WHAT IS SUBTRACTION?

Subtraction is easy if we think of it as adding one number to a negative number:

$A - B \equiv A + (-B)$	Subtraction is equivalent to addition with a negative value.

This means we can utilise the adder hardware we've already created, at no extra cost!

The only difference is we need a binary representation for signed numbers, and luckily three commonly used options for such representation exist:

- Sign-magnitude
- 1's complement
- 2's complement

SIGN-MAGNITUDE

Signed magnitude representation is the first option for representing signed numbers in binary. This represents value the same way unsigned binary does, except the **MSB represents the sign** instead of a number:

-	2^2	2^1	2^0	Decimal result
0	0	0	1	$2^0 = 1$
0	1	0	1	$2^2 + 2^0 = 4 + 1 = 5$
1	1	0	0	$-(2^2) = -4$
1	0	1	1	$-(2^1 + 2^0) = -(2 + 1) = -3$

Using 4 bits in signed magnitude representation, we can represent numbers +7 to -7. However, we know that a 4-bit binary number can represent 16 (2^4) values, and this is only 15 numbers…

The flaw with this system of representing signed numbers is that 2 options exist for representing 0:

$0000_2 = +0_{10}$ $1000_2 = -0_{10}$

1’S COMPLEMENT

1’s complement is another option for representing signed numbers in binary, which defines a negative number to be the *complement* of the binary number of its positive equivalent. This is the same as **flipping each of the 0s to 1s and vice versa**. The 4-bit representation is shown here:

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

Decimal	-7	-6	-5	-4	-3	-2	-1	-0
Binary	1000	1001	1010	1011	1100	1101	1110	1111

The issue still appears that we have two representations for expressing 0, meaning this system is not making use of the full range of numbers it could be representing.

Furthermore, arithmetic isn’t consistent around this point, as if we were to add 1 to 1111, we would get 0000 (with an overflow) which still represents 0 in decimal.

2'S COMPLEMENT

Instead, 2's complement is the binary representation most modern computers now use. This defines the **MSB still as a signed bit**, like with signed magnitude representation, but this but now **also has a weight attached**.

-2^3	2^2	2^1	2^0	Decimal result
0	0	0	1	$2^0 = 1$
0	1	0	1	$2^2 + 2^0 = 4 + 1 = 5$
1	1	0	0	$-(2^3) + 2^2 = -8 + 4 = -4$
1	0	1	1	$-(2^3) + 2^1 + 2^0 = -8 + 2 + 1 = -5$

We can now represent numbers +7 to -8 using 4 bits as there is now only **one way for representing 0**! This is one more possible value than sign-magnitude or 1's complement, taking full use of the whole range that a 4-bit binary number can express.

You might notice that positive numbers are represented the same as in unsigned binary, but it's important to *pad* with at least one 0 as the MSB will represent a negative value.

2’S COMPLEMENT

To convert a binary number to its negative value using 2’s complement, we *simply find the complement (the 1’s complement) and then add 1*!

For example, the process to find the 2’s complement binary representation of the decimal number -7:

Firstly, we need to identify the minimum number of bits needed to represent this number. As the **range** of a N-bit 2’s complement number is $+(2^{N-1}-1)$ to $-(2^{N-1})$, we must use at least 4 bits.

Unsigned	0	1	1	1
Flip all bits				
1’s complement	1	0	0	0
Add 1				
2’s complement	1	0	0	1

SUBTRACTING

Now we have a method for adding two numbers, as well as representing a negative number, we can subtract one number from another!

$A - B \equiv A + (-B)$	$A - B \equiv A + (NOT(B) + 1)$

To alter our adder circuit to enable subtraction with the same architecture, we can use NOT gates to flip bits and the unused C_in input on the first full adder unit in the ripple carry adder allows us to add 1!

$A = 3 = 0011_2$
$B = 7 = 0111_2$

A	0	0	1	1	
NOT(B) + 1	1	0	0	1	+
Sum	1	1	0	0	
Carry		1	1		

Sum = $-(2^3) + 2^2 = -4$

To make a functional adder-subtractor we simply need a method of choosing whether to NOT B's input and add one to the first adder unit.

OVERFLOW

Previously, the overflow bit was simply ignored while adding numbers, but we can now see this now serves an important purpose. When adding binary numbers using 2's complement representation, the overflow is used to help determine whether the result is correct after the extra bit is discarded.

6 + 5 = -5?

