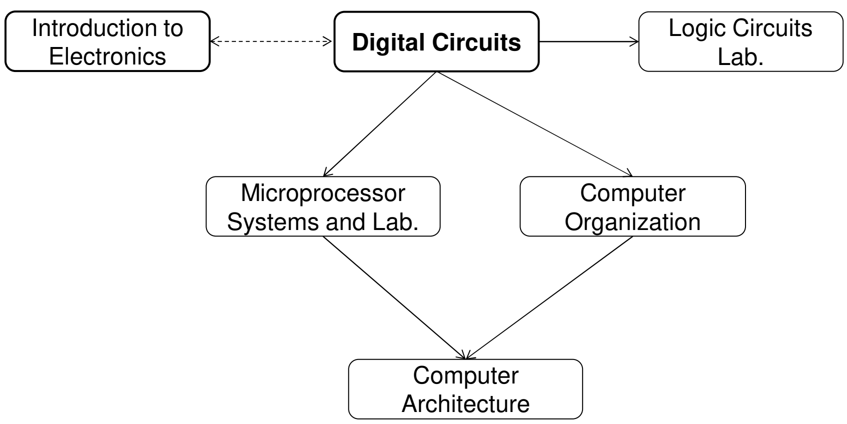
DIGITAL CIRCUITS

Assoc.Prof. Feza BUZLUCA
Istanbul Technical University
Computer Engineering Department
https://akademi.itu.edu.tr/en/buzluca/
https://www.buzluca.info

Connection between hardware-based courses in the İTÜ Computer Engineering Department

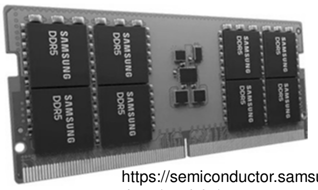


Where are digital circuits used?

Today, almost every electronic device is implemented as a digital circuit.

Examples:

- Central processing unit (CPU): It is a "synchronous sequential" circuit.
  We will see sequential digital circuits in the second part of this course.
- Computer memory: We will see flip-flops and latches, which are digital circuits and building blocks of memory.
- Home electronics: TV, washing machine control unit, video and audio devices.
- Cars: ABS, engine control
- Cell phones



https://semiconductor.samsung.com/dram/module/

https://upload.wikimedia.org/wikipedia/commons/a/a3/AMD_Ryzen_9_7900X.jpg

https://newsroom.intel.de/news-releases/

Analog – Digital Signals:

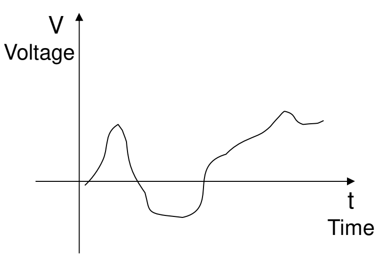
In the real world (where we live), many physical quantities (current, voltage, temperature, light intensity, weight of a person, etc.) vary over a continuous range.

These types of signals, which can take any possible value between the limits of such a range, are called **analog signals**.

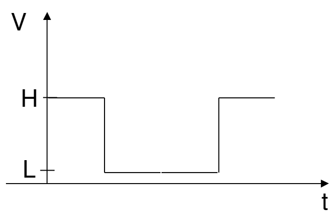
An analog signal has a theoretically infinite resolution.

A **binary digital signal** may at any moment take on only one of two possible values:

0 - 1, high - low, on - off, true - false, open - closed.



Analog Signal (continuous)



One-bit Binary Digital Signal (discrete)

To represent an analog signal, more than one bit is necessary (see coding).

Advantages of Digital Systems:

Because of their advantages over the old analog systems, digital systems are used in many areas today.

Examples: Photography, video, audio, automobile engines, telephone systems, and so on.

