Computer representation of negative integers

- Typically a fixed number of bits is used to represent integers: 8, 16, 32 or 64 bits
  - Unsigned integer can take all space available
- Signed integers
  - Leading sign

$$\begin{aligned} \mathbf{0}\,000\,0001_2 &= 1_{10} \\ \mathbf{1}\,000\,0001_2 &= -1_{10} \end{aligned}$$

    but then

$$\mathbf{1}\,000\,0000_2 = -0_{10} \quad (?!)$$

  - Two's complement:
    given a positive integer a, the **two's complement of** a **relative to a fixed bit length** n is the binary representation of

$$2^n - a.$$

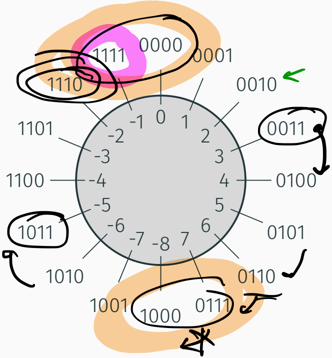
Example: 4-bit two's complement (n=4)

blue

- $a = 1$, two's complement: $2^4 - 1 = 15 = 1111_2 = -1$
- $a = 2$, two's complement: $2^4 - 2 = 14 = 1110_2 = -2$
- $a = 3$, two's complement: $2^4 - 3 = 13 = 1101_2 = -3$
- ...
- $a = 8$, two's complement: $2^4 - 8 = 8 = 1000_2 = -8$

- Positive numbers start with 0, negative numbers start with 1
- 0 is always represented as a string of zeros
- -1 is always represented as a string of ones

Example: 4-bits



- The number range is split unevenly between +ve and -ve numbers
- The range of numbers we can represent in n bits is -2^{n-1} to $2^{n-1} - 1$

Addition

- Easy for computers
- Example: 2+3

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

- A carry that goes off the end can often be ignored
- Example: $-1 + -3$

$$\begin{array}{rcccc} & 1 & 1 & 1 & 1 \\ + & 1 & 1 & 0 & 1 \\ \hline {}_1 & 1 & 1 & 0 & 0 \end{array}$$

Subtraction

- Treat as an addition by negating second operand
- Example: $4 - 3 = 4 + (-3)$

$$\begin{array}{rcccc} & 0 & 1 & 0 & 0 \\ + & 1 & 1 & 0 & 1 \\ \hline {}_1 & 0 & 0 & 0 & 1 \end{array}$$

Overflow

- Example: $4 + 7$

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

- The correct result 9 is too big to fit into 4-bit representation
- Testing for overflow:
  If both inputs to an addition have the same sign, and the output sign is different, overflow has occurred
  - Overflow cannot occur if inputs have opposite sign.

Two's complement and bit negation

Example $n = 4$

- $2^4 - a = ((2^4 - 1) - a) + 1.$
- The binary representation of $(2^4 - 1)$ is 1111_2
- Subtracting a 4-bit number a from 1111_2 just switches all the 0's in a to 1's and all the 1's to 0's.

  For example,

$$\begin{array}{rcccc} & 1 & 1 & 1 & 1 \\ - & 1 & 0 & 0 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

- So, to compute the two's complement of a, flip the bits and add 1.

Example

- Find the 8-bit two's complement of 19.

$$19 = 16 + 2 + 1 = 16 + 0\cdot 8 + 0\cdot 4 + 1\cdot 2 + 1\cdot 1 = 10011_2$$
$$= 00010011$$
$$11101100 + 1 = 11101101$$

- Conversely, observe that

$$2^n - (2^n - a) = a$$

so to find the decimal representation of the integer with a given two's complement
  - Find the two's complement of the given two's complement
  - Write the decimal equivalent of the result

Example: Which number is represented by 1010 1001?

