COM1006 Devices and Networks (Autumn)
COM1090 Computer Architectures

Lecture #2

▶ Computer arithmetic: Integers

Dr Dirk Sudholt
Department of Computer Science
University of Sheffield

d.sudholt@sheffield.ac.uk

https://staffwww.dcs.shef.ac.uk/people/D.Sudholt/campus_only/COM1006.htm

Partly based on 4.7-4.8 in Clements, Principles of Computer Hardware

(we'll talk about adders and other hardware implementations later)

Aims of this lecture

- To explain how binary numbers are added.
- To explain sign-and-magnitude and complementary representations for negative numbers.
- To introduce two's complement numbers.
- To show how arithmetic overflow can occur when adding two's complement numbers, and to explain how overflow can be detected.

Binary arithmetic

- Binary arithmetic can be described by addition, subtraction and multiplication tables in the same way as decimal arithmetic (but the tables are much simpler).

Addition	Subtraction	Multiplication
0 + 0 = 0	0 - 0 = 0	0 x 0 = 0
0 + 1 = 1	0 - 1 = 1 borrow 1	0 x 1 = 0
1 + 0 = 1	1 - 0 = 1	1 x 0 = 0
1 + 1 = 0 carry 1	1 - 1 = 0	1 x 1 = 1

- Subtraction can be implemented by **addition and negation**:

$$X - Y = X + (-Y).$$

- If we can do addition and negation, we can do subtraction.

Adding binary numbers

- To add m-bit binary numbers we need to consider the carry out to the left and carry in from the right.
- Example: `00110111` + `01010110`

```
 00110111
+01010110
 111 11      ← carries
 --------
 10001101
```

Note on *m*-bit arithmetic

- Computers do arithmetic on a fixed word length.
- ***m*-bit arithmetic**: all numbers have **exactly *m* bits** (no less, no more)
- might have to fill up with leading zeros, e.g. in 8-bit arithmetic we write 3_{10} as 00000011_2.
- the result of an *m*-bit addition is an *m*-bit number:

$$0011 + 0100 = 0111$$

- correct result may not fit in *m* bits, still we only have *m* bits available!
- we'll be using fixed numbers of bits in the remainder of this course!

▶Signed numbers

- An n-bit word has 2^n possible values from 0 to 2^n-1 (e.g., 8-bit word has values from 0-255).
- How should we represent **negative** numbers?
- One way is to use the most significant bit to indicate the sign of the number (0 for positive, 1 for negative numbers).
- Example:

$$00001101_2 = +13_{10}$$
$$10001101_2 = -13_{10}$$

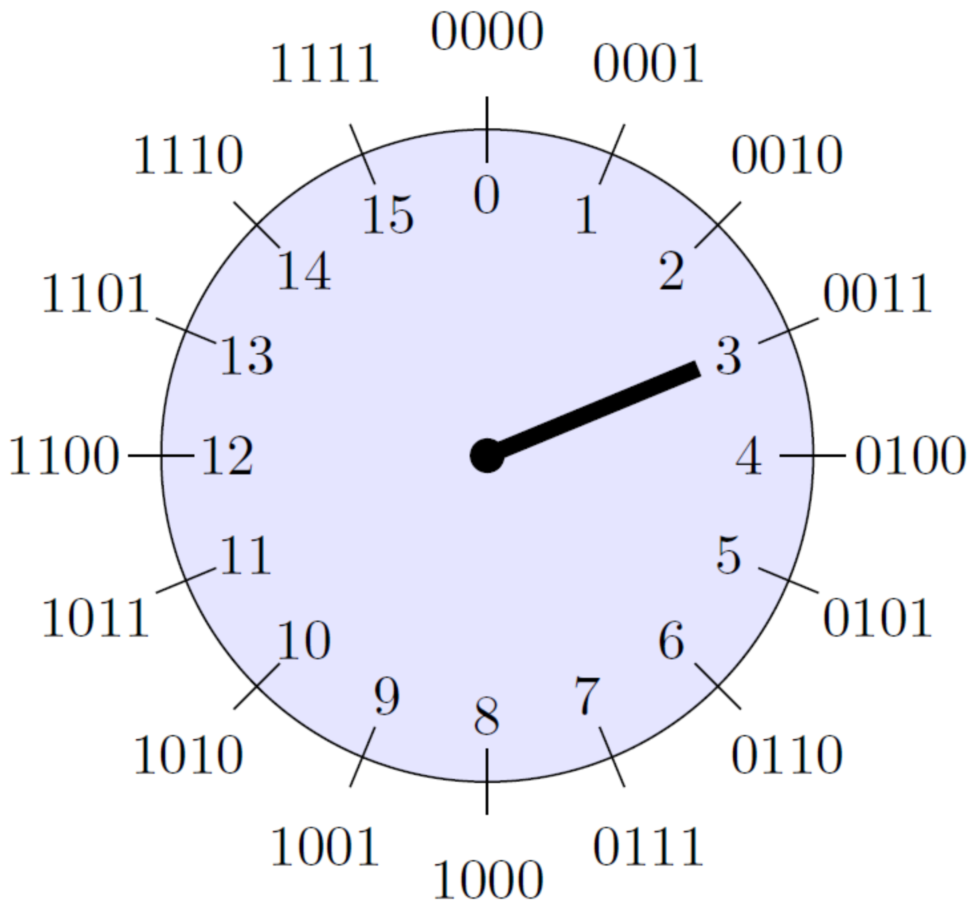
- This is a **sign and magnitude** representation.
- **Negation:** flip (invert) sign bit.

Problems with sign and magnitude form

- The sign and magnitude representation amounts to using 1 bit of an *n*-bit number to represent the sign, so that the remaining *n*-1 bits represent a magnitude in the range $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$, e.g. 8-bit word has range -127 to +127.
- In practice there are objections to this approach:
  - There are two values for zero:
    $00000000_2 = +0$
    $10000000_2 = -0$
  - Addition requires a case distinction:
    operands have the same sign → add *n*-1 bits, take over sign
    operands have different signs → need to do subtraction
- **Complementary arithmetic** provides a better solution.

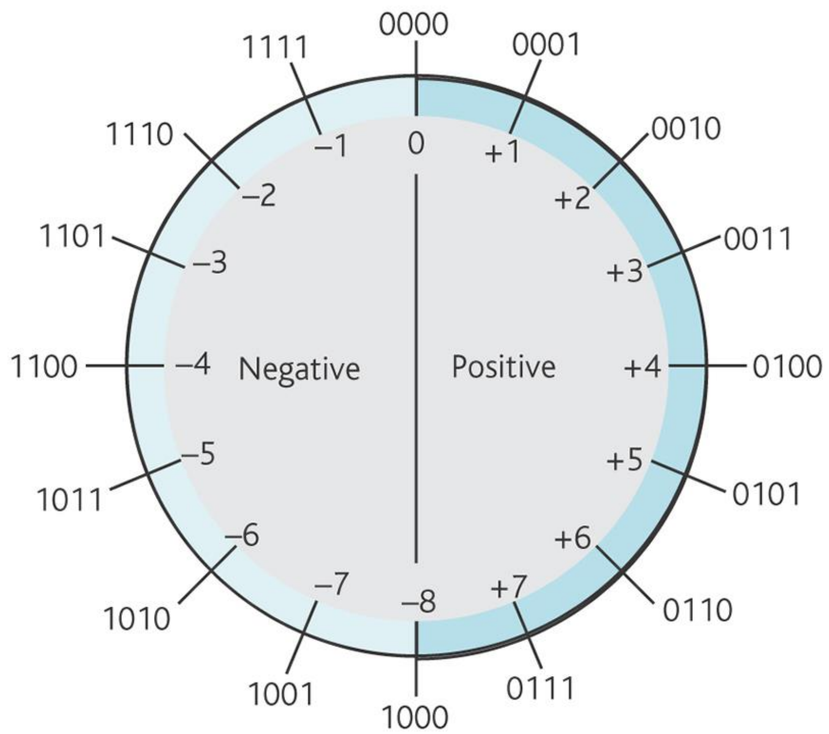
A Wall Clock for Planet Neptune (1 day=16hrs)

