Decimal and binary representation systems

- They both are *positional* representation systems
- In decimal, numbers are represented by the coefficients of the powers of 10
  - Example: $321 = 3 . 10^2 + 2 . 10^1 + 1. 10^0$
- Decomposing 321 in powers of 2 yields $321 = 256 + 64 + 1$
  - or $1. 2^8 + 0. 2^7 + 1. 2^6 + 0. 2^5 + 0. 2^4 + 0. 2^3 + 0. 2^2 + 0. 2^1 + 1. 2^0$
  - or 101000001_2

Positional number systems

- More generally, a positive integer is represented in
  - decimal by $\sum_{i=0}^{n} a_i \times 10^{n-i}$ where $0 \le a_i \le 9$
  - binary by $\sum_{i=0}^{m} b_i \times 2^{m-i}$ where $0 \le b_i \le 1$

Binary and Hexadecimal representation systems

- Writing binary numbers quickly becomes error-prone and unwieldy
- Instead use hexadecimal system, positional representation system in base 16
  - Example: $321 = 1 \, .16^2 + 4 \, .16^1 + 1 \, .16^0 = 141_{16}$
- Since the coefficients are between 0 and 15, we need new symbols to represent 10 through 15. They will be A through F
  - A (hex) = 10 (decimal) = 1010 (binary)
  - B (hex) = 11 (decimal) = 1011 (binary) ...
  - F (hex) = 15 (decimal) = 1111 (binary)

Conversion between binary and hexadecimal

- Group *bits* (abbreviation for binary digits) by groups of four, starting from the right (*least significant bit* or *lsb*)
  - Example: 101000001 = 1 0100 0001 (binary) = 141 (hex)
  - 111001011 = 1 1100 1011 (binary) = 1CB (hex)
  - Note that the greatest magnitude bit , the leftmost one, is called (*most significant bit* or *msb)*
- Why hexadecimal?
  - Very convenient to represent strings of 4, 8, …,16,…32, …64 bits by 1,2,..,4,…8….16 hex digits

Some useful powers of two

$$2^{10} = 1024_{10} \approx 10^3 = 1K(Kilo)$$

$$2^{20} \approx 10^6 = 1M(Mega)$$

$$2^{30} \approx 10^9 = 1G(Giga)$$

- We'll often round-off and talk about, say, 16 KB or 64 MB

Representing positive and negative integers

- In an *n*-bit register, you can represent 2^n patterns
  - Example: in a 32-bit register, we can represent *unsigned integers* in the range $[0:2^{32}-1]$
- How to represent positive **and** negative integers with the following properties:
  - Equal number of positive and negative numbers
  - Unique (and easily testable) representation of zero
  - Easy sign test
  - Easy rules for addition and subtraction

Three representation systems

- Historically, 3 different numbering systems have been used
  - **Two's complement** now used in all machines for integer representation
  - sign and magnitude used (partially) for floating-point representation
  - One's complement (very similar to 2's complement but with a few more drawbacks)

Two’s complement representation

- Positive numbers as unsigned binary with msb always 0
- Zero is represented as a string of 0’s
- To represent a negative number:
  - Consider the representation of its absolute value
  - Flip all 1’s to 0’s and 0’s to 1’s (this is 1’s complement)
  - Add 1 to lsb using binary arithmetic rules

2’s complement

- Example assuming a 4-bit register
  - What is the representation of (decimal) 6?
  - What is the representation of (decimal) -6?
  - What is the representation of 0?
  - What is the range of representation of positive numbers?
  - What is the range of representation of negative numbers?
  - How do I recognize whether a number is positive or negative or zero?

Addition in 2’s complement

- Addition
  - Perform an ordinary binary addition and discard the carry-out
  - If you add 2 numbers of opposite sign, everything will always be all right
  - If you add two positive numbers and the result *appears* to be negative (i.e., msb = 1) then you have an *overflow*. This will generate an *exception* in your program.
  - If you add two negative numbers and the result *appears* to be positive (i.e., msb = 0) then you have an *underflow.*
- Subtraction
  - Take the 2’s complement of the subtrahend and add it to the other operand