Computer Science Fundamentals

Number Systems – Binary Addition & Subtraction, Complement

Technische Hochschule Rosenheim
Winter 2021/22
Prof. Dr. Jochen Schmidt

Overview

- Basic logical operations
- Binary addition
- Representation of negative integers by complements

Basic Logical Operators

- In computers, logical operations are performed bitwise

- Three basic logical functions
  - Logical AND (**conjunction**)
  - Logical OR (**disjunction**)
  - Logical NOT (**negation/inversion**)

- All other logical operations can be derived from these basic functions

Defined using truth tables

		AND	OR	NOT
a	b	$a \land b$	$a \lor b$	$\neg a$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Exercises

- $$\begin{array}{rl} & 10011 \\ \vee & 10101 \\ = & \end{array}$$

- $$\begin{array}{rl} & 10011 \\ \wedge & 10101 \\ = & \end{array}$$

- $$\begin{array}{rl} \neg & 10101 \\ = & \end{array}$$

Logical Operators – XOR

- another important logical function is the exclusive OR (XOR)
- $a \text{ XOR } b = (a \wedge \neg b) \vee (\neg a \wedge b)$

truth table

a	b	$a \text{ XOR } b$
0	0	0
0	1	1
1	0	1
1	1	0

Binary Addition

- Rules for the addition of two binary digits
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 0 carry 1
  - 1 + 1 + 1 (carry) = 1 carry 1

- Identical to the rules of logical XOR plus carry-over!

Binary Addition

- Example: Add the numbers 11 and 14 using binary arithmetic

- Calculation:

	1 0 1 1
	+ 1 1 1 0
carry	1 1 1
result	1 1 0 0 1

Exercises

- Calculate the sum of the following numbers in binary arithmetic
  - 45 and 54
  - 151.875 and 27.625

Binary Subtraction

- Rules for the subtraction of two binary numbers

  - 0 – 0 = 0
  - 1 – 1 = 0
  - 1 – 0 = 1
  - 0 – 1 = 1   carry –1

- We could do that in the computer – but we don't!

Binary Subtraction

- How are negative numbers represented?
  - Usually by their absolute value preceded by a minus sign

- Is this representation also conceivable in a computer?
  - Yes, but
    - Separate sign calculation would have to be carried out
    - Requires an arithmetic unit that can do both, add and subtract

- Is there a way to get by with addition only?
  - Reduce subtraction to addition: complement representation

Method of Complements

- Two types of complement formation, where B is the basis of the numeral system
  - B-Complement and
  - (B-1)-Complement

- For base 2 we have
  - Two's Complement (*Zweierkomplement*) and
  - Ones' Complement (*Einerkomplement*)

- B-complement (i.e., two's-complement) is more common nowadays

- Using complement requires the number of bits used to be fixed!
  - as this is always the case in a computer, this is not a drawback