$$01010110 + 1 = 01010111$$
$$64 + 16 + 4 + 2 + 1$$

8-bit two's complement of 6

$6_{10} = 00000110_2$

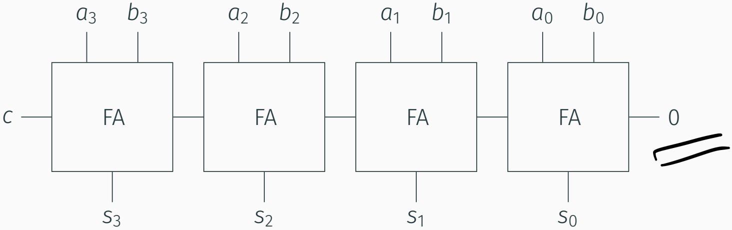
Flip the bits: 11111001

Two's complement: 11111001 + 1 = 11111010

11111001 = -7

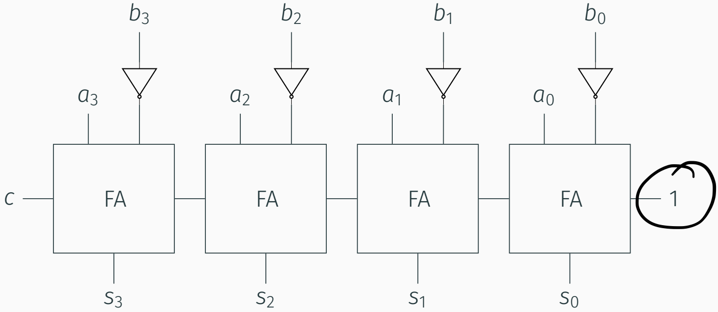
00000110 + 1 = 00000111 = 7

Recall: 4-bit adder

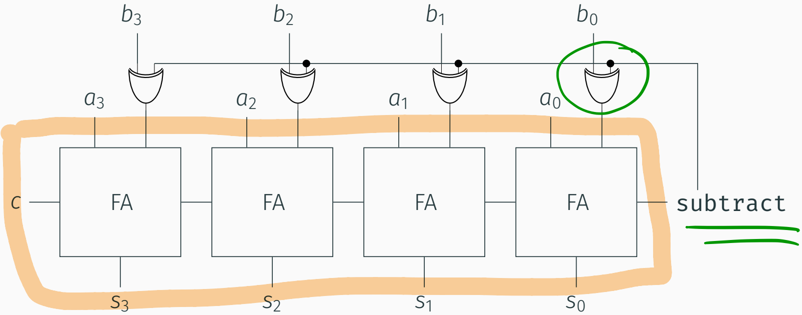


4-bit subtractor

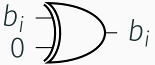
- Implementing $a + b$ as the sum of a and two's complement of b



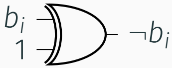
4-bit adder / subtractor



- When **subtract** is 0:



- When **subtract** is 1:



Integer types in high-level languages

E.g. Java has the following integer data types, using 2's complement:

byte	8-bit	-128 to $+127$
short	16-bit	$-32\,768$ to $+32\,767$
int	32-bit	$-2\,147\,483\,648$ to $+2\,147\,483\,647$
long	64-bit	-2^{63} to $+2^{63}-1$

Floating point numbers

- It is not always possible to express numbers in integer form.
- Real, or floating point numbers are used in the computer when:
  - the number to be expressed is outside of the integer range of the computer, like
    $$3.6 \times 10^{40} \text{ or } 1.6 \times 10^{-19}$$
  - or, when the number contains a decimal fraction, like
    $$123.456$$

Scientific notation (AKA standard form)

The number is written in two parts:

- Just the digits (with the decimal point placed after the first digit), followed by
- $\times 10$ to a power that puts the decimal point where it should be (i.e. it shows how many places to move the decimal point).

$$123.456 = 1.23456 \times 10^2$$

In this example, 123.456 is written as 1.23456×10^2 because $123.456 = 1.23456 \times 100 = 1.23456 \times 10^2$

$$12.34 = 10 + 1 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$$
$$10 + 1 + 0.3 + 0.04$$

Likewise, fractions can be represented base 2.

$$\frac{1}{2^2}$$

$$\begin{aligned} 10.01_2 &= 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} \\ &= 1 \times 2 + 0 + 0 + 1 \times 0.25 \\ &= 2.25_{10} \end{aligned}$$

Scientific representation: $10.01_2 = 1.001 \times 2^1$

Note: in binary, for any non-zero number the leading digit is always 1

Computer representation

To represent a number in scientific notation:

- The sign of the number.
- The magnitude of the number, known as the mantissa or significand
- The sign of the exponent
- The magnitude of the exponent

Example: eight characters

S EE MMMMM

- S is the sign of the number
- EE are two characters encoding the exponent
  - both sign and magnitude
- MMMMM are five characters for the mantissa

IEEE 754

- IEEE standard for floating-point arithmetic
- Implemented in many hardware units
- Stipulates computer representation of numbers
- For binary:
  - 16 bit half precision numbers: 5 for exponent, 11 for mantissa
  - 32 bit single precision numbers: 8 for exponent, 24 for mantissa
  - 64 bit double precision numbers: 11 for exponent 53 for mantissa
  - 128 bit quadruple precision numbers: 15 for exponent 113 for mantissa
  - 256 bit octuple precision numbers: 19 for exponent 237 for mantissa