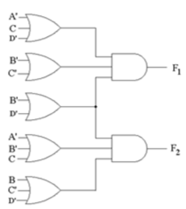
Advantages of Digital Systems:

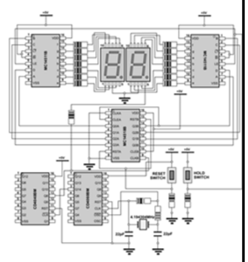
- The mathematics of digital design (Boolean algebra) is more straightforward than the mathematics of analog systems.
- Digital systems are easier to test and debug.
- Digital systems are flexible and programmable. Today, digital systems are implemented in the form of programmable logic devices and computers (embedded systems).

  This way, devices can be reprogrammed according to new requirements without changing the hardware.
- Digital data can be stored and processed in computer systems.
- Digital systems work faster.
- Digital systems continue to evolve (but improvements are slowing down).

See the BLG 322E- Computer Architecture lecture notes.

Steps of Digital System Implementation:





Voltage
Current
Temperature
Speed
Image
Switch

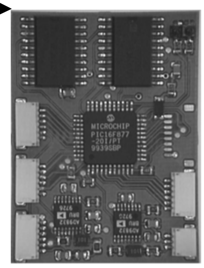
Real World (Physical world, Problem Domain)

0, 1
and, or not
$z = \bar{a} \cdot b + c$

Mathematical World Model
(Boolean Algebra)

HDL Program
(*Hardware Description Language*)

```
WHEN(A & B) THEN X1-C;
ELSE X2 – E # F;
X2 – (!A # C);
......
```

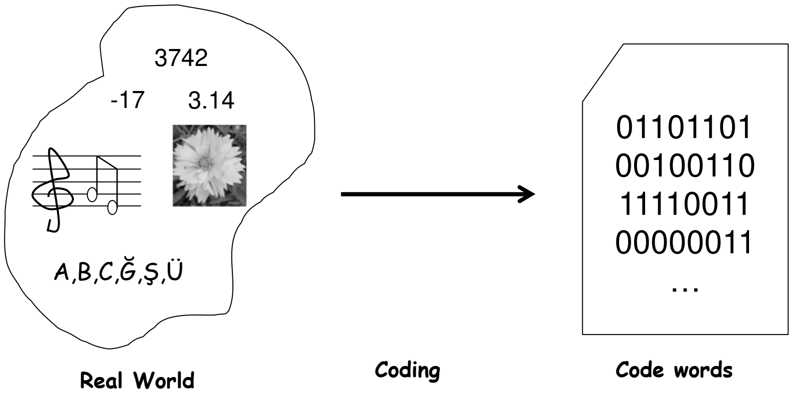


Abstraction, Modeling → **Design** → **Implementation**
(Coding)

Binary Digital Coding:

As digital systems operate on binary digital signals, they can process only two different values (binary data): ON-OFF, LOW-HIGH, **0-1**.

Therefore, physical quantities (voltage, temperature, etc.) and any kind of data (letters, numbers, colors, sounds) must be converted to binary numbers (digitally coded) before digital circuits can process them.



Digital Coding (cont'd):

Using n bits (***b**inary dig**its***), 2^n different "things" can be represented.

n bits $\rightarrow$ 2^n different "things"

For example, an 8-bit (***binary digit***) binary number can represent 2^8 (256) different "things".

These can be 256 different colors, 256 symbols, integers between 0 and 255, integers from 1 to 256, or integers between -128 and +127.

00000000, 00000001, 00000010, … , 11111101, 11111110, 11111111.

There are different **coding systems** (methods) for different types of data.

The meaning of a binary value (for example, 10001101) is determined by the system (hardware or software) that processes this number.

This value may represent a number, a color, or another type of data.

The coding of numbers is especially important.

Therefore, in this course, we will give some basic information about the coding methods of numbers.

BCD (Binary Coded Decimal) Coding System:

Each decimal number between 0 and 9 is represented by a four-bit pattern.