- Assume hour hand **can only go forward**.
- Add 3 hours:
  3 + 3 = 6.
- Subtract 2 hours:
  3 + (16 – 2)
  = 1 (modulo 16).
- „–2“ is expressed as
  16 – 2 (16=#hours on clock)
- Transition from 15 to 0 is harmless here.



Two's complement representation

- In the **two's complement system** both positive and negative numbers are represented in the same form.
- Half the numbers are negative.
- First bit indicates the sign.
- **Numbers increase in clockwise direction** (unlike sign & magnitude).
- **Subtraction by addition** still works like on Neptune: X – Y = X + (16-Y)
- Allows to add **positive and negative** numbers.



Two's complement of an *n*-bit number

- How can we **negate a number**, e.g. turn +6 into -6?
- Looking for definitions that work for *n*-bit numbers.
- The **two's complement** of an *n*-bit binary number N is 2^n-N.
  - two's complement of N represents $-N$ using the binary number for 2^n-N
  - similar to going from N to $-N$, but we add 2^n to get a positive number 2^n-N (which is positive since $N < 2^n$)
- Example for 4 bits:

  The two's complement of $N=6_{10}= 0110_2$ is $2^4-6=10_{10}=1010_2$

  Note that 1010_2 can be interpreted either as the two's complement integer -6 or the unsigned integer +10.

Calculating two's complements in binary

- The two's complement system is attractive because it is easy to form two's complement numbers in binary.
- Note 2^n-N = 2^n-1-N +1 and 2^n-1 = $1111...1_2$
- $1111...1_2 - N$ inverts all bits in N (swapping 0s and 1s), *e.g.*:

```
 11111111
-01010110
_________   (no borrows)
 10101001
```

- Algorithm for complementing N: **invert bits and add 1.**
- Example: 0110_2 → invert → 1001_2 → add 1 → 1010_2

Properties of two's complement numbers

- The two's complement system is a true complement system, in that X+(-X) = 0 (the digit 2^n leads to a carry-out, which is ignored).

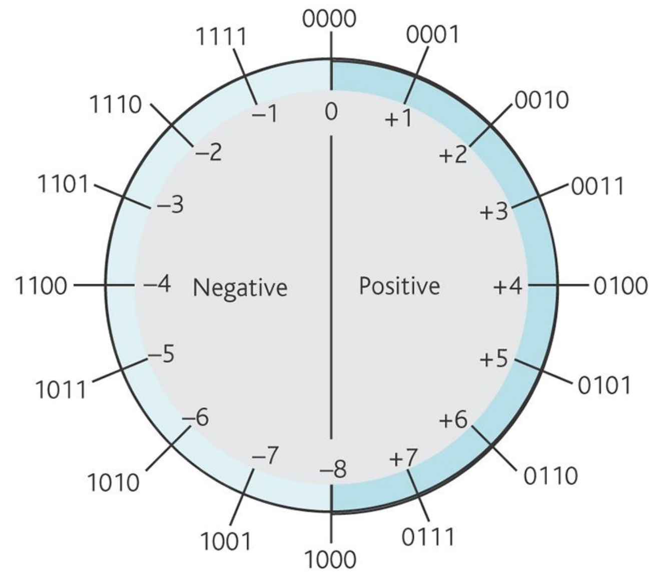
Show that this is true for the 4-bit number 0011_2

- There is one unique zero.
- If the number is positive the most significant bit is 0, and if it is negative the most significant bit is 1.
- The range of two's complement numbers in *n* bits is from -2^{n-1} to $+2^{n-1}-1$. For an 8-bit word, this range is -128 to +127.
- It holds that --X = X.