A		0	1	1	0
B		0	1	0	1 +
Sum	0	1	0	1	1
Carry		1			

-7 + -2 = 7?

A		1	0	0	1
B		1	1	1	0 +
Sum	1	0	1	1	1
Carry	1				

Also known as **underflow** (when the result is too small to be stored)

In both the above examples, we are working in 4-bit binary numbers so bit number 4 of the result must be discarded. However, this changes the sign of the result!

This will flag an overflow error, indicating you have either added two positives and found a (incorrect) negative result, or vice versa.

OVERFLOW IN C

Integer overflow can be the reason a program doesn't work as expected…

```c
int num = INT_MAX;
// prints the maximum value for an int (2147483647 for 32-bit int)
printf("%i\n", num);

num++;
// prints the value after overflow (-2147483648 for 32-bit int)
printf("%i\n", num);
```

Overflow errors demonstrate why it can be a good idea to assume variables are store using the worst-case amount of memory i.e. the minimum size.

If a calculation works on one machine, it may not behave as intended if the program runs on a different machine that uses less memory to store that variable.

FIXED-POINT NUMBERS

With integers we lose precision as we cannot represent anything that's not a whole number. Fixed-point representation is an option for representing fractional numbers that is similar to binary, except some of the bits represent the fraction (still in powers of 2).

Notation	2^1	2^0	2^{-1}	2^{-2}	Decimal result
00.01	0	0	0	1	$2^{-2} = 0.25$
01.01	0	1	0	1	$2^0 + 2^{-2} = 1 + 0.25 = 1.25$
11.00	1	1	0	0	$2^1 + 2^0 = 2 + 1 = 3$
10.11	1	0	1	1	$2^1 + 2^{-1} + 2^{-2} = 2 + 0.5 + 0.25 = 2.75$

Location of decimal point is determined by deciding the **range** (maximum and minimum values) and **precision** (increments between values) required. As we can see from the table, 2 integer bits and 2 fractional bits only lets us represent values between 0 to 3.75, but in increments of 0.25.

We can also have a fixed-point number that uses 2's complement (MSB represents a negative value) if we want negative fractional numbers.

FLOATING-POINT NUMBERS

Floating-point is a more flexible fractional representation as it doesn't require the number of integer and fractional bits to be determined in advance. Instead, it uses the following formula:

$(-1)^S \cdot M \cdot B^E$	S = Sign, M = Mantissa, B= Base, E = Exponent

This representation uses a **biased exponent** (adds a bias so that negative exponents are also stored as a positive value) and a **normalised mantissa** (doesn't store an implicit leading 1).

The most common floating-point precision is defined by the **IEEE 754 technical standard**, which also defines how special values like infinity and NaN are be represented.

What does this look like in practise?

FLOATING-POINT NUMBERS

We first decide on our precision, for this example let's use S(1), E(3), and M(4).

To store a decimal value, we convert the number into binary like for fixed-point numbers:

$2.125_{10} = 10.001_2$ (sign = 0) and $-0.625_{10} = 0.101_2$ (sign = 1)

We then **normalise** these binary numbers, so that they have a single leading 1 before the point:

$10.001_2 = 1.0001_2 \cdot 2^1$ and $0.101_2 = 1.01_2 \cdot 2^{-1}$

These normalised numbers can then be converted into floating-point notation as follows:

Sign	Biased exponent			Normalised mantissa				Decimal result
-	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
0	1	0	0	0	0	0	1	$(-1)^0 \cdot (2^0 + 2^{-4}) \cdot 2^{(4-3)} = 2.125$
1	0	1	0	0	1	0	0	$(-1)^1 \cdot (2^0 + 2^{-2}) \cdot 2^{(2-3)} = -0.625$

For a 3-bit exponent, the bias is 3, so that exponents -3 to 4 can be represented as 0 to 7 e.g. biased exponent of 2 = exponent of -1.

The implicit 1 isn't stored.

FLOATING-POINT ERRORS

The fractional part of floating-point numbers is made from the sum of a selection of negative powers of two (2^{-1} = ½, 2^{-2} = ¼, 2^{-3} = ⅛, …).

Because of this, many simple-looking fractions can't be represented exactly, only approximated, while some difficult-looking fractions can be represented exactly.

```c
float num = 0.3;
// prints 0 i.e. false, as not exactly representable in binary
printf("%i\n", num == 0.3);

num = 0.1328125;
// prints 1 i.e. true, as exactly representable in binary (2^-3 + 2^-7)
printf("%i\n", num == 0.1328125);
```