

# CSCI 112

## Introduction to computer Science -I

Instructor: Santanu Banerjee

# Integer Arithmetic

# Store a number in memory

To store a number in memory

- If positive number
  - Simply convert to binary
- If negative number, convert into 2's comp by
  - Either
    - Convert the magnitude(only) to binary
    - Flip the bits
    - Add 1
  - OR
    - Convert magnitude(only) to hex
    - Subtract the result from 10...0 (#0's to be equal to #nibbles used for storage)

# Store a number in memory

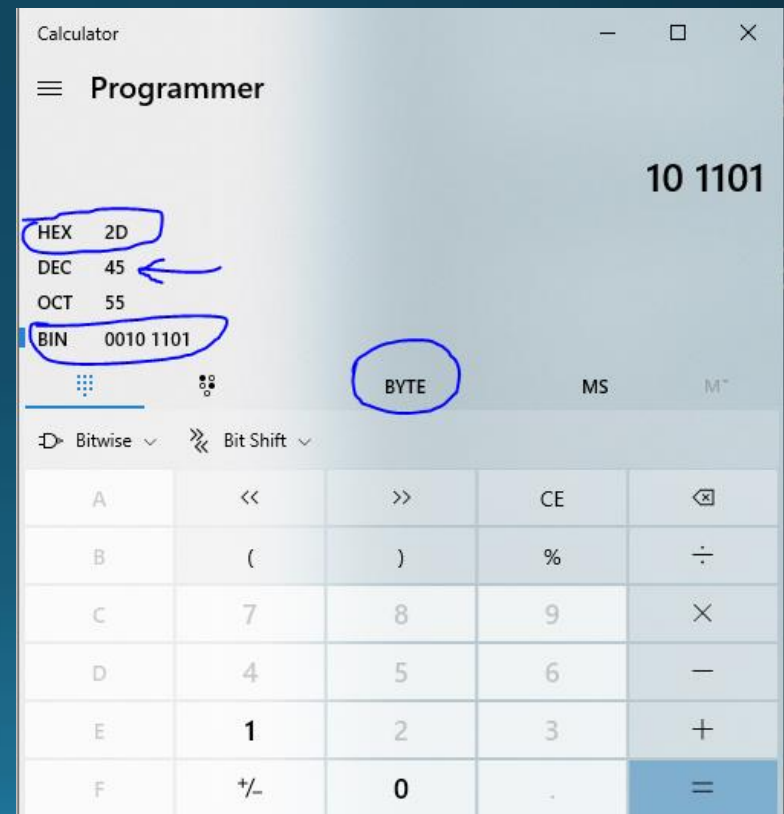
Example: Store 45 in a byte at address 20476

45 => 00101101 or 2D

...	
20473	??
20474	??
20475	??
20476	??
20477	??
20478	??
...	



...	
20473	??
20474	??
20475	??
20476	2D
20477	??
20478	??
...	



# Store a number in memory

Example: Store -285 in a word at address 20490

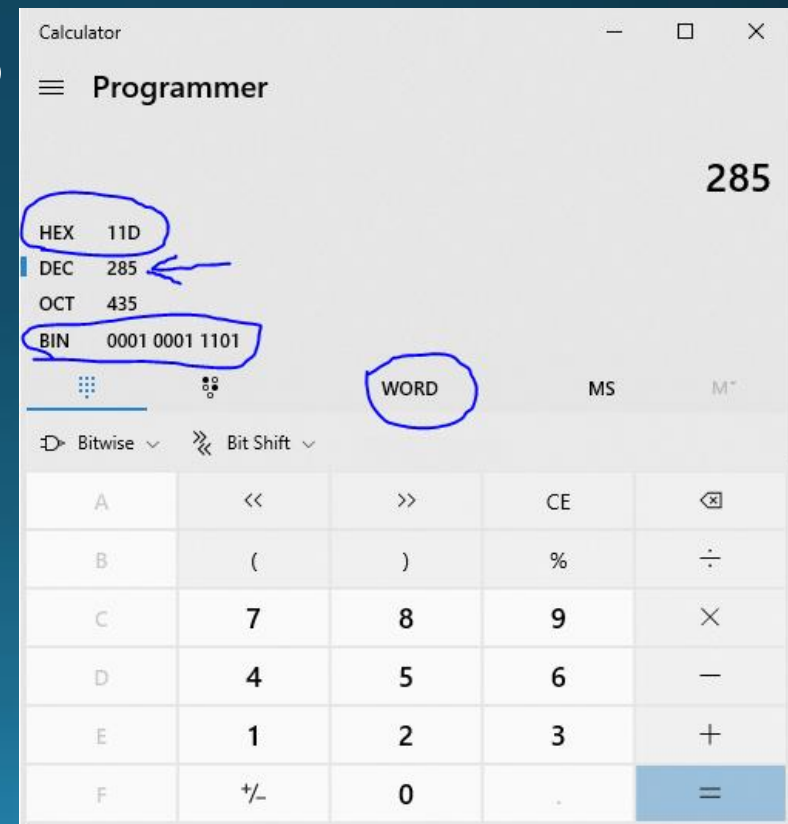
285 => 0000000100011101 or 011D

-285 => 1111111011100011/FEE3

...	
20488	??
20489	??
20490	??
20491	??
20492	??
20493	??
...	