Show that this is true for the 4-bit number 0101_2

Decimal to two's complement

- How to convert a decimal number to two's complement representation, assuming it's in range?
- Positive numbers (e.g. +6):
  1. Convert to binary, done.
- Negative numbers (e.g. -6):
  1. Invert sign to get +6.
  2. Convert +6 to binary.
  3. Take two's complement in binary (invert, add 1). Yields binary representation of -6.



	decimal	binary
Positive	+6	0110_2
Negative	-6	1010_2

Addition in two's complement system

- Adding X, Y in n-bit two's complement system:
  1. **add bit strings bit by bit.**
     - that's it! Works in the same way for positive and negative Y. (Unless an overflow occurs, see later)
- Examples for 5-bit numbers:

$$(+9)+(+4) = 01001_2 + 00100_2 = 01101_2 \quad (13_{10})$$

$$(+9)+(-4) = 01001_2 + 11100_2 = 00101_2 \quad (5_{10})$$

- Note that any carry out after adding the most significant bits is always ignored. Spelling the above calculation out,

$$9 + (-4) = 9 + (2^5-4) = (9-4) + 2^5 = 9-4 \pmod{2^5}$$

Think of 2^5 or 2^n as **extra loop round the clock**.

Subtraction in two's complement

- Take the **two's complement of Y**, which represents –Y.
- Add X + (– Y) bit-wise as before.
- Disambiguation:

Two's complement representation	**Two's complement of a number**
A system for representing positive and negative numbers.	The negative of a number, e.g. two's complement of 6_{10} represents -6_{10} (and two's complement of -3_{10} represents 3_{10})
Analogy: $\mathbb{Z}$ = {... -2, -1, 0, 1, 2, ...}	Analogy: minus sign, 6 → -6

Arithmetic overflow

- Add the 5-bit two's complement numbers 12 and 13:

  $01100_2 + 01101_2 = 11001_2$ (-7_{10} two's complement)

- We expected the answer 25.
- Note that 11001_2 is 25 if we interpret it as unsigned binary, but if using two's complement then all numbers must be interpreted in the same way!
- An **arithmetic overflow** has occurred.
- Arithmetic overflow occurs in two's complement when addition of two positive numbers gives a negative result, or addition of two negative numbers gives a positive result.

Detecting arithmetic overflow

- Consider addition on a bit-by-bit basis.
- We compare the carry-in C_{in} to the last stage (most significant bit, MSB) and the resulting carry-out C_{out} from that stage.
- If $C_{in} = C_{out}$ then there is no overflow.
- If $C_{in} \neq C_{out}$ then overflow has occurred.
- This check can be done easily in circuits (see later)

Example: no overflow

	0	0	1	1	0	Carry-in to MSB is 0
	0	0	1	1	1	Carry-out from MSB is 0
	0	1	1	0	1	
0	**0**	1	1	0	←	**Carry**

Example: overflow

	0	1	1	0	0	Carry-in to MSB is 1
	0	1	1	0	1	Carry-out from MSB is 0
	1	1	0	0	1	
0	**1**	1	0	0	←	**Carry**

Summary

- Adding unsigned binary numbers works like in decimal.
- Two's complement **representation**: an elegant representation of positive and negative numbers in binary.
- Addition in two's complement works bit by bit, regardless of the signs of operands.
- Subtraction can be done by adding the two's complement of the subtrahend: $X+(2^n\text{-}Y)$, ignoring carry-out of 2^n.
- The two's complement **of a number** X gives –X. Can be done in decimal: $-X = 2^n-X$ or in binary: invert bits and add 1.
- **Arithmetic overflow** can occur in two's complement representation, but is straightforward to detect.