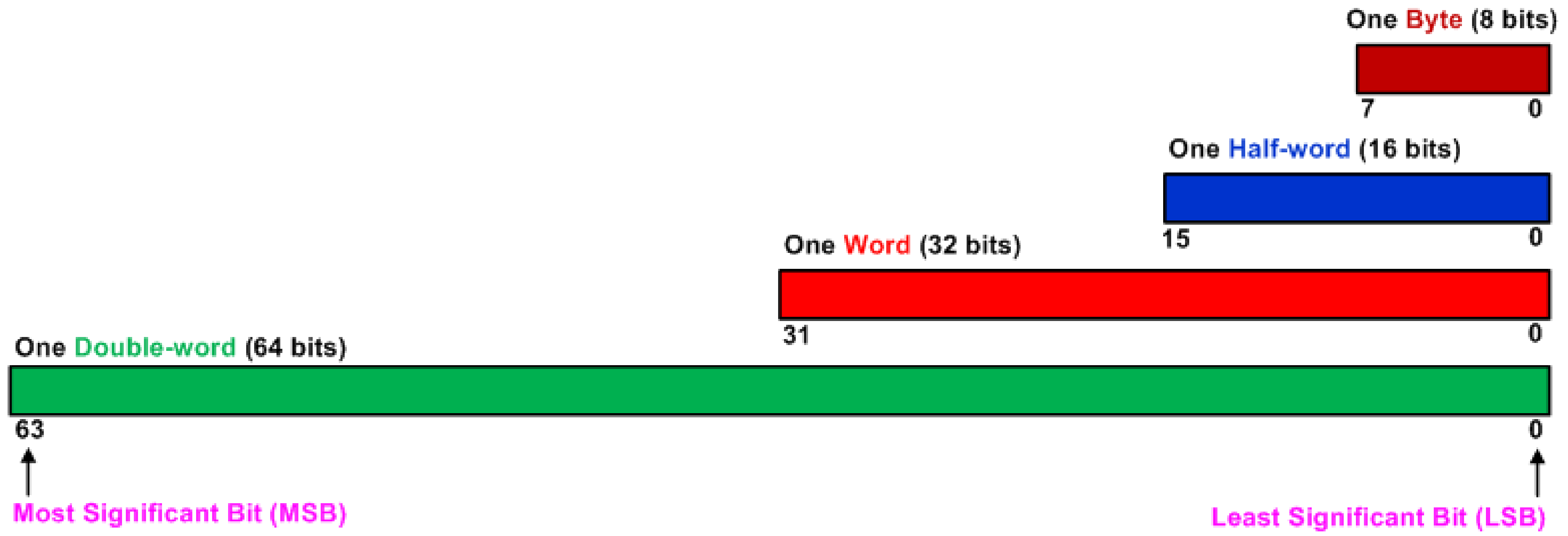
**Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C**

# Chapter 2
# Data Representation

Z. Gu
Hofstra University

Spring 2026

## Bit, Byte, Half-word, Word, Double-Word



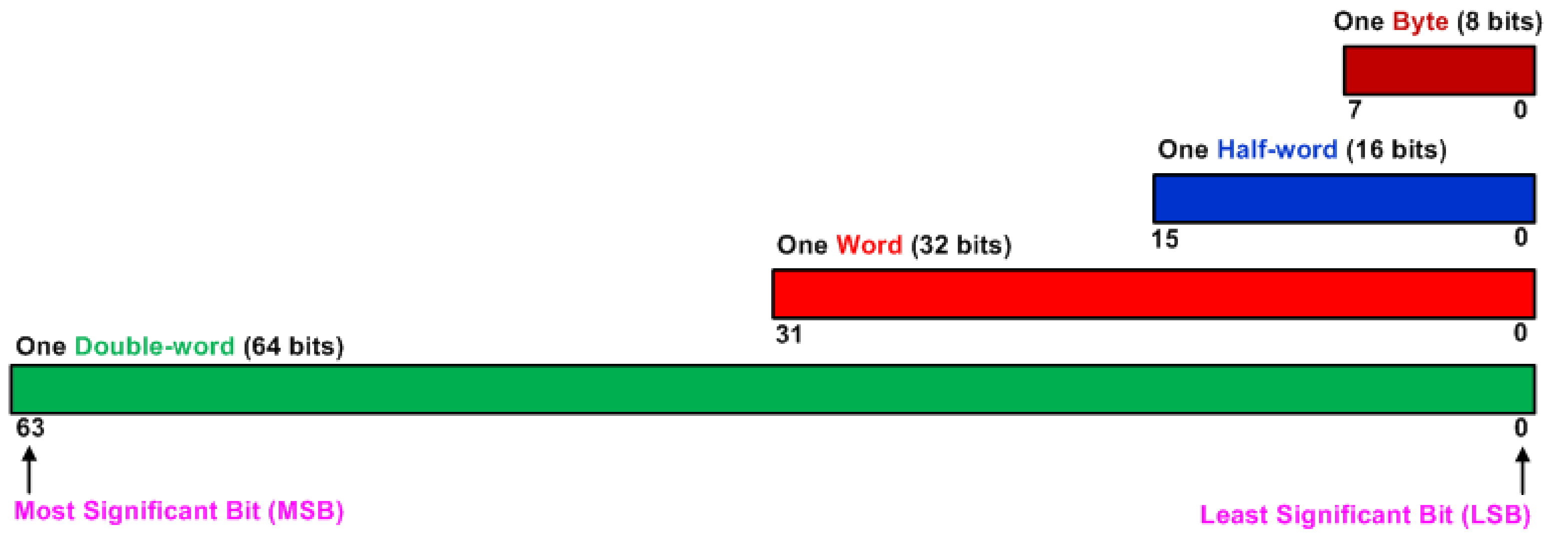
One Byte (8 bits)
7
0
One Half-word (16 bits)
15
0
One Word (32 bits)
31
0
One Double-word (64 bits)
63
0
Most Significant Bit (MSB)
Least Significant Bit (LSB)

## Binary, Decimal and Hex

| Decimal | Binary | Hex |
|---|---|---|
| 0 | 0000 | 0x0 |
| 1 | 0001 | 0x1 |
| 2 | 0010 | 0x2 |
| 3 | 0011 | 0x3 |
| 4 | 0100 | 0x4 |
| 5 | 0101 | 0x5 |
| 6 | 0110 | 0x6 |
| 7 | 0111 | 0x7 |
| 8 | 1000 | 0x8 |
| 9 | 1001 | 0x9 |
| 10 | 1010 | 0xA |
| 11 | 1011 | 0xB |
| 12 | 1100 | 0xC |
| 13 | 1101 | 0xD |
| 14 | 1110 | 0xE |
| 15 | 1111 | 0xF |

0x: Hex

# Range of Unsigned Integers

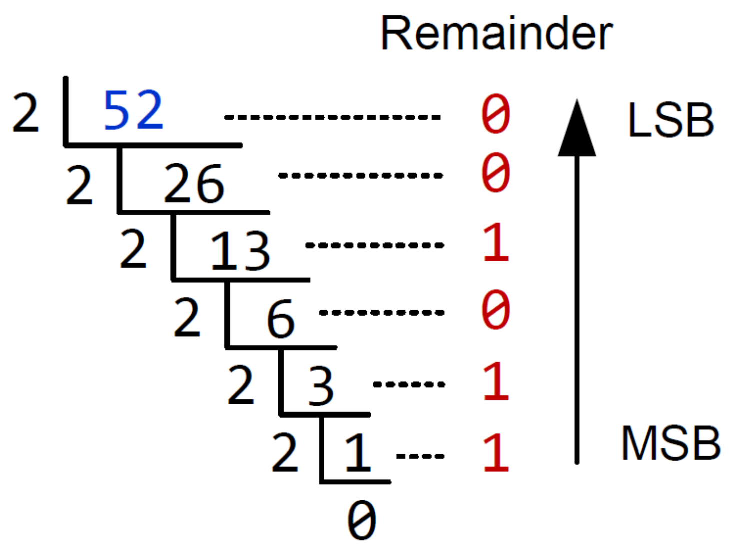

One Byte (8 bits)
7
0
One Half-word (16 bits)
15
0
One Word (32 bits)
31
0
One Double-word (64 bits)
63
0
Most Significant Bit (MSB)
Least Significant Bit (LSB)

| Storage Size | Range | Powers of 2 |
|---|---|---|
| Unsigned Byte | `0 to 255` | `0 to` $2^8$`-1` |
| Unsigned Halfword | `0 to 65,535` | `0 to` $2^{16}$`-1` |
| Unsigned Word | `0 to 4,294,967,295` | `0 to` $2^{32}$`-1` |
| Unsigned Double-word | `0 to 18,446,744,073,709,551,615` | `0 to` $2^{64}$`-1` |

# Unsigned Integers

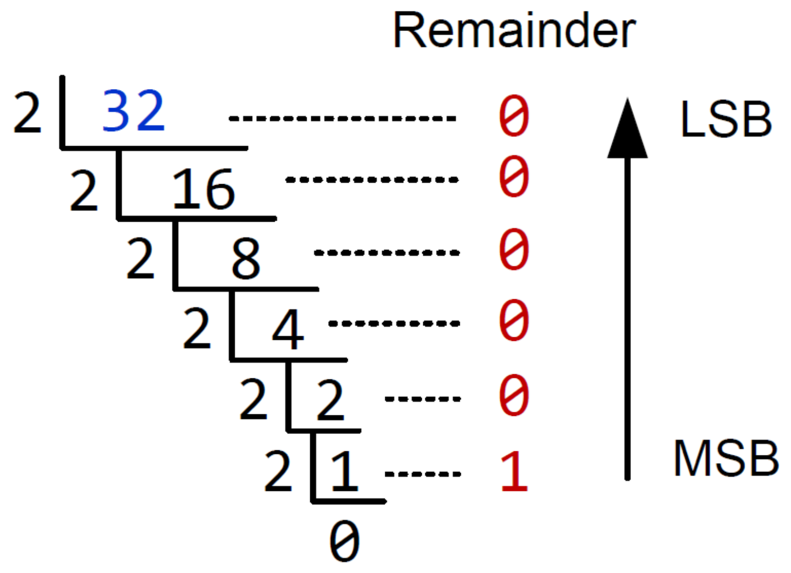
## Convert Decimal to Binary

**Example 1**



**$52_{10}$ = $110100_2$**

**Example 2**



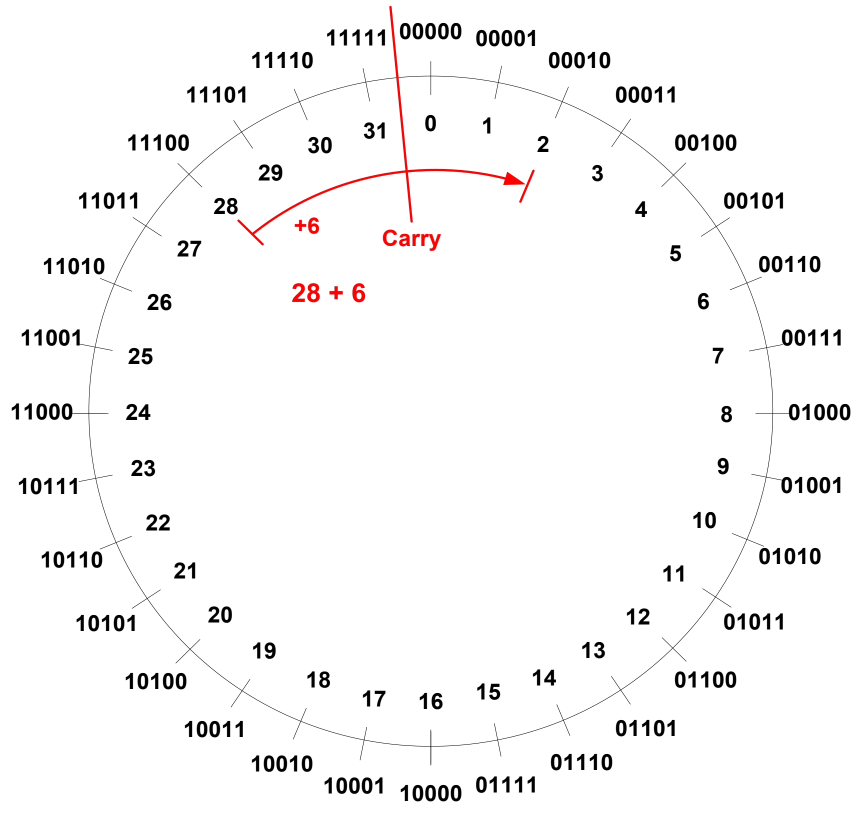
**$32_{10}$ = $100000_2$**

## Carry/borrow flag bit for unsigned arithmetic

- Given unsigned integers $a$ and $b$
- $\boldsymbol{c = a + b}$
  - Carry happens if c is too big to fit in *n* bits (*i*.e., $\mathrm{c} > 2^n - 1$).
- $\boldsymbol{c = a - b}$
  - Borrow happens if $c < 0$.
- On ARM Cortex-M processors, the carry flag and the borrow flag are physically the same flag bit in the status register.
  - For an unsigned subtraction, Carry = NOT Borrow

## Carry/borrow flag bit for unsigned numbers

***If the traverse crosses the boundary between 0 and $2^n - 1$, the carry flag is set on addition and is cleared on subtraction.***



A carry occurs when adding 28 and 6

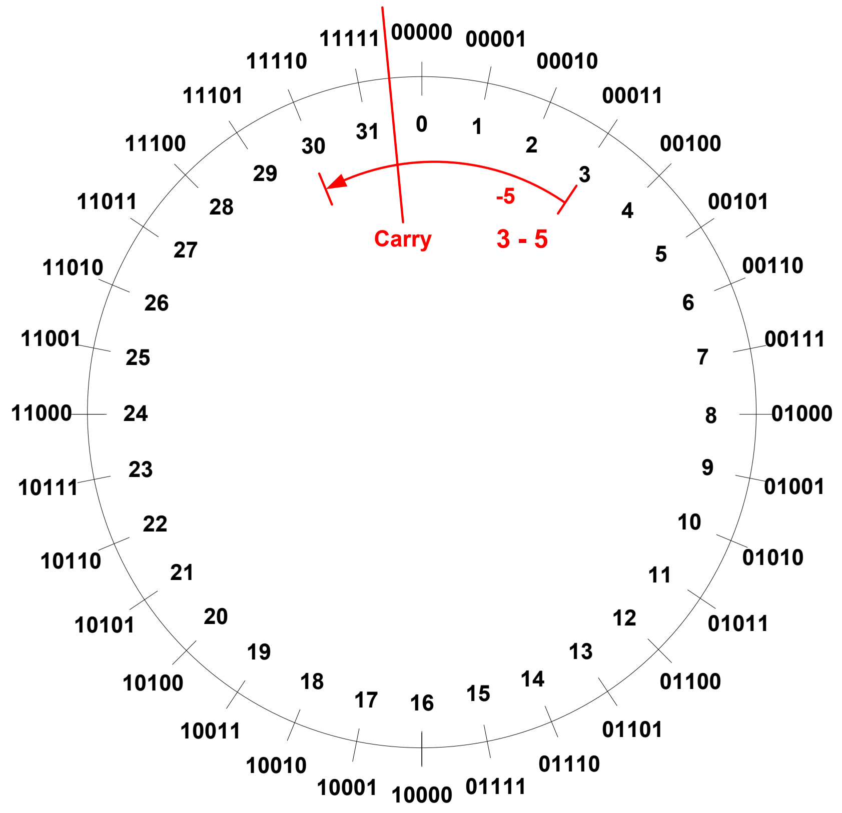
$$\begin{array}{lcccccc} \text{Carry} & 1 & 1 & 1 & 0 & 0 & \\ & & 1 & 1 & 1 & 0 & 0 \\ & + & 0 & 0 & 1 & 1 & 0 \\ \hline & \boxed{1} & \multicolumn{5}{c}{\underbrace{0\ \ 0\ \ 0\ \ 1\ \ 0}_{\text{5-bit result}}} \end{array} \qquad \begin{array}{r} 28 \\ +\ \ 6 \\ \hline 2 \end{array}$$

Extra bit is discarded.

- Carry flag = 1, indicating carry has occurred on unsigned addition.
- Carry flag is 1 because the result crosses the boundary between 31 and 0.

# Carry/borrow flag bit for unsigned numbers

***If the traverse crosses the boundary between 0 and $2^n - 1$, the carry flag is set on addition and is cleared on subtraction.***



A borrow occurs when subtracting 5 from 3.

$$\boxed{\text{Borrow} \quad 1\ 1\ 1\ 0\ 0}$$

$$\begin{array}{r} 0\ 0\ 0\ 1\ 1 \\ -\ \ 0\ 0\ 1\ 0\ 1 \\ \hline \underbrace{1\ 1\ 1\ 1\ 0}_{\text{5-bit result}} \end{array} \qquad \begin{array}{r} 3 \\ -\ \ 5 \\ \hline 30 \end{array}$$

- Carry flag = 0, indicating borrow has occurred on unsigned subtraction.
- For subtraction, carry = NOT borrow.

## Signed Integer Representation

- Three ways to represent signed binary integers:
  - Signed magnitude
    - $\boldsymbol{value} = (-1)^{\boldsymbol{sign}} \times \boldsymbol{Magnitude}$
  - One's complement ($\tilde{\alpha}$)
    - $\boldsymbol{\alpha + \tilde{\alpha} = 2^n - 1}$
  - Two's complement ($\bar{\alpha}$)
    - $\boldsymbol{\alpha + \bar{\alpha} = 2^n}$

| | Sign-and-Magnitude | One's Complement | Two's Complement |
|---|---|---|---|
| **Range** | $[-2^{n-1}+1, 2^{n-1}-1]$ | $[-2^{n-1}+1, 2^{n-1}-1]$ | $[-2^{n-1}, 2^{n-1}-1]$ |
| **Zero** | Two zeroes ($\pm 0$) | Two zeroes ($\pm 0$) | One zero |
| **Unique Numbers** | $2^n - 1$ | $2^n - 1$ | $2^n$ |

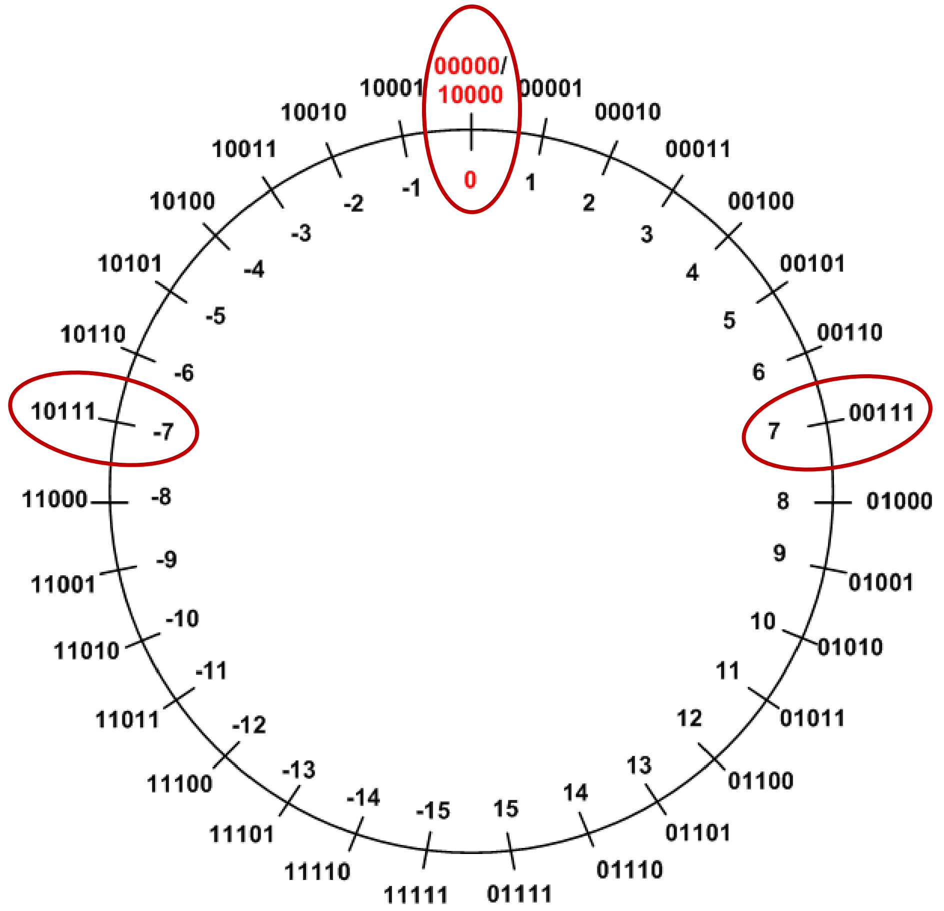
## Signed Integers
## Method 1: Signed Magnitude

***Sign-and-Magnitude:***

$$value = (-1)^{sign} \times Magnitude$$

- The most significant bit is the sign.
- The rest bits are magnitude.

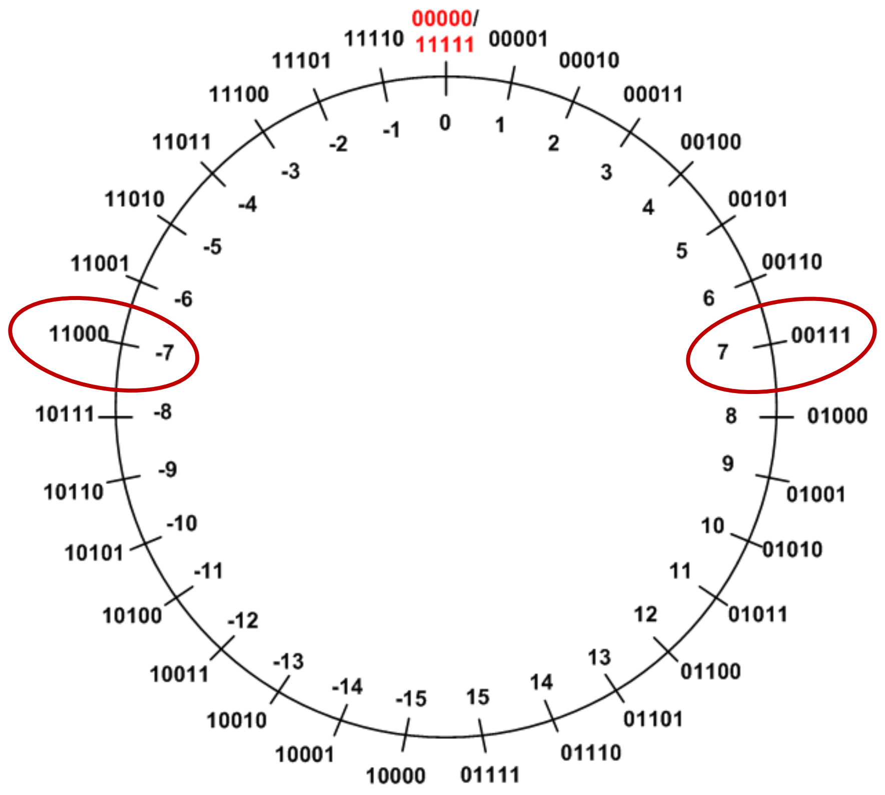
- Example: in a 5-bit system
  - $+7_{10} = 00111_2$
  - $-7_{10} = 10111_2$
- Two ways to represent zero
  - $+0_{10} = 00000_2$
  - $-0_{10} = 10000_2$
- Not used in modern systems
  - Hardware complexity
  - Two zeros



# Signed Integers
## Method 2: One's Complement

***One's Complement ($\widetilde{\alpha}$):***

$$\alpha + \widetilde{\alpha} = 2^n - 1$$



**The one's complement representation of a negative binary number is the bitwise NOT of its positive counterpart.**

Example: in a 5-bit system

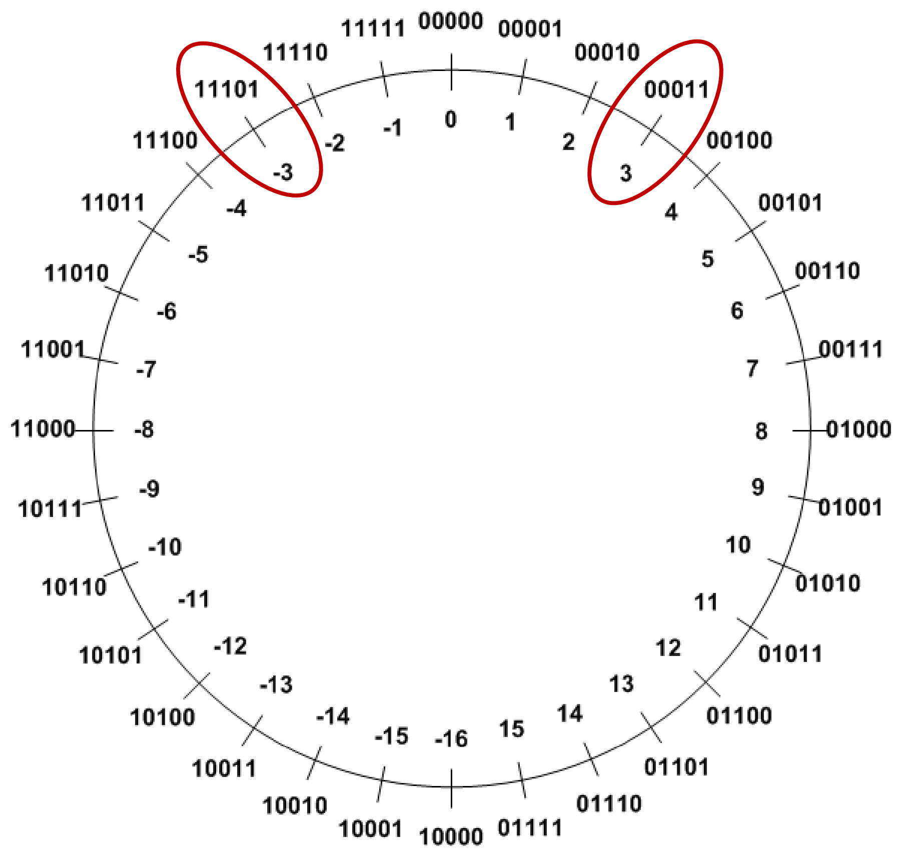
$$+7_{10} = 00111_2$$
$$-7_{10} = 11000_2$$

$$\begin{aligned} +7_{10} + (-7_{10}) &= 00111_2 + 11000_2 \\ &= 11111_2 \\ &= 2^5 - 1 \end{aligned}$$

## Signed Integers
## Method 3: Two's Complement (TC)

***Two's Complement ($\overline{\alpha}$):***

$$\alpha + \overline{\alpha} = 2^n$$



**TC of a negative number can be obtained by the bitwise NOT of its positive counterpart plus one.**

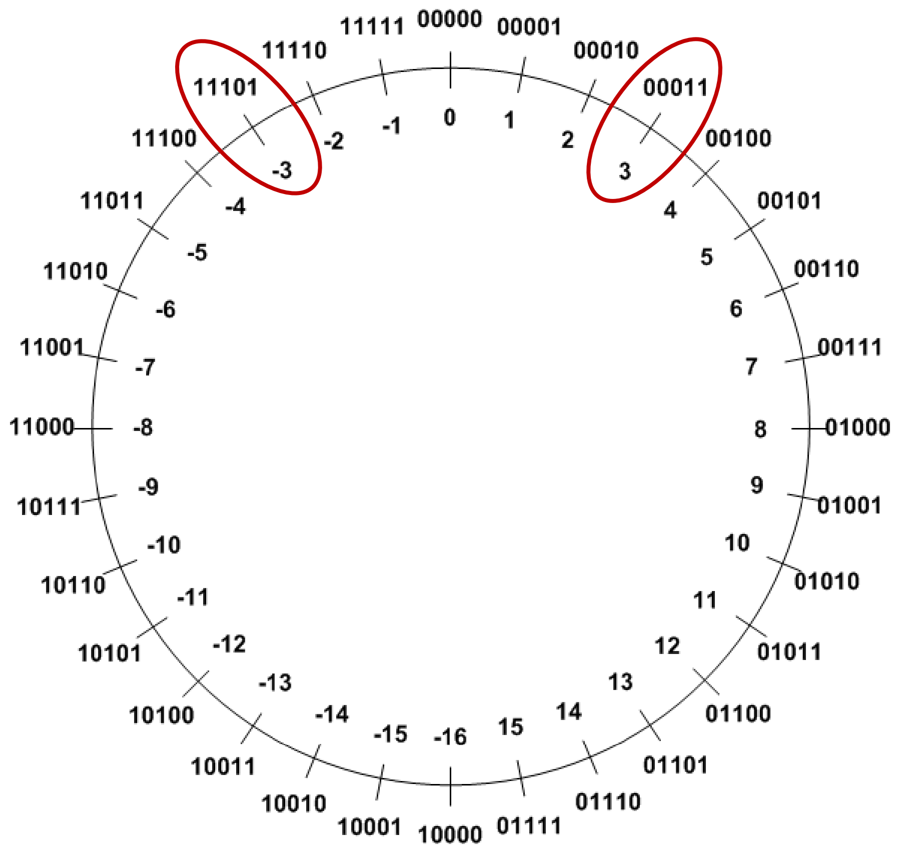
Example **1**: TC(3)

| | Binary | Decimal |
|---|---|---|
| Original number | 0b00011 | 3 |
| Step 1: Invert every bit | 0b11100 | |
| Step 2: Add 1 | + 0b00001 | |
| Two's complement | 0b11101 | -3 |

## Signed Integers
## Method 3: Two's Complement (TC)

***Two's Complement (TC)***

$$\alpha + \bar{\alpha} = 2^n$$



**TC of a negative number can be obtained by the bitwise NOT of its positive counterpart plus one.**

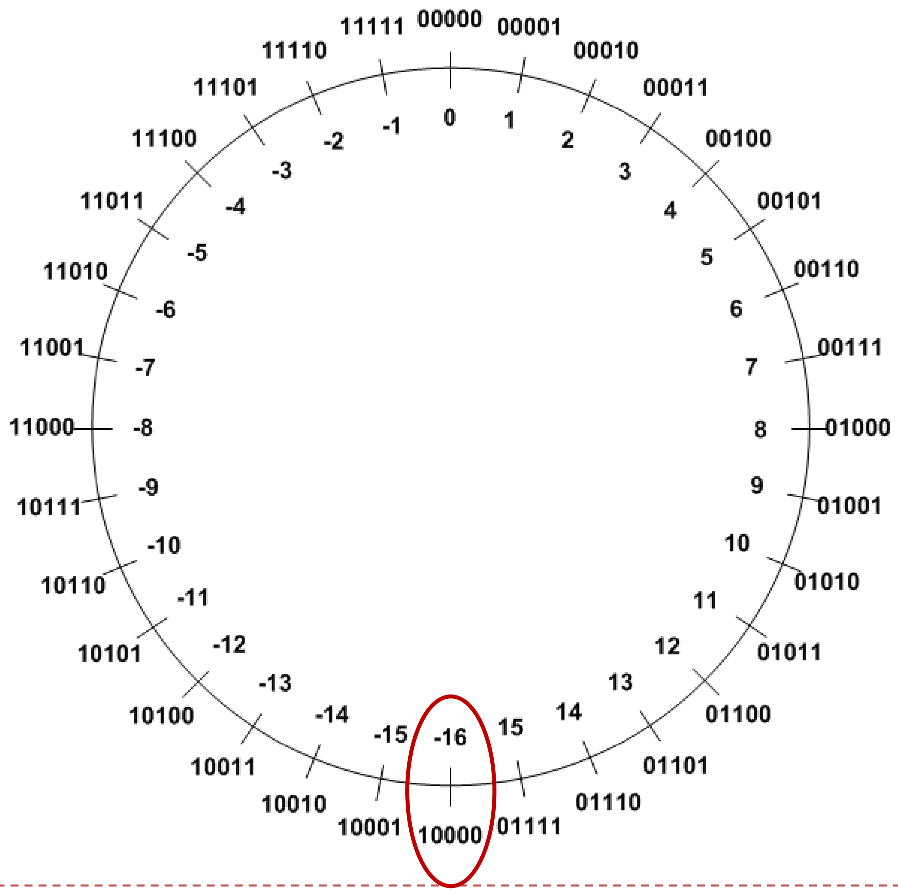
Example 2: TC(-3)

| | Binary | Decimal |
|---|---|---|
| Original number | 0b11101 | -3 |
| Step 1: Invert every bit | 0b00010 | |
| Step 2: Add 1 | + 0b00001 | |
| Two's complement | 0b00011 | 3 |

## Signed Integers
## Method 3: Two's Complement (TC)

***Two's Complement (TC)***

$$\alpha + \bar{\alpha} = 2^n$$



**TC of a negative number can be obtained by the bitwise NOT of its positive counterpart plus one.**

Example 3: TC(-16)

| | Binary | Decimal |
|---|---|---|
| Original number | 10000 | -16 |
| Step 1: Invert every bit | 01111 | |
| Step 2: Add 1 | + 10000 | |
| Two's complement | 10000 | -16 |

Negation of -16 in 5-bit two's complement wraps back to itself, meaning the most negative number's two's complement is itself. (Number range is [-16, 15], so 16 is out of range)

## Quiz

- Calculate TC(-6) for a 6-bit system

## Quiz ANS

- Calculate TC(-6) for a 6-bit system
- For a 6-bit two's complement number: the range of representable integers is from −32 to +31.
- −16 in 6-bit two's complement is 110000
  - Write 16 in binary: 010000
  - Take the two's complement (invert bits and add 1):
  - Invert bits: 101111, Add 1: 101111+1=110000
- To take the negation of this (i.e., find the two's complement of 110000):
  - Invert bits: 001111, Add 1: 001111+1=010000
  - This is 16 in binary, so the negation of −16 is +16 as expected
- Unlike the 5-bit case where −16 is the minimum and its negation wraps onto itself, in 6 bits −16 behaves normally with correct negation

## Two's Complement (TC)

- Two's complement gets its name from the rule that "*the unsigned sum of an n-bit number and its n-bit negative is* $2^n$"; hence, the negation or complement of a number x is $2^n - x$, or its "Two's complement"
- Signed arithmetic:
  - For 5-bit system, signed number x = 00011 = 3, TC(x) = 11101 = -3, so their signed sum is 00011 + 11101 = 00000 (in decimal 3 + (-3) = 0)
- Unsigned arithmetic:
  - Unsigned number 00011=3, 11101=29, so their unsigned sum is 00011 + 11101 = 00000 (in decimal 3 + 29 = 32 = $2^5$). The result is incorrect as Carry flag = 1: 32 cannot be represented in 5 bits since it exceeds the largest unsigned value of $2^5 - 1$

# Two's Complement for 8-bit System

| 8-bit signed Int (Two's Complement) | 8-bit unsigned Int | Binary |
|---|---|---|
| -128 | 128 | 1 0 0 0  0 0 0 0 |
| -127 | 129 | 1 0 0 0  0 0 0 1 |
| … | … | … |
| -2 | 254 | 1 1 1 1  1 1 1 0 |
| -1 | 255 | 1 1 1 1  1 1 1 1 |
| 0 | 0 | 0 0 0 0  0 0 0 0 |
| 1 | 1 | 0 0 0 0  0 0 0 1 |
| … | … | . . . |
| 127 | 127 | 0 1 1 1  1 1 1 1 |

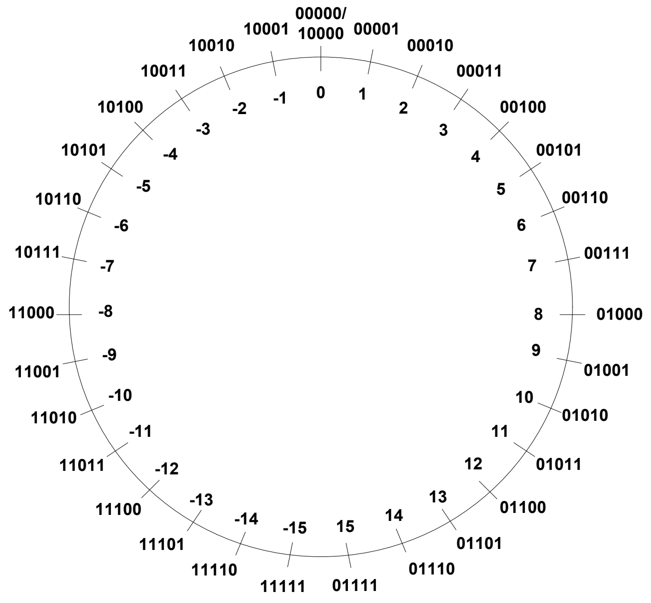
Note: Most significant bit (MSB) is the sign bit for signed int

## Sign Extension

| Decimal | Binary | | |
|---|---|---|---|
| | 4-bit | 8-bit | 32-bit |
| $3_{ten}$ | $0011_{two}$ | $0000\ 0011_{two}$ | $0000\ 0000\ 0000\ 0011_{two}$ |
| $-3_{ten}$ | $1101_{two}$ | $1111\ 1101_{two}$ | $1111\ 1111\ 1111\ 1101_{two}$ |

- Sign extension for unsigned int: fill in 0's from the left
- Sign extension for signed int: fill in the sign bit from the left

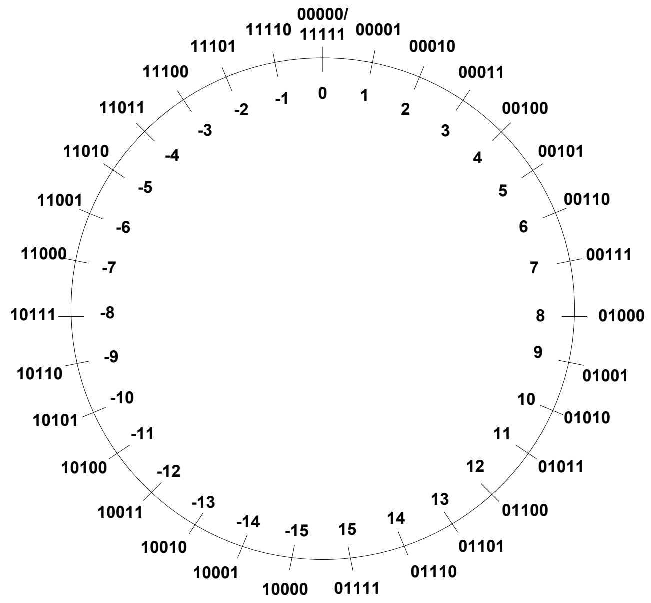
# Comparison



Signed magnitude representation

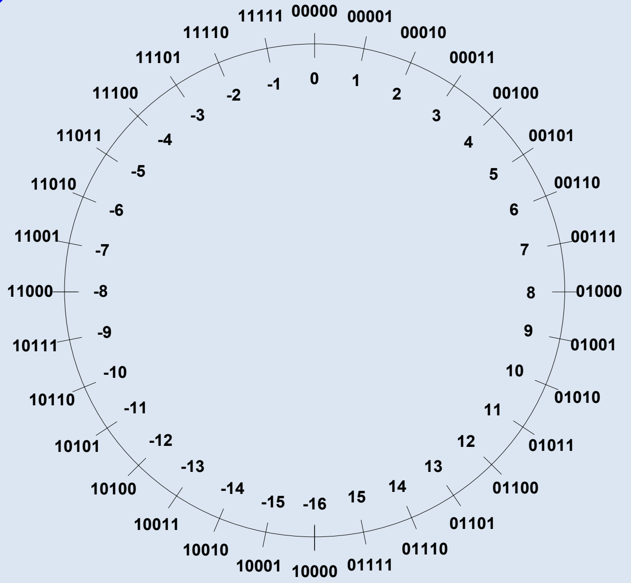
0 = positive

1 = negative



One's complement representation

Negative = invert all bits of a positive

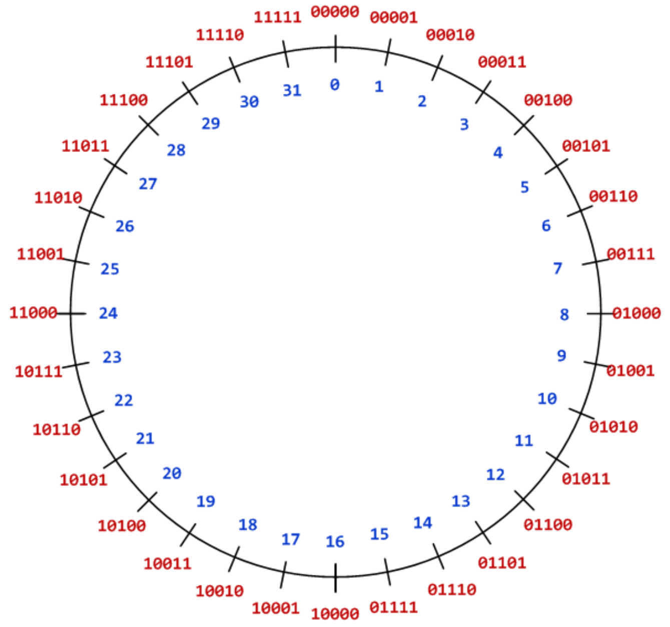


Two's Complement representation

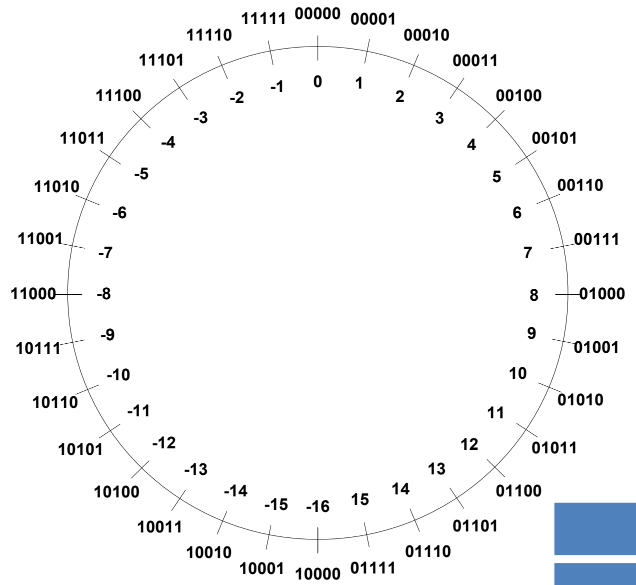
TC = invert all bits, then plus 1

Used in modern computers!

# Comparison: unsigned vs. signed



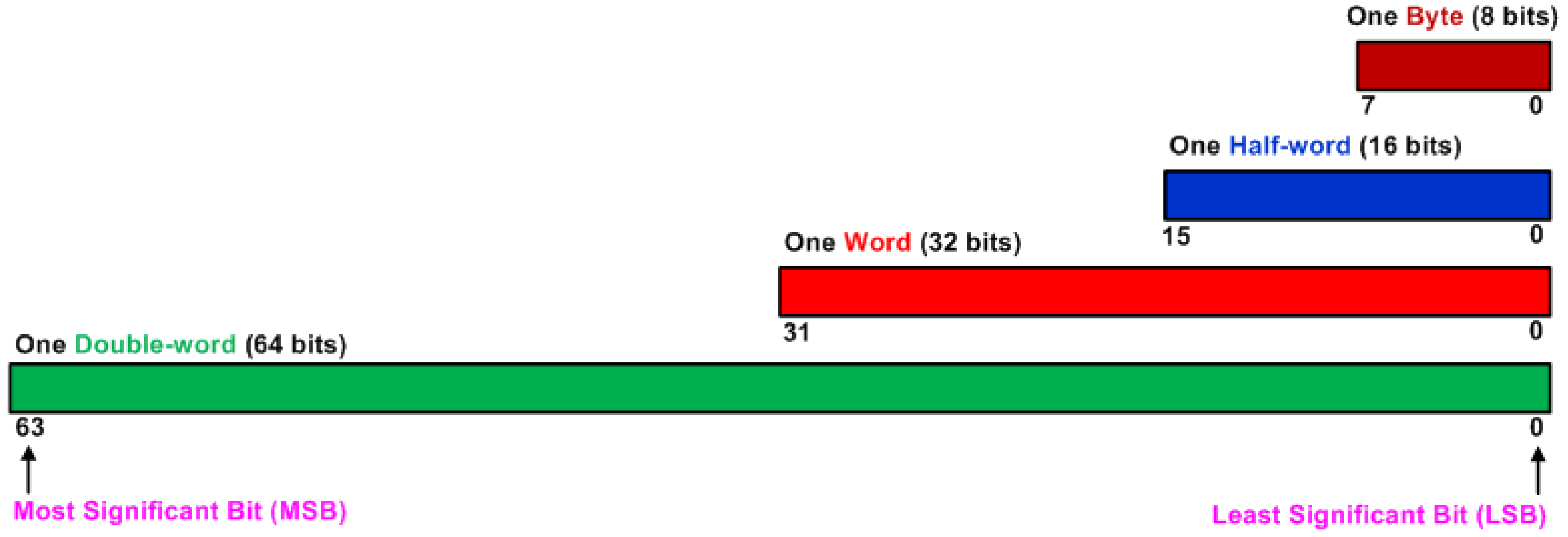
Unsigned int representation
Range [0, 31]



Two's Complement representation
Range [-16, 15]
TC = invert all bits, then add 1

| | Unsigned | Two's Complement Signed |
|---|---|---|
| **Range** | $[0, 2^{n-1}]$ | $[-2^{n-1}, 2^{n-1} - 1]$ |
| **Zero** | One zero | One zero |
| **Unique Numbers** | $2^n$ | $2^n$ |

# Range of Signed Integers (Two's Complement)



One Byte (8 bits)
7
0
One Half-word (16 bits)
15
0
One Word (32 bits)
31
0
One Double-word (64 bits)
63
0
Most Significant Bit (MSB)
Least Significant Bit (LSB)

| Storage Size | Range | Powers of 2 |
|---|---|---|
| Signed Byte | -128 to +127 | $-2^7$ to $2^7$-1 |
| Signed Halfword | -32,768 to +32,767 | $-2^{15}$ to $2^{15}$-1 |
| Signed Word | -2,147,483,648 to +2,147,483,647 | $-2^{31}$ to $2^{31}$-1 |
| Signed Double-word | -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 | $-2^{63}$ to $2^{63}$-1 |

## Overflow Flag for Signed Arithmetic

- When adding signed numbers represented in two's complement, overflow occurs only in two scenarios:
  1. adding two positive numbers but getting a non-positive result, or
  2. adding two negative numbers but yielding a non-negative result.

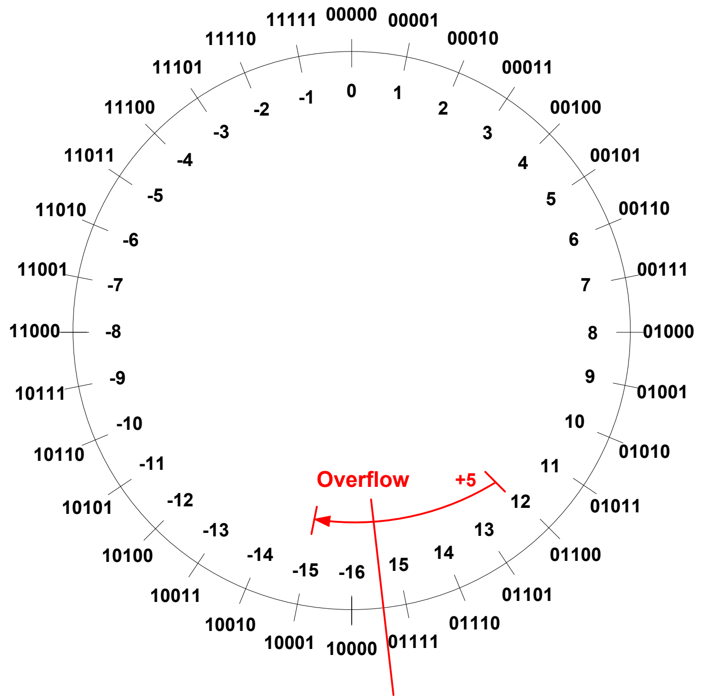
- Similarly, when subtracting signed numbers, overflow occurs in two scenarios:
  1. subtracting a positive number from a negative number but getting a positive result, or
  2. subtracting a negative number from a positive number but producing a negative result.

- Overflow cannot occur when adding operands with different signs or when subtracting operands with the same signs.
  - Why?

## Overflow Flag for Signed Arithmetic

- Overflow cannot occur when adding 2 operands with different signs or when subtracting 2 operands with the same sign. Proof:
- A n-bit signed int has the range $[-2^{n-1}, 2^{n-1}-1]$
  - n = 4, number range [-16, 15]
- 2 operands with different signs: positive one in the range of $[0, 2^{n-1}-1]$, negative one in the range of $[-2^{n-1}, -1]$. So the range of their sum must be $[0-2^{n-1}, 2^{n-1}-1+(-1)]=[-2^{n-1}, 2^{n-1}-2] \in [-2^{n-1}, 2^{n-1}-1]$
  - Positive number range [0, 15], negative number range [-16, -1]. Range of their sum [0-16, 15-1]=[-16, 14]
- 2 operands with the same sign: if both are positive and in the range of $[0, 2^{n-1}-1]$, then the range of their difference must be $[0-(2^{n-1}-1), 2^{n-1}-1-0]=[-(2^{n-1}-1), 2^{n-1}-1]$; if both are negative and in the range of $[-2^{n-1}, -1]$, then the range of their difference must be $[-2^{n-1}-(-1), -1-(-2^{n-1})]=[-2^{n-1}+1, 2^{n-1}-1] \in [-2^{n-1}, 2^{n-1}-1]$
  - Both positive numbers [0, 15], range of difference [0-15, 15-0]=[-15, 15]
  - Both negative numbers [-16, -1], range of difference [-16-(-1), -1-(-16)]=[-15, 15]

# Overflow for Signed Add

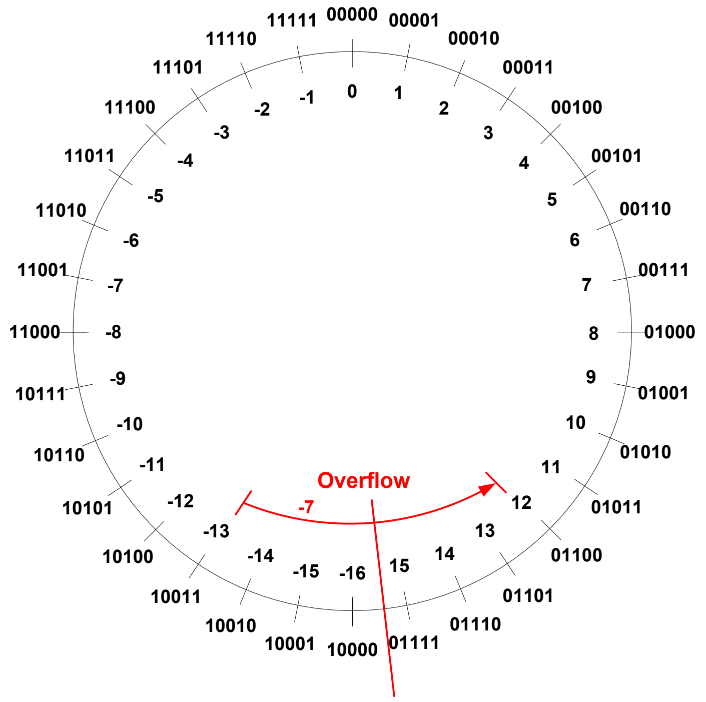


Overflow occurs when adding two positive integers but getting a negative result.

$$\begin{array}{r} 0\ 1\ 1\ 0\ 0 \\ +\ \ 0\ 0\ 1\ 0\ 1 \\ \hline \underbrace{1\ 0\ 0\ 0\ 1}_{\text{5-bit result}} \end{array} \qquad \begin{array}{r} 12 \\ +\ \ 5 \\ \hline -15 \end{array}$$

1. On addition, overflow occurs if $sum \geq 2^4$ when adding two positives.
2. Overflow never occurs when adding two numbers with different signs.

# Overflow for Signed Add



Overflow occurs when adding two negative integers but getting a positive result.

$$
\begin{array}{cccccccc}
 & & 1 & 0 & 0 & 1 & 1 \\
 & + & 1 & 1 & 0 & 0 & 1 \\
\hline
\boxed{1} & & 0 & 1 & 1 & 0 & 0
\end{array}
\qquad
\begin{array}{r}
-13 \\
+\ -7 \\
\hline
12
\end{array}
$$

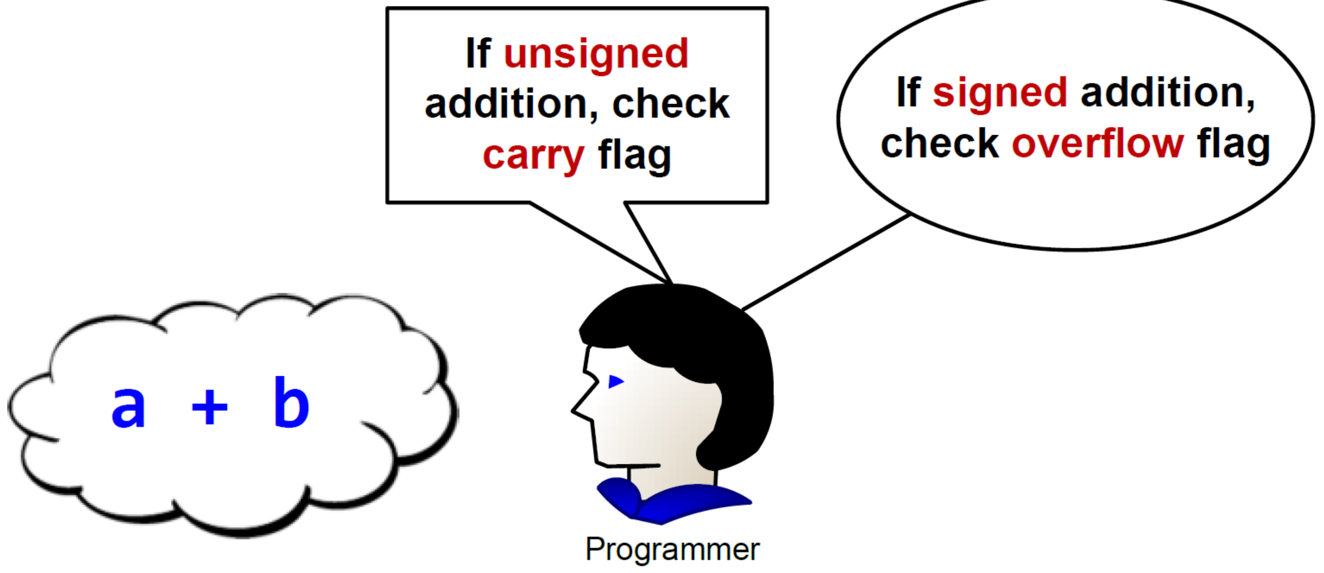
Extra bit is discarded.

5-bit result

On addition, overflow occurs if $sum < -2^4$ when adding two negatives.

## Signed or unsigned

- Whether the carry flag or the overflow flag should be used depends on the programmer's intention.


If unsigned addition, check carry flag
If signed addition, check overflow flag
a + b
Programmer

- When programming in high-level languages such as C, the compiler automatically chooses to use the carry or overflow flag based on how this integer is declared in source code ("`int`" or "`unsigned int`").

## Signed or Unsigned

```
a = 0b10000
b = 0b10000
c = a + b
```

- Whether the carry flag or the overflow flag should be used depends on the programmer's intention: Are *a* and *b* signed or unsigned numbers?

```
uint a;
uint b;
…
c = a + b
…
```

C Program

**Check the carry flag!**

```
int a;
int b;
…
c = a + b
…
```

C Program

**Check the overflow flag!**

## Signed or Unsigned

```
a = 0b10000
b = 0b10000
c = a + b
```

- Are *a* and *b* signed or unsigned numbers?

- CPU does not know and does not care; it sets up both carry flag and overflow flags.
- It is software's (programmer/compiler) responsibility to interpret the flags.
  - The C compiler uses either the carry or the overflow flag based on how this integer is declared in source code ("uint" or "int").

```
If unsigned:
uint a, b;
a = 16
b = 16
c = a + b
  = 32 > 2^5-1
Carry flag set
```

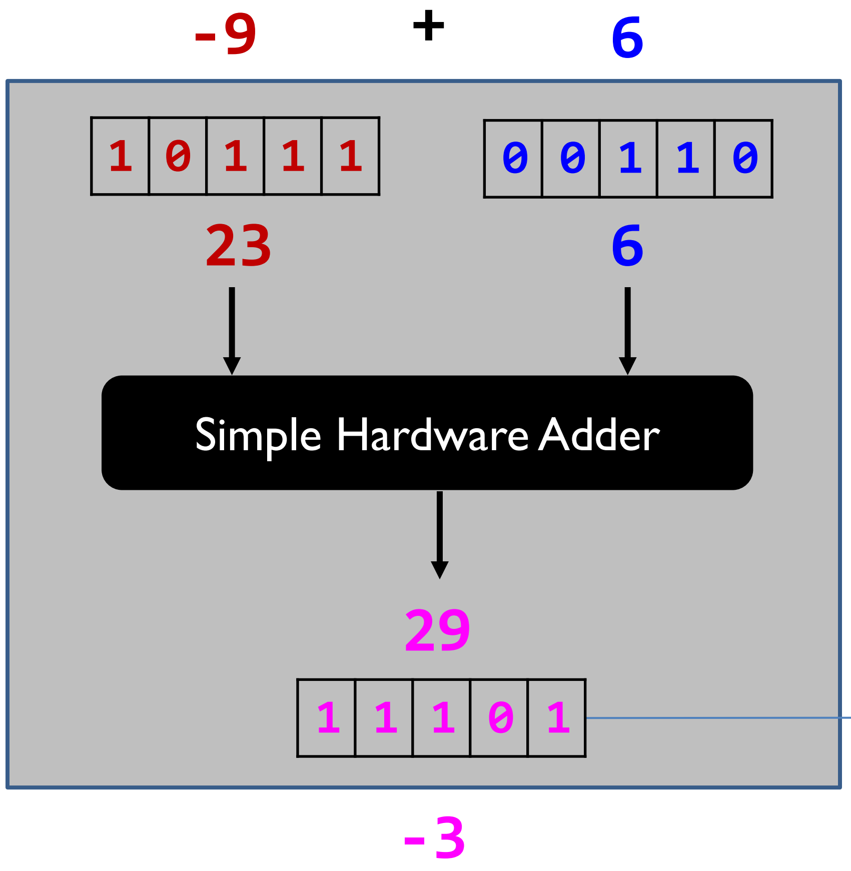
```
If signed:
int a, b;
a = -16
b = -16
c = a + b
  = -32 < -2^4
Overflow flag set
```

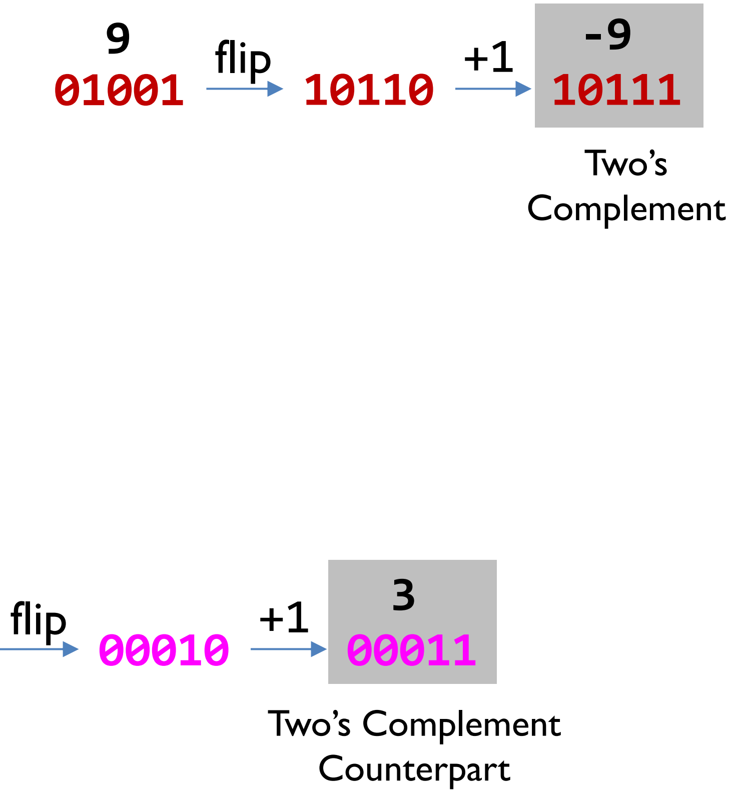
## Two’s Complement Simplifies Hardware Implementation

- In two’s complement, the same hardware works correctly for both signed and unsigned addition/subtraction.
- If the product is required to keep the same number of bits as operands, unsigned multiplication hardware works correctly for signed numbers.
- However, this is not true for division. (not discussed in this course)

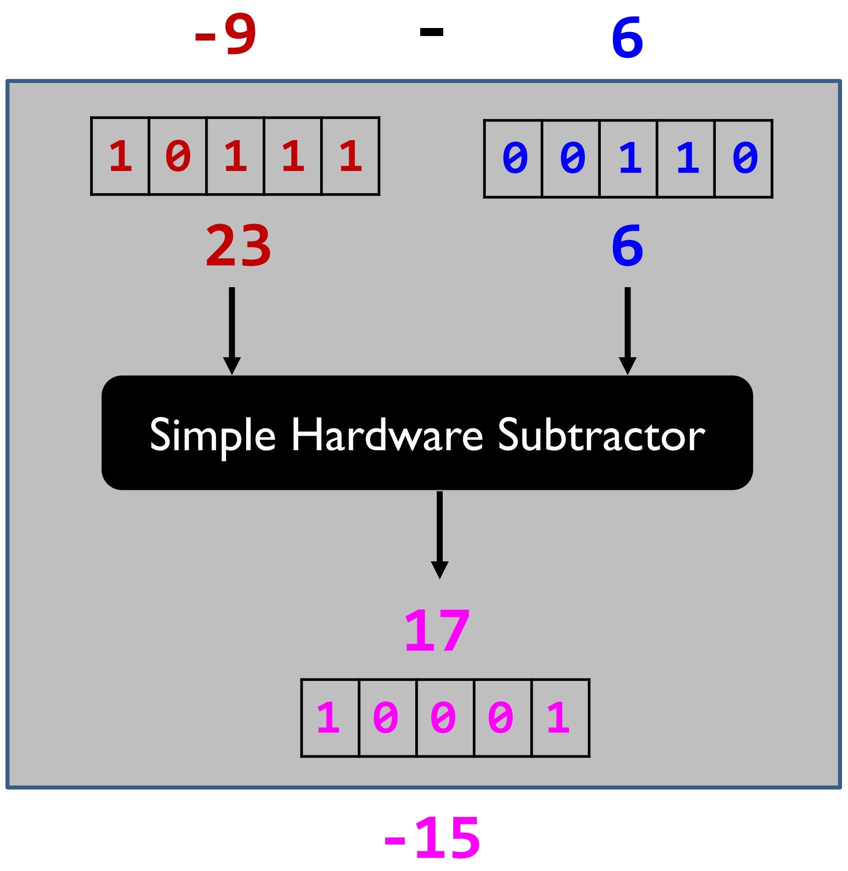
| Operation | Are signed and unsigned operations the same? |
|---|---|
| Addition | Yes |
| Subtraction | Yes |
| Multiplication | Yes if the product is required to keep the same number of bits as operands |
| Division | No |

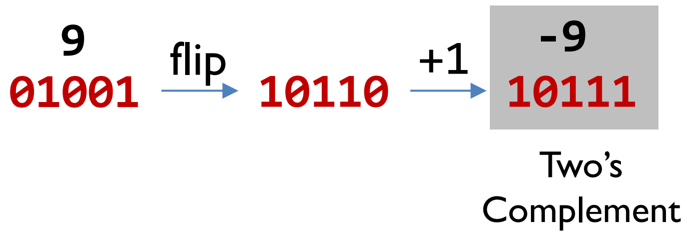
## Adding two signed integers: (-9) + 6

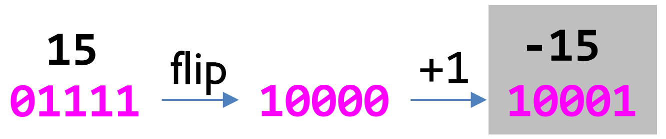




## Subtracting two signed integers: `(-9) - 6`

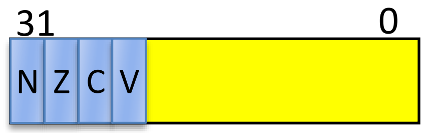






## Condition Codes

| Bit | Name | Meaning after add or sub |
|---|---|---|
| N | negative | result is negative |
| Z | zero | result is zero |
| C | carry | signed arithmetic out of range |
| V | overflow | signed arithmetic out of range |



CPSR (Current Program Status Register)

- C is set upon an **unsigned** addition if the answer is wrong
- C is cleared upon an **unsigned** subtract if the answer is wrong
- V is set upon a **signed** addition or subtraction if the answer is wrong

Why do we care about these bits?

IMPORTANT

## Carry and Overflow Flags

Carry flag C = 1 (Borrow flag = 0) upon an **unsigned** addition if the answer is wrong (true result > $2^n$-1)
Carry flag C = 0 (Borrow flag = 1) upon an unsigned subtraction if the answer is wrong (true result < 0)
Overflow flag V =1 upon a **signed** addition or subtraction if the answer is wrong (true result > $2^{n-1}$-1 or true result < -$2^{n-1}$)

$$c = a \pm b$$

| | **Carry (for unsigned)** | **Overflow (for signed)** |
|---|---|---|
| Add | $Carry = 1$ if $c$ is too large to fit in. | $Overflow = 1$ if $c$ is too large or too small to fit in |
| Subtract | $Borrow = 1$, i.e. $Carry = 0$ if $a < b$. | |

- ARM Cortex-M has no dedicated borrow flag, carry flag is reused.
- For unsigned subtract, $Borrow = \overline{Carry}$

- Signed Subtraction is converted to sign addition
- $a - b = a + (-b)$

## Characters

**A**merican
**S**tandard
**C**ode for
**I**nformation
**I**nterchange

- Standard ASCII
  0 - 127
- Extended ASCII
  0 - 255
- ANSI
  0 - 255
- Unicode
  0 - 65535

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | NUL | 32 | 20 | SP | 64 | 40 | @ | 96 | 60 | ‘ |
| 1 | 01 | SOH | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 02 | STX | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 03 | ETX | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 04 | EOT | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 05 | ENQ | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 06 | ACK | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 07 | BEL | 39 | 27 | ’ | 71 | 47 | G | 103 | 67 | g |
| 8 | 08 | BS | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 09 | HT | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | LF | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | VT | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | FF | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | CR | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | SO | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | SI | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | DLE | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | DC1 | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | DC2 | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | DC3 | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | DC4 | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | NAK | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | SYN | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | ETB | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | CAN | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | EM | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | SUB | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | ESC | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | FS | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | GS | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | RS | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | US | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | DEL |

Standard ASCII: Encoding 128 characters

# Null-terminated String

```
char str[13] = “ARM Assembly”;
// The length has to be at least
// 13 even though it has 12
// letters. The NULL terminator
// should be included.
```

or simply

```
char str[] = “ARM Assembly”;
```

| Memory Address | Memory Content | Letter |
|---|---|---|
| str + 12 → | 0x00 | \0 |
| str + 11 → | 0x79 | y |
| str + 10 → | 0x6C | l |
| str + 9 → | 0x62 | b |
| str + 8 → | 0x6D | m |
| str + 7 → | 0x65 | e |
| str + 6 → | 0x73 | s |
| str + 5 → | 0x73 | s |
| str + 4 → | 0x41 | A |
| str + 3 → | 0x20 | space |
| str + 2 → | 0x4D | M |
| str + 1 → | 0x52 | R |
| str → | 0x41 | A |

## String Comparison

Strings are compared based on their ASCII values

- "j" < "jar" < "jargon" < "jargonize"
- "CAT" < "Cat" < "DOG" < "Dog" < "cat" < "dog"
- "12" < "123" < "2"< "AB" < "Ab" < "ab" < "abc"

## String Length

- Stings are terminated with a null character (NUL, ASCII value 0x00)

**Pointer dereference operator ***

```
int strlen (char *pStr){
    int i = 0;

    // loop until *pStr is NULL
    while( *pStr ) {
        i++;
        pStr++;
    }
    return i;
}
```

**Array subscript operator [ ]**

```
int strlen (char *pStr){
    int i = 0;

    // loop until pStr[i] is NULL
    while( pStr[i] )
        i++;

    return i;
}
```

# Convert to Upper Case

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 4A | 4B | 4C | 4D | 4E | 4F | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 5A |

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 6A | 6B | 6C | 6D | 6E | 6F | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 7A |

‘a’ – ‘A’ = 0x61 – 0x41 = 0x20 = 32

**Pointer dereference operator ***

```
void toUpper(char *pStr){
  char *p;

  for(*p = pStr; *p; ++p){
    if(*p >= ’a’ && *p <= ’z’)
      *p -= ‘a’ – ‘A’;
      //or: *p -= 32;
  }
}
```

**Array subscript operator [ ]**

```
void toUpper(char *pStr){
  int i;
  char c = pStr[0];
  for(i = 0; c; i++, c = pStr[i];) {
    if(c >= ’a’ && c <= ’z’)
      pStr[i] -= ‘a’ – ‘A’;
      // or: pStr[i] -= 32;
  }
}
```

## Summary

- Unsigned integer arithmetic
- Signed integer arithmetic
  - 2's complement
- ASCII strings

## References

- Lecture 1. Why use two's complement?
  - https://www.youtube.com/watch?v=IJCefqV80ck&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=1
- Lecture 2: Carry flag for unsigned addition and subtraction
  - https://www.youtube.com/watch?v=MxGW2WurKuM&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=2
- Lecture 3: Overflow flag for signed addition and subtraction
  - https://www.youtube.com/watch?v=BIn6iyYIGio&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=3