

Lecture 4

Binary Arithmetic

Unsigned Integers

- Memory addresses are unsigned integers.
- If memory address (of a location) is 16 bit, then addresses would run from 0x0000 to 0xFFFF.
- There are no signed or negative addresses.

Addition of Unsigned Integers

- Simply add the bits: $101 + 1001 = 1110$.
- The result may not fit in the allocated number of bits (in this case 4). The extra bit is carried over to a carry bit (if one is allocated).

Addend	0	0	1	1
Augend	<u>+0</u>	<u>+1</u>	<u>+0</u>	<u>+1</u>
Sum	0	1	1	0
Carry	0	0	0	1

Signed Binary Integers

- Positive and negative integers.
- This is how a computer stores or represents integers in a memory word.
- There are 3 types of signed binary integers:
 - A. Signed magnitude
 - B. 1's complement
 - C. 2's complement
- Processor must be able to add or subtract these numbers very easily.

A. Signed Magnitude

- Leftmost bit in the word is the *sign bit*.
 - ‘0’ for positive numbers, ‘1’ for negative numbers.
- Remaining bits hold the absolute value of the number (*e.g.*, 8-bit word: 1 bit for sign, 7 bits for value).

Example: 8-bit numbers. This includes 1 bit for the sign.

01001011 $\rightarrow +75_{10}$ 10001111 $\rightarrow -15_{10}$

Addition: $7_{10} + 12_{10} = 19_{10}$ 0111 + 1100 = 1 0011 The 1 is an extra carry (note that the sign

Subtraction: $9_{10} - 6_{10} = 3_{10}$ 1001 - 0110 = 0011 bit is not shown in these two examples)

- What happens if the result is < 0 ? How do we store negative numbers?
- Problems: There are two representations for zero:
 - A ‘positive’ zero (0 000 0000) and a ‘negative’ zero (1 000 0000).
- Addition of 2 positive numbers is straightforward.
- But subtraction is tricky.
 - Sign of the result of subtraction can be either positive or negative value.
 - The sign depends on the magnitudes of the two numbers involved.

Example: $75 - 15 = 60$ (sign bit is ‘0’) $15 - 75 = -60$ (sign bit is ‘1’)

- First, we have to compare their absolute magnitudes and then decide how to do the subtraction.
- Then, the sign of the result is decided.
- Actual subtraction is then performed
- Subtraction is a slow and complicated process.

B. 1’s complement

- The 1’s complement representation in binary of a **positive** integer is no different from the sign-magnitude representation of that integer.
- Rule: The 1’s complement in binary of a **negative** integer is obtained by subtracting its magnitude from $2^n - 1$, where n is the number of bits used to store the integer in binary.
- This also uses a sign bit, ‘0’ for positive and ‘1’ for negative.
- To negate a number, determine the 1’s complement of the corresponding positive integer and invert all bits.
- Two representations for zero (a ‘positive’ zero and a ‘negative’ zero).
- No longer really used.

- For subtraction, you find the 2's complement of the number and add. Ignore any carry bit.

Example: $7 - 4 = 3$, $0111 - 0100 \rightarrow 0111 + 1100 = 0011$

- How to detect overflow or underflow?

- Overflow: we exceed the limit of positive numbers.
- Underflow: we exceed the limit of negative numbers.

Example: $5 + 4 = 9$, $0101 + 0100 = 1001$ (this is overflow, not a negative number).

Example: $-7 + -6 = -13$, $1001 + 1010 = 1\ 0011$ (this is underflow and not a positive answer).

- Observation: We **may** have an overflow or underflow, if the two numbers have the **same** sign.
We will **never** have over/underflow if the two numbers are of **opposite** sign.
- Overflow rule: If both numbers are positive or both negative then we have an over/underflow if and only if the result has the opposite sign

- Usually, the ALU has extra hardware that monitors sign bits of both numbers and the result.

- An overflow bit is set if it is detected by the hardware.
- There are assembly language instructions which allow you to test this bit:
 - "If overflow, do *this*, else do *that*".

Sign Extension

- During arithmetic operations, processor sometimes needs to increase ('extend') the size or width of the signed number.
- Here we want the 'extended' number to have same sign and the same magnitude as the original.
 - For a positive number: add zeros in the 'extra' bits to the left.
 - For a negative number: add ones in the 'extra' bits to the left.

Example: 8-bit signed number \rightarrow 16-bit signed number.

We want the 'extended' 16-bit number to have same sign and the same magnitude as the original 8-bit number.

$-7_{10} = 11111001$ in 8bits \rightarrow $11111111\ 11111001$ in 16 bits

- An *arithmetic* right shift retains the sign.

N decimal	N binary	-N signed mag.	-N 1's compl.	-N 2's compl.	-N excess 128
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Nonexistent	Nonexistent	Nonexistent	10000000	00000000

- 8- bit numbers.
- In 2's complement, there is -128 but no +128.
- In signed magnitude and 1's complement, there is no -128 and no +128.