...	
20488	??
20489	??
20490	E3
20491	FE
20492	??
20493	??
...	



# Interpret a number from memory

To interpret a number from memory

- If leftmost bit is set, convert into 2's comp by
  - Either
    - Flip the bits
    - Add 1
  - OR
    - Convert to hex
    - Subtract the result from 10...0 (#0's to be equal to #nibbles used for storage)
  - Convert the result from either step to the target number system and use a negative sign.
- Else (i.e. leftmost bit is not set)
  - Simply convert to the target number system

# Interpret a number from memory

Example: Read the byte size number at address 20476

2D => 45<sub>10</sub>

...	
20473	??
20474	??
20475	??
20476	2D
20477	??
20478	??
...	

# Interpret a number from memory

Example: Read the word size number at address 20490

E3 FE (in LE)  $\Rightarrow$  FE E3  $\Rightarrow -285_{10}$

Because:  $10000 - FEE3 = 285_{10}$

...	
20488	??
20489	??
20490	E3
20491	FE
20492	??
20493	??
...	



# Addition

- Same for unsigned and 2's complement signed numbers, but the results may be interpreted differently
- The two numbers will be byte-size, word-size, doubleword-size, or quadword-size.
- Add the bits and store the sum in the same length as the operands, discarding an extra bit (if any).

# Subtraction

- Take the 2's complement of second number and add it to the first number.
- Standard rules for addition will apply.