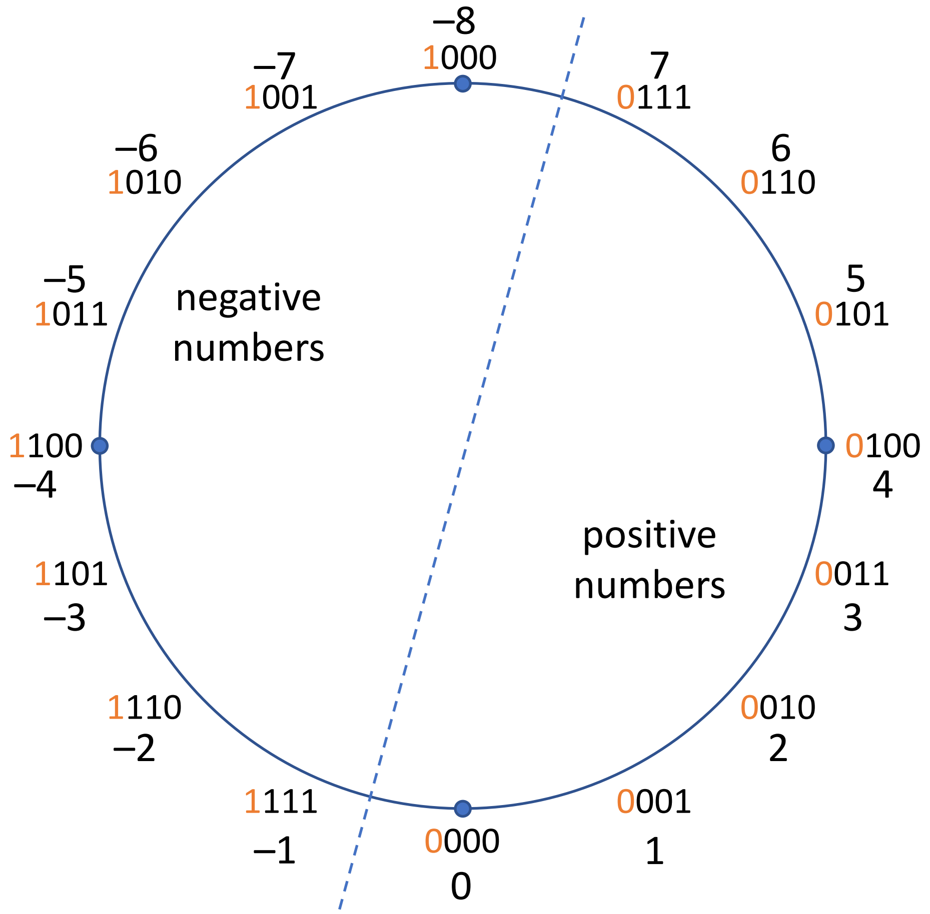
- Complement representation is used only for integers
  - Floating-point representation → later

Two’s Complement: 4 Bit Example

Two’s Complement = B-complement with base 2

```
0000 = 0        1000 = -8
0001 = 1        1001 = -7
0010 = 2        1010 = -6
0011 = 3        1011 = -5
0100 = 4        1100 = -4
0101 = 5        1101 = -3
0110 = 6        1110 = -2
0111 = 7        1111 = -1
```

leftmost Bit (= most significant bit, MSB)
0 ⟶ positive number
1 ⟶ negative number



BUT: This is not a sign-bit in the sense of sign/value-notation!

How to Obtain the Two’s Complement

- positive integers: conversion decimal ⟷ dual as discussed before
  - but with a fixed number of bits, i.e., leading zeros

- negative integers – conversion decimal ⟶ dual
  1. convert the corresponding positive decimal to dual with fixed width
  2. apply a NOT-operation, i.e., invert all bits
  3. add one

- negative integers – conversion dual ⟶ decimal
  1. apply a NOT-operation, i.e., invert all bits
  2. add one
  3. convert to decimal and add a minus-sign

4 Bit example

+5 =	0101
+5 =	0101
	1010
+1	0001 =
	1011 = –5
	1011 = ?
	0100
+1	0001 =
	0101 = +5
⟶	1011 = –5

Advantage of Complement Representation

A computer does not have to be able to subtract, but can execute any subtraction a – b by adding a and (–b)

Example (Two's Complement)

2 – 4 = 2 + (–4)

```
  0010 =  2
+ 1100 = -4
---------------
  1110 = -2
```

6 – 2 = 6 + (–2)

```
   0110 =  6
+  1110 = -2
---------------
  1|0100 =  4
```

Overflow – gets discarded!

No problem here, as the result is within the representable range.

Two's Complement

Technische Hochschule Rosenheim

Caution:

If the calculation returns a result that is not in the representable number range, then you get an overflow and an incorrect result

Example:
With 5 available bits, we want to perform binary subtraction: $(-9)_{10} - (13)_{10}$

Range: $-2^4 \dots 2^4 - 1 = -16 \dots +15$

$$
\begin{array}{rrcr}
 & (-9)_{10} & : & (10111)_2 \\
+ & (-13)_{10} & : & (10011)_2 \\
\hline
 & (+10)_{10} & : & 1|(01010)_2
\end{array}
$$

The overflow is discarded, but would have been relevant
⟶ Wrong result!

Exercises

Form the corresponding B-complement to the following numbers

- 10101_2
- 785_{10}
- 453_{16}

Exercise

Subtract the following numbers using base 2 with 8 digits and the Two's complement:

$57_{10} - 122_{10}$

Two's complement: Most Negative Number

- The most negative number is special in Two's complement
- that's because of the asymmetry – there is no corresponding positive number

- 4-Bit example: range $-8, \dots, +7$; most negative number: $-8 = 1000$
  - taking the absolute value/changing sign/multiplication by one leads to incorrect result: $-(-8) = -8$
–8 =	1000
invert	0111
add 1	<u>0001</u>
result	1000 = –8
  - similar for division by –1/modulo operation

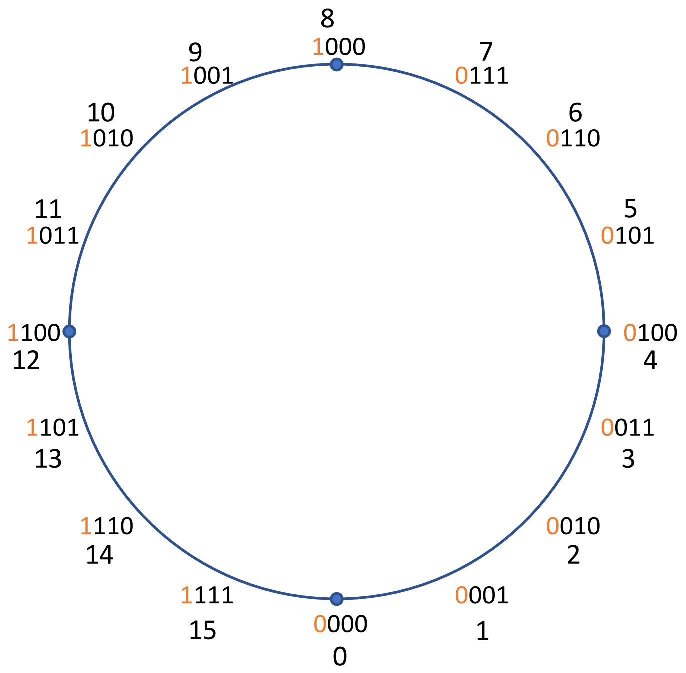
- in standard C/C++, the behavior for the above cases is undefined
  - i.e., **anything** can happen
  - the same is true for an overflow (e.g., adding 1 to the most positive number)
  - this allows for compiler optimizations to take place

Why does Complement Work?

Technische Hochschule **Rosenheim**

- In a computer we use defined data types for numbers
  - e.g., int, long int, unsigned int, …
  - these do have a defined fixed number of bits
- therefore
  - we do not actually use natural numbers or integers
  - but rather modular arithmetic, i.e., division remainders
- algebraically:
  we use a quotient ring (*Restklassenring*) $\mathbb{Z}/n$ (or $\mathbb{Z}_n$)
  - pronounced "$\mathbb{Z}$ modulo n"

- Example: unsigned integers, N Bits
  - $n = 2^N$, range $0, 1, \dots, 2^{N-1}$ $\longrightarrow$ mod 2^N
  - 8 Bits: range $0, 1, \dots, 255$ $\longrightarrow$ mod 256
  - 4 Bits: range $0, 1, \dots, 15$ $\longrightarrow$ mod 16



Why does Complement Work?

- as we use only remainders of division (modulo) by 2^N
  - we can add/subtract multiples of the modulus 2^N without changing anything
    - 4 Bits/mod 16:

$$
\begin{gathered}
0 \equiv 16 \equiv 32 \equiv 48 \equiv \cdots \\
0 \equiv -16 \equiv -32 \equiv -48 \equiv \cdots \\
1 \equiv 17 \equiv 33 \equiv 49 \equiv \cdots \\
1 \equiv -15 \equiv -31 \equiv -47 \equiv \cdots \\
15 \equiv 31 \equiv 47 \equiv 63 \equiv \cdots \\
15 \equiv -1 \equiv -17 \equiv -33 \equiv \cdots
\end{gathered}
$$

  - we can choose an arbitrary range of 2^N integers
    - 4 Bits/mod 16: instead of range $0, 1, \dots, 15$ we can, e.g., use the range $-8, \dots, 0, \dots, 7$
    - 4 Bits/mod 16: Want to know what $-3 \bmod 16$ would be in the standard range $0, 1, \dots, 15$?
      Just add 16 as many times as required: $-3 \bmod 16 = (-3) + 16 \bmod 16 = 13$
      - convert 13 to dual – you get the Two's complement
      - so, the Two's complement of a number is just the difference of the modulus and the desired negative number

- Inversion/adding a one – just a fast way to compute the difference to the modulus 2^N
  - Inversion = difference to $2^N - 1$ (e.g., difference to $2^4 - 1 = 15$: $15 - 3 = 12$)
  - add 1 to get the difference to 2^N (e.g., $12 + 1 = 13 \equiv -3$)

- ignoring an overflow is just reduction mod 2^N

Ones' Complement: 4 Bit Example

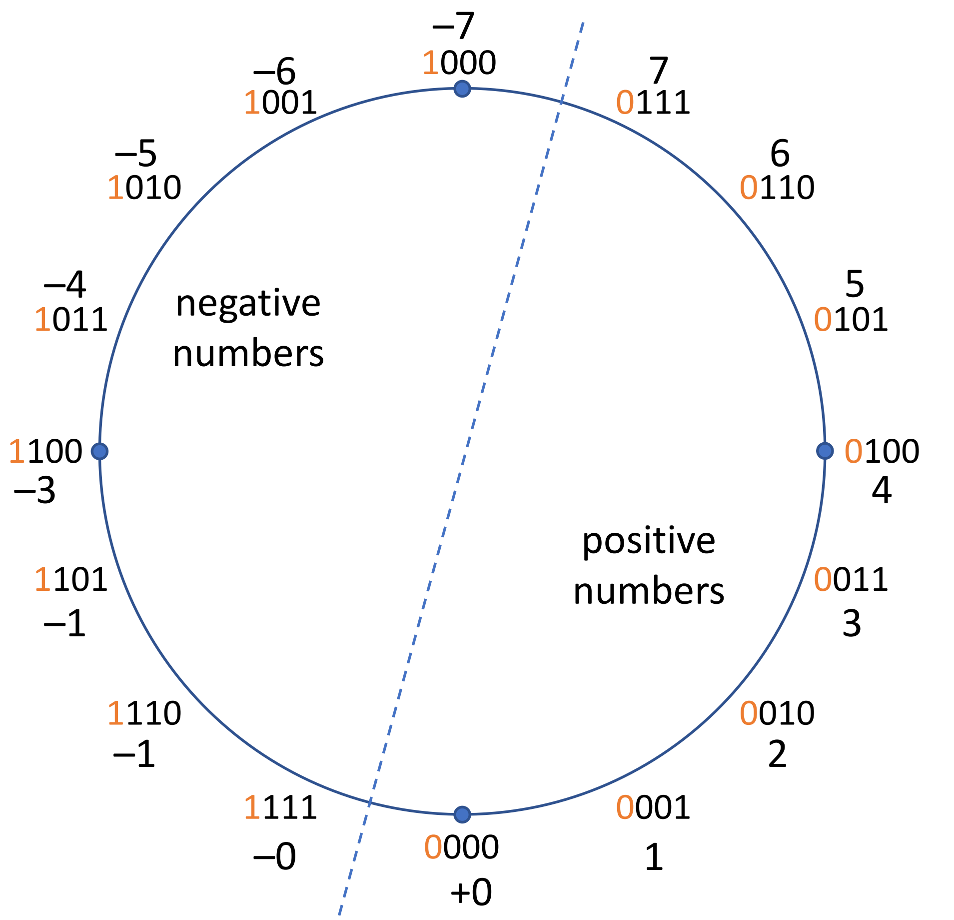
Ones' complement = (B–1)-complement with base 2

```
0000 = +0      1000 = -7
0001 =  1      1001 = -6
0010 =  2      1010 = -5
0011 =  3      1011 = -4
0100 =  4      1100 = -3
0101 =  5      1101 = -2
0110 =  6      1110 = -1
0111 =  7      1111 = -0
```

symmetric – modular arithmetic mod $2^N - 1$,
but we have a positive as well as a negative zero!

leftmost Bit (= most significant bit, MSB)
0 ⟶ positive number
1 ⟶ negative number

BUT: This is not a sign-bit in the sense of sign/value-notation!



How to Obtain the Ones' Complement

- positive integers: conversion decimal ⟷ dual as discussed before
  - but with a fixed number of bits, i.e., leading zeros

- negative integers – conversion decimal ⟶ dual
  1. convert the corresponding positive decimal to dual with fixed width
  2. apply a NOT-operation, i.e., invert all bits

- negative integers – conversion dual ⟶ decimal
  1. apply a NOT-operation, i.e., invert all bits
  2. convert to decimal and add a minus-sign

4 Bit example

+5 = 0101

+5 = 0101

1010 = –5

1010 = ?

0101

0101 = +5

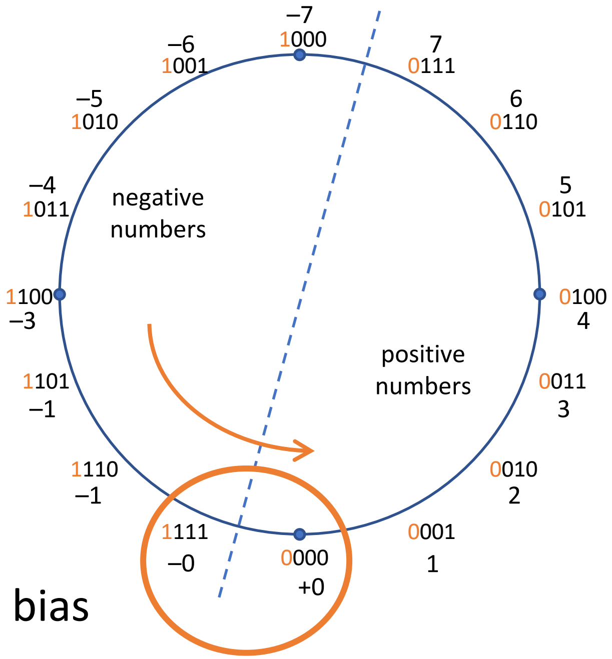
⟶ 1010 = –5

Seems much simpler than Two's complement? Well, there's a catch…

Adding Numbers Using Complements

- Two’s complement: Overflow is always discarded
  - as long as the result is within range, it will be correct

- Ones’ complement: Overflow is wrapped around
  - **if** there is an overflow, this “one” is added to the rightmost bit
  - this is called end-around carry (*Einerrücklauf*)
  - as with the Two’s complement, the result has to be in range to be correct



Why an end-around carry in Ones’ complement?
- an overflow happens, when crossing zero
- but we have two zeros: add one to correct for that bias

Ones’ Complement – Examples

We subtract the following numbers using base 2 with 5 digits and Ones’ complement:

$14_{10} - 7_{10}$

14_{10} =	01110_2	14	01110
7_{10} =	00111_2	+ (–7)	11000
Ones’ complement of 7_{10} (i.e., –7) =	11000	=	1 | 00110
		+1 (end-around carry)	00001
		= +7	00111

$9_{10} - 13_{10}$

9_{10} =	01001_2	9	01001
13_{10} =	01101_2	+ (–13)	10010
Ones’ complement of 13_{10} (i.e., –13) =	10010	= –4	11011

Exercise

Subtract the following numbers using base 2 with 8 digits and the Ones' complement:

- $43_{10} - 11_{10}$
- $17_{10} - 109_{10}$

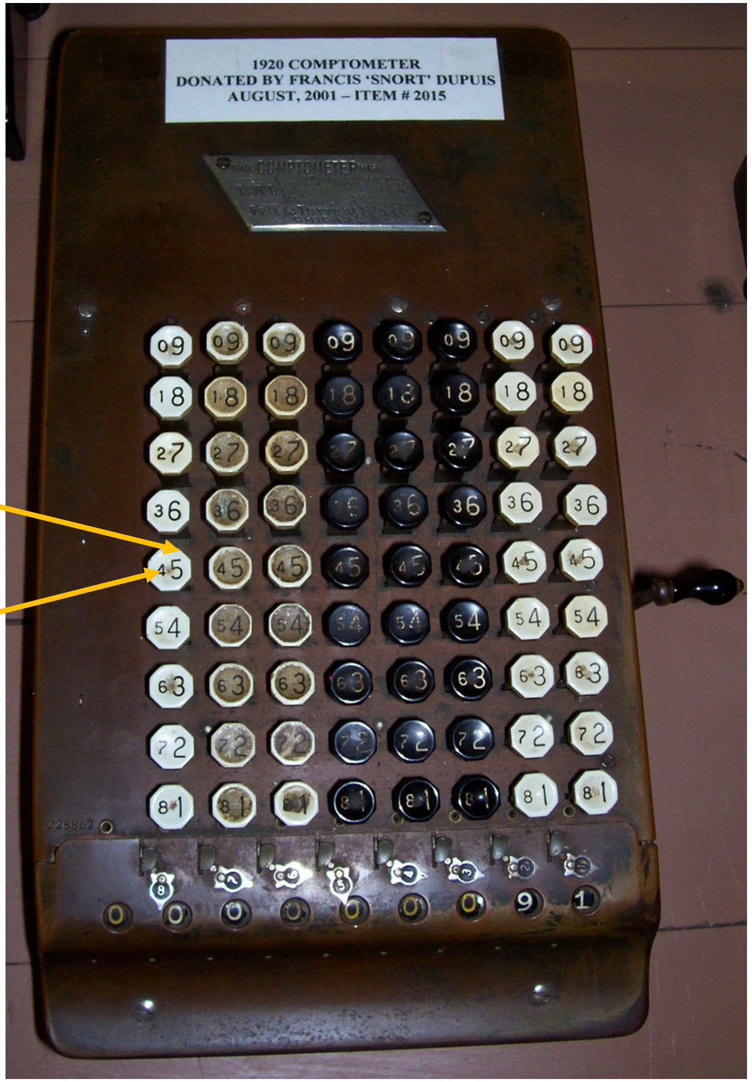
Historical Example: Comptometer

Uses base 10

- calculations executed using 10-Complement
- Input of numbers using 9-Complement

Large numbers
Labels for addition

Small numbers
Labels for subtraction
(as 9-complement)



Concluding Remarks

- complements also work for fixed-point arithmetic
  - omit the point, i.e., treat number as integer
  - convert/perform arithmetic operations
  - re-insert point at correct position
  - float/double data types are not fixed-point – complement is not used here

- when extending the length (in bit) of a data type in complement representation, the leftmost bit must be repeated
  - unsigned integer (4 Bit to 8 Bit): +6 = 0110 ⟶ 0000 0110
    +9 = 1001 ⟶ 0000 1001
  - signed integer, two's complement (4 Bit to 8 Bit): +6 = 0110 ⟶ 0000 0110
    −7 = 1001 ⟶ 1111 1001

- when we see 1111 1001 in memory, how do we know which representation (data type) it is? unsigned integer, ones' or two's complement, fixed-point, floating-point, ASCII, …?