Natural BCD:

Number:	BCD Code:	Number:	BCD Code:
0:	0000	5:	0101
1:	0001	6:	0110
2:	0010	7:	0111
3:	0011	8:	1000
4:	0100	9:	1001

Example:

Number:	8	0	5
BCD	:1000	0000	0101

It is a **redundant code**.
Using 4 bits, 16 different code values can be created, but only 10 of them are used.

It is difficult to perform arithmetic operations on BCD numbers.

Therefore, today's computer systems do not use BCD coding to represent numbers.

Positional (weighted) Coding:

Each digit of the number has an associated weight.

Natural Binary Coding: Numbers are represented using positional (weighted) coding in binary (base-2).

Example: (unsigned) 11010 = $1{\cdot}2^4 + 1{\cdot}2^3 + 0{\cdot}2^2 + 1{\cdot}2^1 + 0{\cdot}2^0 = 26$

The leftmost bit is called the *Most Significant Bit – MSB (high-order)*.

The rightmost bit is called the *Least Significant Bit – LSB (low-order)*.

In today's computers, natural binary coding is used to represent numbers.

Hamming distance: The Hamming distance between two n-bit long code words is the number of bit positions at which the corresponding bits are different.

Example: Hamming distance between 011 and 101 is 2.

Richard Wesley Hamming (1915-1998) Mathematician, USA

Adjacent Codes: Between each pair of successive code words, the Hamming distance is 1 (only one bit changes).

In addition, if the Hamming distance between the first and last code words is 1, then this code is called **cyclic** (circular).

Example: A **cyclic BCD** code (different from natural BCD !)

Number:	Code:	Number:	Code:
0:	0000	5:	1110
1:	0001	6:	1010
2:	0011	7:	1000
3:	0010	8:	1100
4:	0110	9:	0100

Gray Code: A **binary** (base 2), **nonredundant,** and **cyclic** (also adjacent) coding system that represents 2^n elements is called a **Gray code**.

Example: A 2-bit Gray code:

Num.:	Code:
0:	00
1:	01
2:	11
3:	10

The patent of the Gray Code was issued to physicist *Frank Gray* in 1953 when he was working at Bell Labs.

Representation of Numbers in Digital Systems (and Computers)

In this course, we will deal with *integers*.

Representation of the *floating point* numbers will be covered in the Computer Architecture course (See: https://web.itu.edu.tr/buzluca/course.html).

Before coding the numbers, we must decide what type of numbers (**unsigned** or **signed)** we will work with.

This is because unsigned and signed numbers are encoded and represented differently.

Representation of Unsigned Numbers (Integers):

In computers, unsigned integers are represented using "natural binary weighted (positional) coding".

n-bit unsigned binary integer: $a_{n-1}\, a_{n-2} \ldots a_2\, a_1\, a_0$, $a_i \in B=\{0,1\}$

a_{n-1}: High-order bit "*Most Significant Bit – MSB*"

a_0 : Low-order bit "*Least Significant Bit – LSB*"

Converting binary to decimal:

$$\text{Decimal value} = a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \ldots + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

weights

Representation of Unsigned Integers (cont'd):

Example: 8-bit unsigned integer

$(1101\ 0111)_2 = 1{\cdot}2^7 + 1{\cdot}2^6 + 0{\cdot}2^5 + 1{\cdot}2^4 + 0{\cdot}2^3 + 1{\cdot}2^2 + 1{\cdot}2^1 + 1{\cdot}2^0 = 215_{10}$

Converting decimal to binary:

Example: 215_{10}

```
215/2 = 107 remainder  1  (low-order bit "Least Significant Bit – LSB")  last (rightmost) bit
107/2 = 53  remainder  1
 53/2 = 26  remainder  1
 26/2 = 13  remainder  0
 13/2 = 6   remainder  1
  6/2 = 3   remainder  0
  3/2 = 1   remainder  1
  1/2 = 0   remainder  1  (high-order bit "Most Significant Bit – MSB")  first (leftmost) bit
```

The **largest unsigned** integer that can be represented with n bits: $\mathbf{2^n - 1}$

Example:

For n=8, largest unsigned integer is $1111\ 1111_2 = 255_{10}$

The **smallest unsigned** integer that can be represented with 8 bits: **0**

$0000\ 0000_2 = 0_{10}$

Representation of Signed Numbers (Integers):

The (high-order) most significant bit denotes the sign of the number.

- Positive numbers start with a "0",
- Negative numbers start with a "1".

Positive integers:

In computers, **positive** integers are represented (like unsigned integers) using "natural binary weighted (positional) coding".

Remember: Positive binary numbers must start with 0.

Examples of Positive Numbers:

8-bit $+5_{10}$: **0**000 0101
8-bit $+100_{10}$: **0**110 0100
4-bit $+5_{10}$: **0**101
4-bit $+7_{10}$: **0**111

The range of **positive signed** integers that can be represented with n bits:

00…00 to **0**1…11 (n bits) (Decimal: **0** to + $\mathbf{2^{n-1} - 1}$)

Example: n=8

The range of **positive signed** integers that can be represented with 8 bits:

0000 0000 to **0**111 1111 (Decimal: 0 to +127)

Negative integers:

Negative integers are represented using **2's complement**.

In this representation, negative integers are represented by the 2's complement of the positive number (absolute value).

To obtain the 2's complement:

- First, invert (1's complement) the number. Change 0 to 1, 1 to 0.
- Then, add 1 to the inverted number.

2's complement of $(A) = \overline{A} + 1$ $\qquad \overline{A}$ denotes 1's complement of A.

The 2's complement representation makes **adding or subtracting two numbers easy** without sign and magnitude checks.

This coding system makes it possible to perform subtraction using circuitry designed only for addition (we will see adder and subtracter circuits).

Thus, it simplifies the design of digital circuits for arithmetic operations.

Examples of Negative Numbers:

4-bit $+7_{10}$	: **0**111	8-bit $+6_{10}$	: **0**000 0110
1's complement	: 1000	1's complement	: 1111 1001
Add 1	: + 1	Add 1	: + 1
Result -7_{10}	: **1**001	Result -6_{10}	: **1**111 1010

2's complement system (cont'd):

Taking the 2's complement of a number **changes the sign** of the number.

Taking the 2's complement of a negative number makes it positive (see 1.17 for special cases).

2's complement operation:

positive → negative
negative → positive

It will be updated on slide 1.18.

Example: Making a negative number positive :

8-bit -6_{10}	: **1**111 1010
1's complement	: 0000 0101
Add 1	: + 1
Result: $+6_{10}$	: **0**000 0110

Examine the programs `integer_interpret_8.cpp` and `integer_interpret_16.cpp`

The range of negative signed integers that can be represented with n bits:

10…00 to **1**1…11 (n bits) Decimal: ($-\mathbf{2^{n-1}}$ to $-\mathbf{1}$)

Example: n=8

The range of negative signed integers that can be represented with 8 bits:

1000 0000 to **1**111 1111 Decimal: (-128 to -1)

Special cases in 2's complement representation

The negative integer with the largest absolute value (-8 in the case of 4 bits) is a special case in 2's complement representation.

Remember: Using n bits, we can represent signed decimal numbers between

$$-2^{n-1} \text{ and } +(2^{n-1} - 1).$$

For example, using 4 bits, we can represent signed decimal numbers between

$$-8 \text{ and } +7.$$

The number -8 can be represented with 4 bits: -8 = 1000.

However, +8 **cannot** be represented with 4 bits.

If we take the 2's complement of 1000 (-8) to obtain +8, the result is 1000. However, 1000 cannot be a positive number because it starts with 1.

4-bit -8_{10}	: **1000**
1's complement	: 0111
Add 1	: + 1
Result: ?	: 1000

If we take the 2's complement of 4-bit "-8", we find its 4-bit **magnitude** (**unsigned**), i.e., 1000 = (unsigned) 8 (special case for 4 bits).

Special cases in 2's complement representation (cont'd)

To obtain the 2's complement 4-bit binary number for the decimal number -8, we also start with the 4-bit **magnitude** (unsigned absolute value) of 8.

We start with the unsigned absolute value of 8. Its UNSIGNED 4-bit binary number equivalent is 1000.

8 (4-bit magnitude)	: 1000
1's complement	: 0111
Add 1	: + 1
Result: -8 (4-bit)	: **1000**

Based on this information we can update the explanation about the 2's complement operation (given in slide 1.16) as follows:

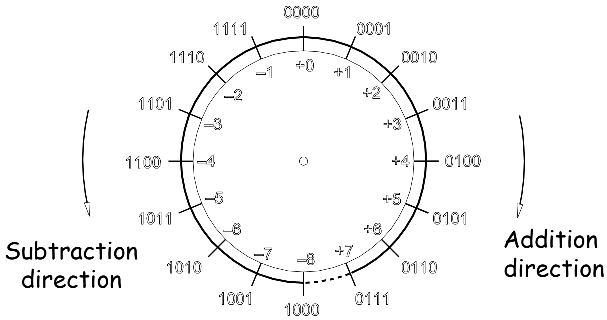
2's complement operation:

unsigned magnitude $\rightarrow$ negative

negative $\rightarrow$ unsigned magnitude

Representing signed numbers on a circular graphic:

We can represent signed numbers on a number wheel.
Example: 4-bit numbers:



Using 4 bits, we can generate 16 codewords and thus represent 16 integers.

When we use the 2's complement method, 8 of the codewords represent positive integers (including 0): `0000 - 0111`.

The remaining 8 codewords represent negative integers: `1111 - 1000`.

Extension (Sign Extension) of Binary Numbers

In digital systems, a certain number of bits (memory locations) are allocated for binary numbers, e.g., 16 bits, 32 bits, etc.

- In some cases, it is necessary to write a number to a memory space with more bits than necessary (for example, 8-bit number to 16-bit memory).
- Sometimes, it is necessary to operate on two numbers of different lengths.

In such cases, the shorter number is extended (word length is increased).

For example, extension from 4 bits to 8 bits or from 8 bits to 16 bits.

The extension operation is different for unsigned and signed numbers.

Unsigned Numbers: The high-order part of the binary number is filled with "0"s.

Example: 4-bit 3_{10} = `0011` 8-bit 3_{10} = `0000 0011`
Example: 4-bit 9_{10} = `1001` 8-bit 9_{10} = `0000 1001`

Signed Numbers: The high-order part of the binary number is filled with the value of the sign bit. This operation is called **sign extension**.

Example: 4-bit $+3_{10}$ = `0011` 8-bit $+3_{10}$ = `0000 0011`
Example: 4-bit -7_{10} = `1001` 8-bit -7_{10} = `1111 1001`
Example: 4-bit -1_{10} = `1111` 8-bit -1_{10} = `1111 1111`

Hexadecimal (Base-16) Numbers

Binary numbers are necessary because digital circuits can directly process binary values.

However, it is difficult for humans to work with large numbers of bits in even relatively small binary numbers. (Example: 1110010001011010)

Therefore, for documentation (to write and read easily), *hexadecimal* (base-16) numbers are used.

Converting from binary to hexadecimal is easy:

- Each hex. digit maps to 4 bits. See the table.
- Separate binary number into groups of 4 bits.
- Find the single hexadecimal digit that corresponds to each group of 4 bits.

Example:

01011101_2 = 0101 1101 (Binary)
= 5 D (Hexadecimal)

Example:

$87 = $1000\ 0111_2$

Hexadecimal → Binary

Decimal	Binary	Hex.
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

To express hexadecimal numbers, the symbols **$** and **h** are usually used.
Example: $5D , $87 or 5Dh , 87h.

Addition and Subtraction Operations in Digital Systems

In computers, the *Arithmetic Logic Unit* (ALU) performs the integer arithmetic operations.

As an advantage of 2's complement representation, integer addition and subtraction operations are performed on unsigned and signed numbers in the same way.

However, the result is interpreted differently for unsigned and signed numbers.

Before an operation on numbers of different lengths, a sign extension is necessary.

Remember: The extension operations are different for unsigned and signed numbers.

Addition:

The result of the addition of two bits (a and b) is a 2-bit number.

The LSB of the result is the 1-bit sum, and the MSB is the carry.

The rules of binary addition are given in the table on the right.

a b	Carry	Sum
0+0	0	0
0+1	0	1
1+0	0	1
1+1	1	0

Addition:

Unsigned Integers:

The result of the addition of two n-bit unsigned numbers can be a (n+1)-bit number. The $(n+1)^{st}$ bit is called the **carry**.

Examples: Addition of two 8-bit unsigned numbers

```
   01110101 : 117
 + 01100011 :  99     Only for testing
   11011000 : 216
No Carry (Carry=0)
```

```
   11111111 : 255
 + 00000001 :   1     Only for testing
  1 00000000 : 256+0
(Carry=1)
```

The 8-bit result is 0.
However, using the carry, we can interpret the result as 256.
$2^8 + 0 = 256$

Addition:

Signed Integers:

- The operation is performed as with unsigned numbers, but the result is interpreted differently.
- As an advantage of 2's complement representation, adding 2's-complement numbers requires no special processing even if the operands have opposite signs.
- If an $(n+1)^{st}$ bit arises as a result of adding two n-bit signed numbers, this bit is ignored.

Examples: Addition of 8-bit signed numbers

```
   11111111 :  -1
 + 00000001 :  +1
  100000000 :   0
Ignored  Sign (+)
```

```
   11111111 :  -1
 + 11111111 :  -1
  111111110 :  -2
Ignored  Sign (-)
```

Attention:

- While working with n-bit numbers, the sign bit is always the most significant bit (counting from right to left), i.e., the n^{th} bit, not the $(n+1)^{st}$ one.
- The $(n+1)^{st}$ bit is the carry bit.

Overflow (signed integers):

The result of the addition of n-bit signed numbers can be too large for the binary system to represent (greater than n bits).

For example, with 8 bits, we can represent signed numbers between -128 and +127. If the result is out of this range, an **overflow** occurs.

The existence of overflow after addition can be detected by checking the signs of the operands and the result.

In an addition operation, overflow can occur in two cases:

positive + positive $\rightarrow$ negative and negative + negative $\rightarrow$ positive

Examples:

```
  01111111:+127
+ 00000010:  +2
  10000001: Cannot be represented
```

Both operands are positive.

The result is negative.

There is an **overflow**.

Note: $(n+1)^{st}$ bit is zero.
It is ignored.

```
  10000000:-128
+ 11111111:  -1
 101111111:Cannot be represented
```

Both operands are negative.

The result is positive.

There is an **overflow**.

Note: $(n+1)^{st}$ bit is one.
It is ignored.

Examine the program `integer_overflow.cpp`

Subtraction:

- Computers usually use the method of complements to implement subtraction.
- 2's complement of the second operand is added to the first number.

$$A - B = A + (-B)$$
$$= A + 2's\ complement\ (B)$$
$$= A + \overline{B} + 1$$

So, only one addition circuit is sufficient to perform both addition and subtraction (benefit of 2's complement system).

In Section 5, we will cover addition and subtraction circuits.

Like addition, subtraction operations are performed on unsigned and signed numbers in the same way (because of 2's complement representation).

However, the interpretation of the result is different for unsigned and signed numbers.

Subtraction (cont'd):

Unsigned Integers:

If the result of subtracting two n-bit unsigned numbers, performed using 2's complement, is a (n+1)-bit number (there is a carry), then there is no borrow, and the result is valid.

If the $(n+1)^{st}$ bit of the result is zero (no carry), the first operand is smaller than the second, and there is **a borrow**.

Carry = 1 → no borrow
Carry = 0 → borrow

Examine the program `borrow.cpp`

Examples: Subtraction of 8-bit unsigned numbers

```
  00000101: 5                       00000101: 5
- 00000001: 1   2's complement →  + 11111111:-1
                                   100000100: 4
```

Carry=1 : **No Borrow**

```
  00000001: 1                       00000001: 1
- 00000101: 5   2's complement →  + 11111011:-5
                                   011111100:
```

Cannot be represented

No Carry: **Borrow**

Subtraction (cont'd):

Signed Integers:

Subtraction on signed integers is also performed using 2's complement.

The carry bit is ignored.

Example:

```
  00000101: +5                       00000101: +5
- 00001100:+12   2's complement →  + 11110100:-12
                                     11111001:   2's comp.: 00000111:-7
```

To determine its decimal equivalent, we take its 2's complement first

Sign **1**, result: **negative**

Overflow: Just as in addition, an overflow can occur in subtracting signed numbers. In subtraction, overflow can occur in two cases:

pos. – neg. → neg. and neg. – pos. → pos.

Example:

```
  11111101: -3                        11111101:   -3
- 01111111:+127   2's complement →  + 10000001:-127
                                     101111110:
```

cannot be represented

Neg. – pos. = pos. ⇒ **Overflow**

Sign:**0**, result: **positive**.

Comparing integers:

The subtraction operation is used to compare two integers.

After the operation **R = A - B**, related flags (status bits: carry, overflow) are checked.

Unsigned Integers:

Result (R)	Carry , Borrow	Comparison
=0	X (not important)	A=B
≠0	Carry = 1, NO	A>B
≠0	Carry = 0, YES	A<B

Signed Integers:

Result (R)	Overflow	Comparison
=0	X (not important)	A=B
Positive, ≠0	NO	A>B
Negative	NO	A<B
Positive	YES	A<B
Negative	YES	A>B

Because of overflow, the sign of the result is not correct.

Summary of Carry, Borrow, Overflow

Carry: It can occur in the addition of unsigned numbers.

It indicates that the result cannot be represented with n bits, and an (n+1)st bit is necessary.

Borrow: It can occur in the subtraction of unsigned numbers (A - B).

It indicates that the first number (A) is smaller than the second one (B, the number being subtracted), i.e., A < B.

The result cannot be represented with unsigned numbers.

If the result of the subtraction using 2's complement is an n-bit number (no carry), then there is a borrow, and the result is invalid.

Overflow: It can occur only on signed numbers in addition and subtraction operations.

It indicates that the result cannot be represented with n bits.

Overflow can be detected by checking the signs of operands and the result.

There is an overflow in the following cases:

pos. + pos. → neg.   pos. – neg. → neg.

neg. + neg. → pos.   neg. – pos. → pos.

Summary of Carry, Borrow, Overflow

	Unsigned		Signed
Event	Carry	Borrow	Overflow
Affected Operation	Addition n-bit integers	Subtraction n-bit integers	Addition/Subtraction n-bit integers
Meaning	The result cannot be represented with n bits. Carry can be used to interpret the result.	A < B, The result cannot be represented with unsigned numbers.	The result cannot be represented with n bits. The sign of the result is inverted.
Detection	(n+1)st bit of the result is 1. Carry = 1	If the result is an n-bit number (Carry=0), there is a borrow, and the result is invalid.	Check the signs of operands and the result pos. + pos. -> neg. pos. – neg. -> neg. neg. + neg. -> pos. neg. – pos. -> pos.