# 8-bit Two's Complement Add/Sub

$$54_{10} = 0011 \ 0110$$

$$+ \ 44_{10} = 0010 \ 1100$$

---

$$98_{10} = 0110 \ 0010$$

$$44_{10} = 0010 \ 1100$$

$$+ \ -48_{10} = 1101 \ 0000$$

---

$$-4_{10} = 1111 \ 1100$$

$$54_{10} = 0011 \ 0110$$

$$+ \ -48_{10} = 1101 \ 0000$$

---

$$6_{10} = 0000 \ 0110$$

$$-44_{10} = 1101 \ 0100$$

$$+ \ -48_{10} = 1101 \ 0000$$

---

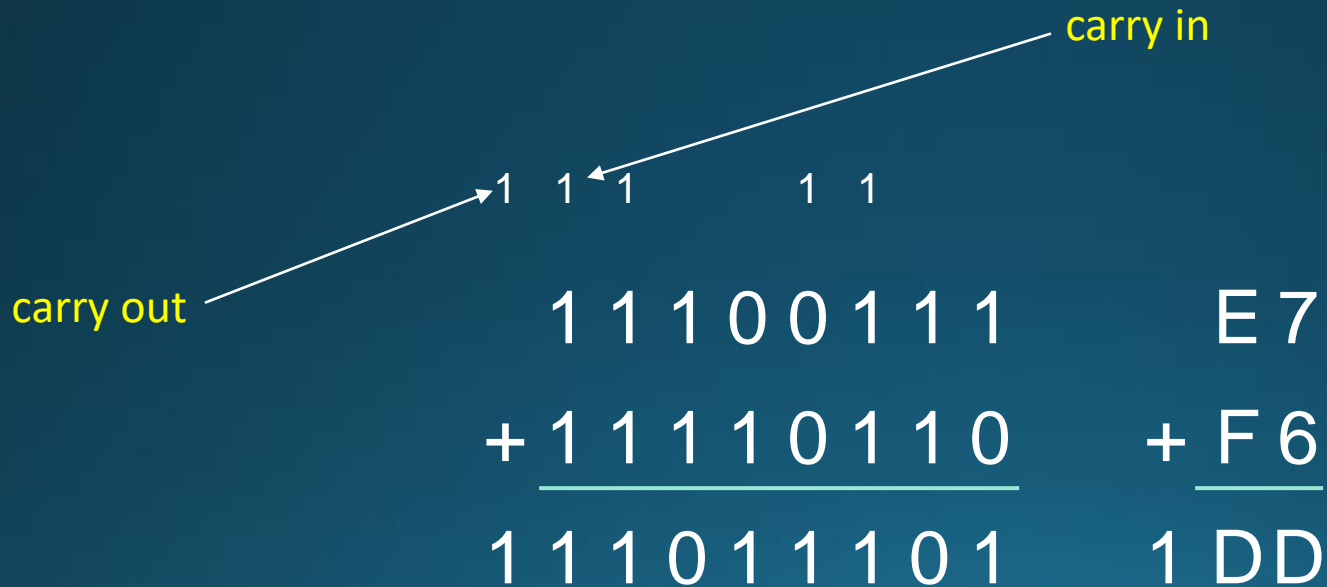
$$-92_{10} = 1010 \ 0100$$

# Carry

- If the sum of two numbers is one bit longer than the operand size, the extra 1 is a *carry* (or carry out).
- A carry is discarded when storing the result, but we'll see how the 80x86 CPU records it.
- For *unsigned* numbers, a carry means that the result was too large to be stored – the answer (as recorded in the available bits) is wrong.

# Carry In

- A 1 carried into the high-order (sign, leftmost) bit position during addition is called a *carry in*.
- Byte-length example



# Overflow

- An overflow occurs if adding two positive numbers yields a negative result or if adding two negative numbers yields a positive result.
- Adding a positive and a negative number never causes an overflow.
- Carry out of the most significant bit does not indicate an overflow.
- Overflow occurs when there is a carry in but no carry out, or when there is a carry out but no carry in. It is recorded by the CPU.
- If overflow occurs when adding two signed numbers, then the result will be incorrect.

# Two's Comp Overflow Examples

- Word-length example( $ci+co \Rightarrow \Theta F$ )

- 0206 is +ve  $\Rightarrow$  518

- FFB0 can be

- Case 1 (signed) : -ve  $\Rightarrow$  -80

- Case 2 (unsigned) : +ve  $\Rightarrow$  65456

0206

+ FFB0

????

Case1

Case 2

0206

518

518

+ FFB0

+ (-80)

+ 65456

101B6

438

65974



# Two's Comp Overflow Examples

Another set of examples (signed interpretation)

- using binary

$(ci + \cancel{co} \Rightarrow OF)$

$$\begin{array}{rcl} 54_{10} & = & 0011\ 0110 \\ +\ 108_{10} & = & 0110\ 1100 \\ \hline 162_{10} & \neq & 1010\ 0010 \end{array}$$

$(\cancel{ci} + co \Rightarrow OF)$

$$\begin{array}{rcl} -103_{10} & = & 1001\ 1001 \\ +\ -48_{10} & = & 1101\ 0000 \\ \hline -151_{10} & \neq & 0110\ 1001 \end{array}$$



# Sign Extension

## **The operation**

Increasing the number of bits of a binary number while preserving the number's sign (positive/negative) and value.


## **Process**

This is done by appending digits to the most significant side of the number, following a procedure dependent on the particular signed number representation used.

# Sign Extension - example

- If 8 bits("0000 1010" ) are used to represent the number decimal positive 10 and the sign extend operation increases the length to 16 bits, then the new representation is simply "0000 0000 0000 1010". Thus, both the value and the fact that the value was positive are maintained.
- If 8 bits("1111 0001" ) are used to represent the value decimal negative 15 using two's complement, and this is sign extended to 16 bits, the new representation is "1111 1111 1111 0001". Thus, by padding the left side with ones, the negative sign and the value of the original number are maintained.

Decimal	8-bit	16-bit
+10	0000 1010	0000 0000 0000 1010
-15	1111 0001	1111 1111 1111 0001



# Two's Comp Sign Extension

Why does sign extension work?

- For positive numbers
  - Adding leading zeros would not change the sign or value
- For negative numbers

-x is represented as  $2^8 - x$  in 8-bit (i.e. in hex,  $100 - x$ )

-x is represented as  $2^{16} - x$  in 16-bit (i.e. in hex,  $10000 - x$ )

$$2^8 - x + ??? = 2^{16} - x$$

$$??? = 2^{16} - 2^8$$

$$1\ 0000\ 0000\ 0000\ 0000 = 65536$$

$$\begin{array}{r} 1\ 0000\ 0000 = 256 \\ - \end{array}$$

$$\hline 1111\ 1111\ 0000\ 0000 = 65280$$

# Subtraction

- Take the 2's complement of second number and add it to the first number.
- Overflow occurs in subtraction if it occurs in the corresponding addition.
- CF is set for a *borrow* in subtraction – when the second number is larger than the first as unsigned numbers. There is a borrow in subtraction when there is *not* a carry in the corresponding addition.

# Multiplication

- Similar to what you do in decimal multiplication.

- Example:

$$\begin{array}{r} 6_{10} = 0110 \\ \times 10_{10} = 1010 \\ \hline 0000 \\ + 0110 \\ + 0000 \\ + 0110 \\ \hline 60_{10} = 0111100 \end{array}$$

# Division

- Similar to what you do in decimal division.
- Example:

