Chapter 2 – Binary Arithmetics

1 Adding Binary Numbers

How can we perform arithmetic operations in binary ?

Idea just like in decimal, we compute the sum column by column, from right to left.

Addition table:

- 0+0 = 0
- 1+0 = 0+1 = 1
- 1+1 = 10. We write the zero and **carry** the one to the next column.

Exercise Convert both numbers to binary, perform addition, then convert sum back to decimal.

- 6 + 5
- 7 + 13
- 25 + 25

Idea On paper we write as many digits as is necessary, but computers store numbers on a fixed number of bits. During an addition, it may happen that a carry bit reaches a position beyond the leftmost column. In that case the computer will simply ignore the carry, and produce an "incorrect" result. This situation is called an **arithmetic overflow**.

Exercise Write both numbers in binary on 8 bits, and perform addition on 8 bits.

- 150 + 150
- 127 + 129

2 Encoding Negative Numbers in Two's Complement

How can we represent negative numbers just with bits ?

Idea Analogy with how we read time: "8:40" may be read "eight fourty" or "twenty to nine". In other words, if we discard the "hour" column and only keep the "minutes", then "+40" and "−20" are the same.

Most importantly, addition still works (if we keep discarding the "hour" column)

- example 1: adding +10 to +40 aka −20 yields +50 aka −10
- example 2: adding +30 to +40 aka −20 yields 1h10 i.e. +10

The same idea works in binary: one byte can either represent numbers from 0 to 255, or from −128 to 127, depending on how we (humans) decide to interpret the bit pattern. These conventions are known as **unsigned** and **signed** interpretations.

Example with numbers encoded on four bits:

bit pattern	signed decimal	unsigned decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Remarks

- Many properties of binary numbers remain true in two's complement. For instance the addition algorithm only cares about bits (not number values) and it works just as well for signed numbers.
- However, in two's complement, "writing a zero on the left" of a number may change its value, so we must always specify how many bits we're working with.

Exercise Perform these additions in two's complement on 4 bits (remember that you have only 4 columns to work with: any bits that would be carried to the fifth column are simply discarded)

- $5 + 2$
- $5 + 5$
- $-3 + 5$
- $-1 + 1$
- $-2 + 2$
- $-7 + 7$

3 Sign Extension

How to add two numbers represented on different widths ?

Idea We can increase the number of bits of number while preserving its sign and value: this is done by duplicating its **sign bit** (i.e. leftmost bit in two's complement representation) enough times.

For instance, on 4 bits -3 is encoded as **1**101 (sign bit in bold). To get the 8 bit encoding of -3 we just have to duplicate this "1" bit four times: **1111**1101 (new bits in bold).

Positive numbers have a sign bit of "0", so extending those is just writing more zeroes on the left.

Exercise Work out the two's complement representation of -1 on 4 bits, then on 8 bits, then on 32 bits.

4 Subtraction: going from positive to negative and vice versa

"Subtracting x" is the same thing as "adding $-x$", but how to get one from the other ?

Idea In two's complement, it is possible to flip the sign of a number while keeping its (absolute) value. There are two methods, which (obviously) produce the same result. For example, working with eight bits, let's start with number 44 = 0b00101100 and work out -44.
Method A:

- Starting from the right, find the first "1", for instance: 00101**1**00
- Invert all the bits to the left of that "1", for instance: **11010**100

Method B:

- Perform a **bitwise** negation i.e. invert each bit, for instance: 00101100 $\rightarrow$ 11010011
- Then add one, discarding any carry bit beyond the nth column: 11010011 + 1 $\rightarrow$ 11010100

Exercise Work out the two's complement encoding of -1 on 8 bits using method A, then method B.

Exercise Using your favorite method, write the two's complement encoding on 8 bits of "-0".

Exercise Compute $112 - 54$

Exercise Still in two's complement on 8 bits, compute $127 - 127$.

5 Multiplication

Addition and subtraction are cool, but how about other operations ?

Idea Arithmetic operations work the same in binary as they do in decimal. We just have to restrict everything to base 2.

Below is an example multiplication of two (unsigned) numbers A and B encoded on 4 bits:

					A_3	A_2	A_1	A_0	first operand A
				$\times$	B_3	B_2	B_1	B_0	second operand B
					A_3B_0	A_2B_0	A_1B_0	A_0B_0	partial product $A\times B_0$
+				A_3B_1	A_2B_1	A_1B_1	A_0B_1		$A\times B_1$ shifted left
+			A_3B_2	A_2B_2	A_1B_2	A_0B_2			$A\times B_2$ shifted left twice
+		A_3B_3	A_2B_3	A_1B_3	A_0B_3				$A\times B_3$ shifted by three columns
=	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0	result = sum of partial products

Remarks

- Just like in decimal, multiplying two n digits numbers may produce a result on $2n$ digits.
- The standard algorithm is actually simpler in binary than in decimal, as it involves no "multiplication table": each partial product P_i is either zero, or just A shifted by i positions if B_i is "1".
- Multiplying negative numbers is a bit more complicated and requires a few changes in the algorithm (e.g. sign extensions of partial products) but the idea is the same.

Exercise In decimal, compute the multiplication of 1337 by 42.

Exercise In binary, compute these multiplications: 6×7, 12×12.