

Representation of Numbers

DSC 315: Computer Organization & Operating Systems

Dr. Laltu Sardar

School of Data Science,
Indian Institute of Science Education and Research Thiruvananthapuram (IISER TVM)



9th January, 2026



Topics to Be Covered

- 1 Representation of Unsigned Integers**
- 2 Representation of Signed Integers**
 - Sign-Magnitude Representation
 - 1's Complement Representation
 - 2's Complement Representation
 - Excess Representation
- 3 Representation of Floating-Point Numbers**
- 4 Addition, Subtraction, Multiplication, and Division for**
 - Unsigned Integers
 - Signed Integers in 2's Complement Form
 - Floating-Point Numbers
- 5 For each arithmetic operation, we study**
 - Algorithm
 - Flowchart
 - Hardware Circuit Diagram

Background and References

Follow Ref. [1] for representations of integers

Key sections to focus on:

Sections 3.1 to 3.6 and Section 3.9

Unsigned Integers

Unsigned Integer Representation

Unsigned integers represent **non negative integers only** and use all bits for magnitude.

For an n bit unsigned integer:

$$\text{Value} = \sum_{i=0}^{n-1} b_i 2^i$$

- No sign bit is used
- Each bit contributes a positive power of two
- All bit patterns correspond to valid values

Range:

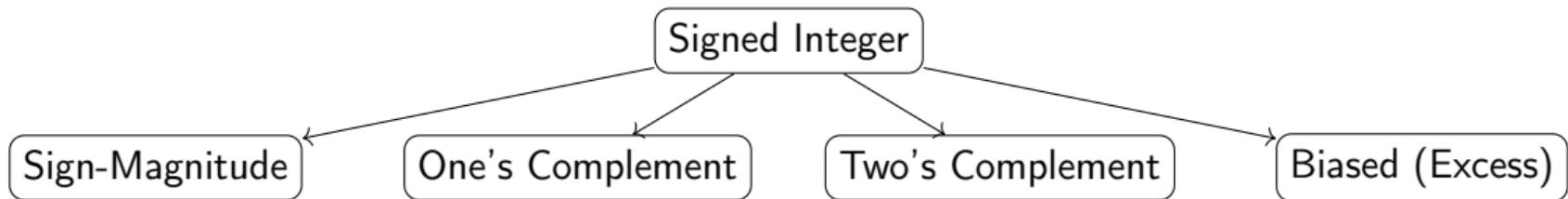
$$0 \leq \text{Value} \leq 2^n - 1$$

Example (8 bit unsigned integer):

- Minimum: $0000000_2 = 0$
- Maximum: $1111111_2 = 255$

Signed Integers

Signed Integer Representations



Sign-Magnitude Representation

- The most significant bit (MSB) represents the sign
 - 0 indicates a positive number
 - 1 indicates a negative number
- The remaining bits store the magnitude in unsigned binary
- Example using 8-bit representation:
 - $+5 \rightarrow 00000101$
 - $-5 \rightarrow 10000101$
- Drawbacks:
 - Two representations of zero: $+0$ and -0
 - Arithmetic operations are complex on hardware. Why?
 - Rarely used in modern computer systems

One's Complement Representation

- Positive numbers are represented using standard binary notation
- Negative numbers are obtained by taking the bitwise complement of the positive value
- Example using 8-bit representation:
 - $+5 \rightarrow 00000101$
 - $-5 \rightarrow 11111010$
- Key properties:
 - Two representations of zero: 00000000 and 11111111
 - Addition may require an end-around carry
- Largely obsolete, replaced by two's complement in modern systems

Signed Integer Representation (Two's Complement)

Signed integers represent **both positive and negative integers** using two's complement.

For an n bit signed integer:

$$\text{Value} = \begin{cases} \sum_{i=0}^{n-1} b_i 2^i, & \text{if } b_{n-1} = 0 \\ -2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i, & \text{if } b_{n-1} = 1 \end{cases}$$

- Most significant bit is the sign bit
- Positive numbers are represented as usual binary
- Negative numbers are represented using two's complement

Range: $-2^{n-1} \leq \text{Value} \leq 2^{n-1} - 1$

Example (8 bit signed integer):

- Maximum: $01111111_2 = +127$; Minimum: $10000000_2 = -128$

Conversion Table with 2's Complement

Decimal	Bin	Flipped Bits	2's Complement	-ve
0	0000 0000	1111 1111	0000 0000	0
1	0000 0001	1111 1110	1111 1111	-1
2	0000 0010	1111 1101	1111 1110	-2
64	0100 0000	1011 1111	1100 0000	-64
126	0111 1110	1000 0001	1000 0010	-126
127	0111 1111	1000 0000	1000 0001	-127
128	1000 0000	0111 1111	1000 0000	-128

Table: Conversion Table with 2's Complement

Overflow and Underflow in Integer Arithmetic

Overflow

- Occurs when the result of an integer operation exceeds the **maximum representable value**
- Example: adding two large positive integers
- In fixed width integers, overflow may cause **wrap around**

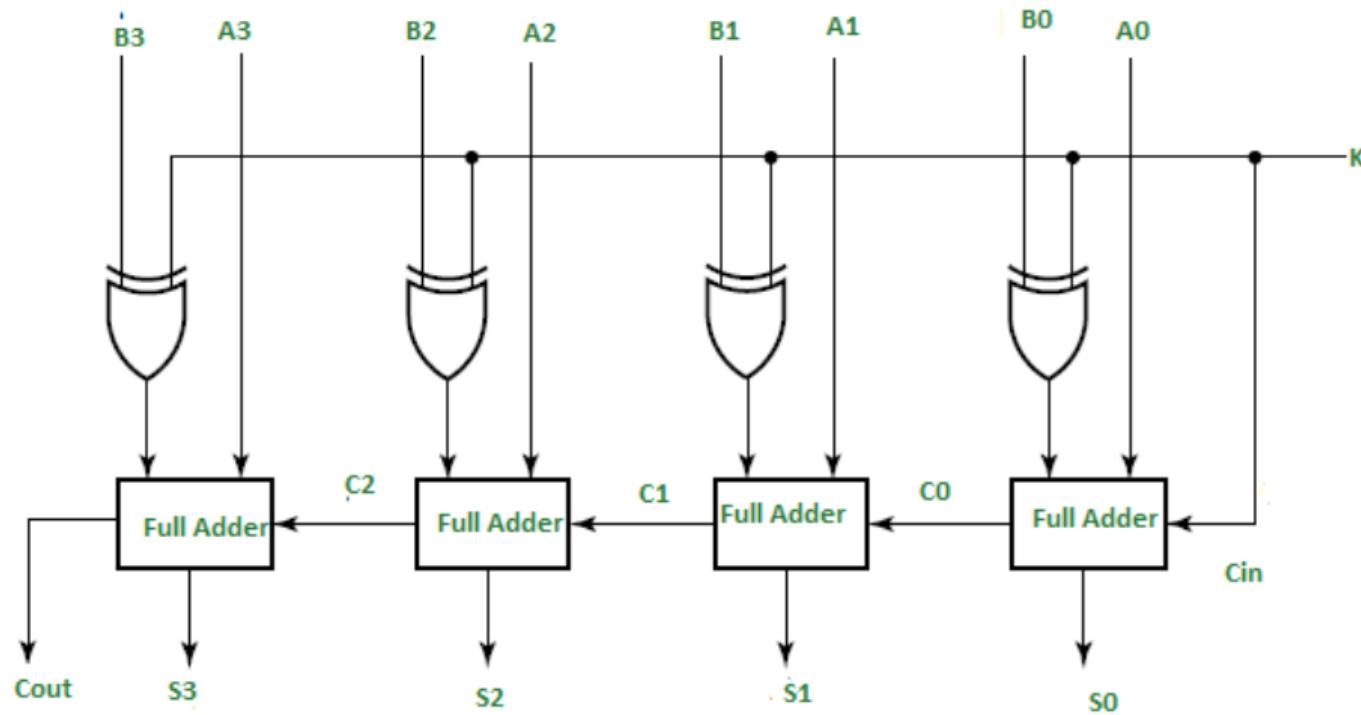
Underflow

- Occurs when the result of an integer operation goes below the **minimum representable value**
- Example: subtracting a large integer from a smaller one
- In fixed width integers, underflow may also cause **wrap around**

Note

- Behavior depends on whether integers are **signed or unsigned**

Circuit for Binary number addition (2's Complement)



Homework

- Consider the decimal values 123, -123, 73, -73, 0, -0 (or number of your choice). Finds their

- 1 Unsigned representation
- 2 Sign-Magnitude representation
- 3 1's complement representation
- 4 2's complement representation

Find their representations in both 8-bits and 32-bits.

- Consider the following binary numbers (or number of your choice)
 - a) 1100 1010 ; b) 0101 0101 c) 0111 0010 d) 1111 1010
 - f) 0000 0000 0000 0000 0000 1100 1010
 - g) 1111 1111 1111 1111 1111 1111 1010

Find their decimal values considering them as a) Unsigned representation, b) Sign-Magnitude representation, c) 1's complement representation and d) 2's complement representation

Biased (Excess) Representations

Biased (Excess) Representation

- Values are stored with a fixed bias added to the true integer value
- Actual value is obtained by subtracting the bias from the stored value
- General form:

$$\text{Stored Value} = \text{Actual Value} + \text{Bias}$$

- Example using Excess-127:
 - Stored value 130 represents the actual value 3
- Use cases:
 - Exponent field in IEEE 754 floating-point representation
 - Simplifies comparison operations between signed values
 - Avoids explicit sign bit handling in hardware

Biased (Excess) Representation: Example with Binary

Code	Value
<hr/>	
00000000	-127 <--- smallest negative value with 8 bits (-2^7)
00000001	-126
<hr/>	
01111000	-7
01111001	-6
01111010	-5
01111011	-4
01111100	-3
01111101	-2
01111110	-1
01111111	0
10000000	1
10000001	2
10000010	3
10000011	4
10000100	5
10000101	6
10000110	7
10000110	8
10000111	9
10001000	10
<hr/>	
11111111	128 <--- largest positive value with 8 bits (2^7-1)

- minimal negative value is represented by all-zeros

Hexadecimal Numbers

Hexadecimal Number System

- Hexadecimal is a base-16 number system.
- Digits used: 0–9 and A–F (A=10, B=11, ..., F=15).
- One hexadecimal digit represents exactly 4 binary bits.
- Hexadecimal is widely used to compactly represent binary data.

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

Why Do We Need Hexadecimal Numbers?

- Computers work internally using binary numbers.
- Binary representation is long and difficult for humans to read.
- Hexadecimal provides a compact representation of binary data.
- One hexadecimal digit corresponds to exactly 4 binary bits.
- Conversion between binary and hexadecimal is simple and direct.
- Hexadecimal is widely used for:
 - memory addresses
 - machine instructions
 - debugging and low-level programming

Example: Decimal to Binary to Hexadecimal

Step 1: Decimal to Binary

$$24015_{10} = 0101110111001111_2$$

Step 2: 32-bit Binary Representation

00000000 00000000 01011101 11001111

Step 3: Partition into 4-bit Groups

0000 0000 0000 0000 0101 1101 1100 1111

Step 4: Hexadecimal Representation

0000 0000 0000 0000 0101 1101 1100 1111 \Rightarrow 0x00005DCF

Important Conceptual Questions on Integer Representations

- Why is **two's complement** preferred over sign magnitude and one's complement?
- Why does two's complement allow **simple and uniform arithmetic** (addition and subtraction)?
- Why is the signed integer range **asymmetric** in two's complement?
- Why does overflow behave **differently** for signed and unsigned integers?
- Why is the most significant bit treated as the **sign bit** in signed integers?

Homework: Integer Representations

- 1 Write a C program to print the maximum value of an unsigned int.
 - limits.h must **not** be used.
 - Only stdio.h is allowed.
- 2 Write a C program to print the maximum and minimum values that a signed int can hold.
Do not use predefined constants or headers for limits.
- 3 Given a set of decimal numbers: +260109, -2601, +109, -901062, +0, -1
 - convert them to unsigned binary representation (if \neq -ve)
 - convert them to one's complement form
 - convert them to two's complement form
 - convert them to excess-127 representation
- 4 Write a C program to detect signed and unsigned overflow during addition without using any built-in overflow checking functions.
- 5 Given a binary number, identify whether it represents a valid two's complement number and determine its decimal value.

Floating Point Numbers

Floating Point Representation Formula (Normal Numbers)

For a **normalized IEEE 754 single precision floating point number**:

$$(-1)^s \times (1.f) \times 2^e$$

- s : sign bit (0 for positive, 1 for negative)
- f : fraction field (23 bits)
- E : stored exponent (8 bits)
- Bias : 127 (Excess-127))
- Actual exponent:

$$e = E - 127$$

Conditions for normal numbers:

- $1 \leq E \leq 254$
- Implicit leading bit is 1
- Exponent values $E = 0$ and $E = 255$ are reserved for special cases

Extremal Values in IEEE 754 Single Precision (32-bit)

■ Highest positive normal

- Binary:

0 11111110 11111111111111111111111111111111

- Decimal:

$$(2 - 2^{-23}) \times 2^{127} \approx 3.4028235 \times 10^{38}$$

■ Lowest positive normal

- Binary:

0 00000001 00000000000000000000000000000000

- Decimal:

$$2^{-126} \approx 1.17549435 \times 10^{-38}$$

Floating Point Representation Formula (Subnormal Numbers)

For a **subnormal IEEE 754 single precision floating point number**:

$$(-1)^s \times (0.f) \times 2^e$$

- s : sign bit (0 for positive, 1 for negative)
- f : fraction field (23 bits)
- E : stored exponent (8 bits)
- Bias : 127
- Actual exponent:

$$e = -126$$

Conditions for subnormal numbers:

- $E = 0$
- Fraction field is **non zero**
- No implicit leading 1 in the significand
- Allows gradual underflow toward zero

Extremal Subnormal Values in IEEE 754 Single Precision (32-bit)

■ Highest positive subnormal

- Binary:

0 00000000 11111111111111111111111111111111

- Decimal:

$$(1 - 2^{-23}) \times 2^{-126} \approx 1.17549421 \times 10^{-38}$$

■ Lowest positive subnormal

- Binary:

0 00000000 00000000000000000000000000000001

- Decimal:

$$2^{-149} \approx 1.40129846 \times 10^{-45}$$

Special Numbers in IEEE 754 Single Precision (32-bit)

- Positive Zero (+0)

- Binary:

0 00000000 00000000000000000000000000000000

- Negative Zero (-0)

- Binary:

1 00000000 00000000000000000000000000000000

- Positive Infinity ($+\infty$)

- Binary:

0 11111111 00000000000000000000000000000000

- Negative Infinity ($-\infty$)

- Binary:

1 11111111 00000000000000000000000000000000

- NaN (Not a Number)

- Binary:

Sign 11111111 Fraction $\neq 0$

Important Conceptual Questions on Floating Point Representation

- Why is the exponent stored using a **bias** instead of two's complement?
- Why is the **bias equal to 127** in single precision?
- Why are the exponent values $E = 0$ and $E = 255$ **reserved**?
- Why do floating point numbers have **two zeros** (+0 and -0)?
- Why does the range of positive and negative values become **asymmetric**?
- Why are **subnormal numbers** needed?
- Why is the exponent for subnormal numbers **-126 and not -127**?
- Why is there an **implicit leading 1** in normal numbers?
- Why is **NaN** needed instead of signaling an error?
- Why are floating point numbers **not closed under arithmetic**?

Addition of Unsigned Integers

Unsigned Binary Addition

- Unsigned integers are stored in binary.
- Addition is performed bit by bit from LSB to MSB.
- Each bit addition produces:
 - a sum bit
 - a carry to the next higher bit

Unsigned Overflow

- Unsigned overflow occurs when the true sum exceeds $2^n - 1$.
- In hardware, overflow happens when there is:
 - a carry out of the most significant bit (MSB)

Example (4-bit):

$$1111_2 + 0001_2 = 1\ 0000_2$$

Stored result: 0000_2

Overflow Detection

- Unsigned overflow is detected using the carry out of the MSB.
- CPUs store this information in a status flag:
 - Carry Flag (CF)
- If CF = 1, unsigned overflow has occurred.

How the Machine Handles Overflow

- The CPU does not stop execution on unsigned overflow.
- Result wraps around modulo 2^n .
- It is the responsibility of software to:
 - check the carry flag
 - take corrective action if required

Addition of Signed Integers: 2's Complement

Signed Number Representation

- Signed integers are represented using two's complement.
- Signed addition is performed using ordinary binary adders.
- The same hardware is used for signed and unsigned addition.
- Bits are added from LSB to MSB with carry propagation.
- Interpretation of the result depends on two's complement rules.

Example (4-bit):

$$0101_2 + 1101_2 = 0010_2$$

When Does Signed Overflow Occur?

- Signed overflow occurs when the true result is outside the representable range.
- Overflow occurs only in the following cases:
 - positive + positive = negative
 - negative + negative = positive
- Adding numbers with different signs never causes overflow.

Signed Overflow Detection

- Carry out of MSB alone cannot detect signed overflow.

- Let:

- C_{in} be carry into MSB
 - C_{out} be carry out of MSB

- Signed overflow condition:

$$C_{in} \oplus C_{out} = 1$$

- Equivalent sign-based rule:

- operands have same sign
 - result has different sign

Machine Handling of Signed Overflow

- CPU records signed overflow using the Overflow Flag (OF).
- OF = 1 indicates signed overflow.
- The processor does not halt on overflow.
- Result wraps around modulo 2^n .
- Software must explicitly check and handle overflow.

Final statements

- Signed numbers use two's complement representation.
- Same binary adder is used for signed and unsigned addition.
- Signed overflow depends on operand signs, not carry alone.
- Overflow detection uses carry into and out of MSB.
- Handling overflow is the responsibility of software.

Multiplying unsigned integers

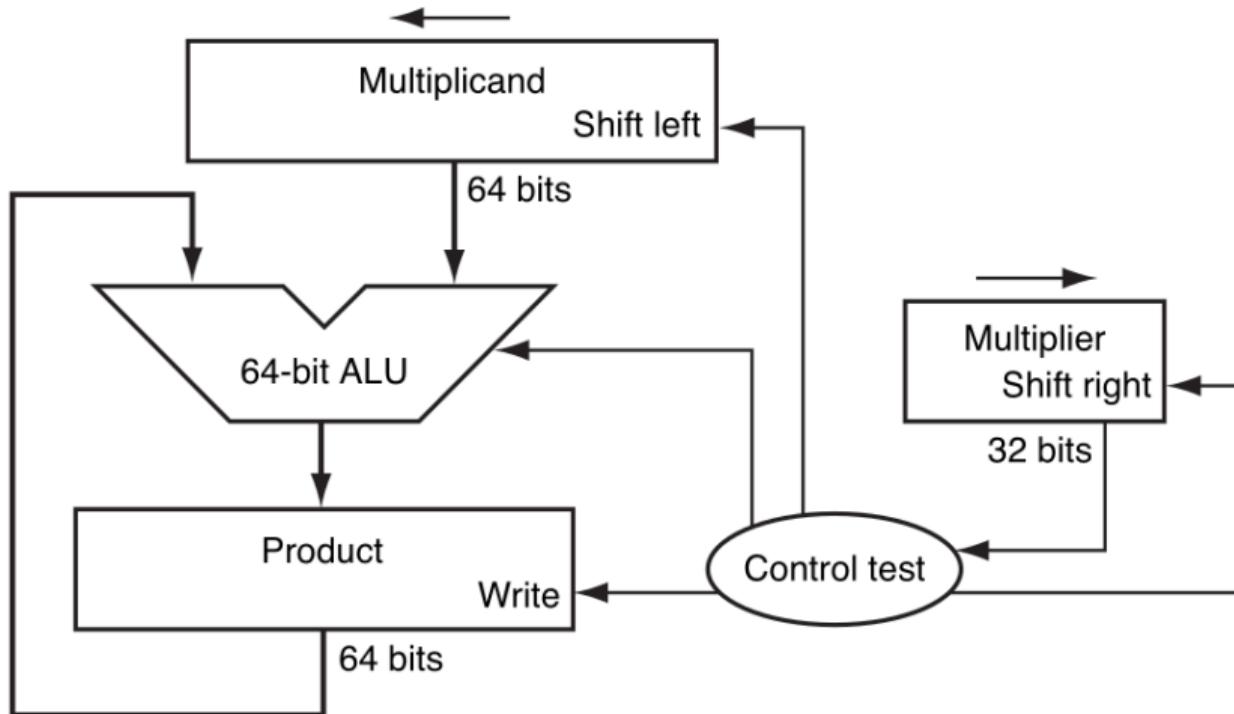
Topics

- 1 Assume you only have adder: wire and XOR gate, other basic gate, we studied the followings
- 2 **Algorithm to multiply,**
- 3 Flowchart of the algorithm (Fig 3.5, Page 232, Ref [1], Volume 4)
- 4 Circuit logic (Fig 3.4, Page 231, Ref [1], Volume 4)

Multiplying signed integers

For details, see page 234 of Ref [1], Volume 4

Circuit Logic: Integer Multiplication



Observations

What if overflow occurs? Hardware ignores; Software designer's responsibility

Multiplying Signed Integers

So far, we have dealt with **positive numbers**. To handle **signed numbers**, the idea is to separate the **sign handling** from the **magnitude computation**.

- First convert the **multiplier** and **multiplicand** to positive values
- Record the original signs separately
- Run the multiplication algorithm for **31 iterations**, ignoring signs during computation
- Negate the final product **only if the original signs differ**

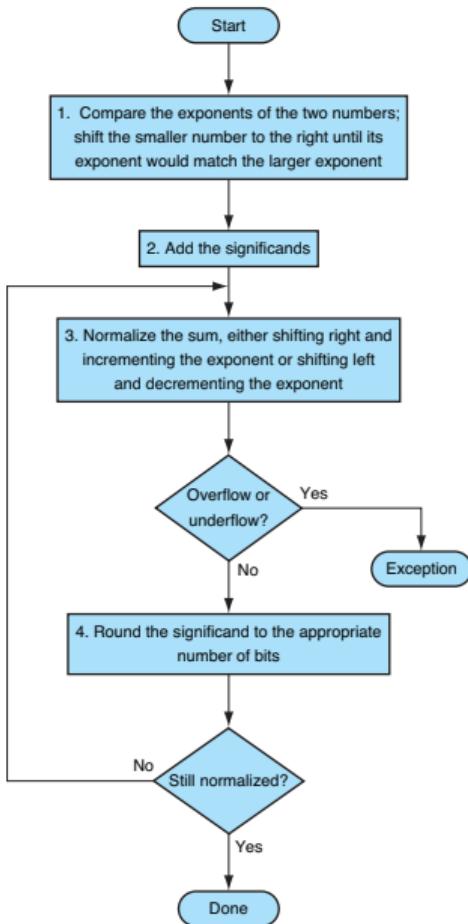
It turns out that the same multiplication algorithm works for signed numbers if we remember that:

- Signed numbers are conceptually represented using **infinite sign extended bits**
- In practice, we store only **32 bits**
- During shifting, the sign bit must be **extended** (arithmetic shift)

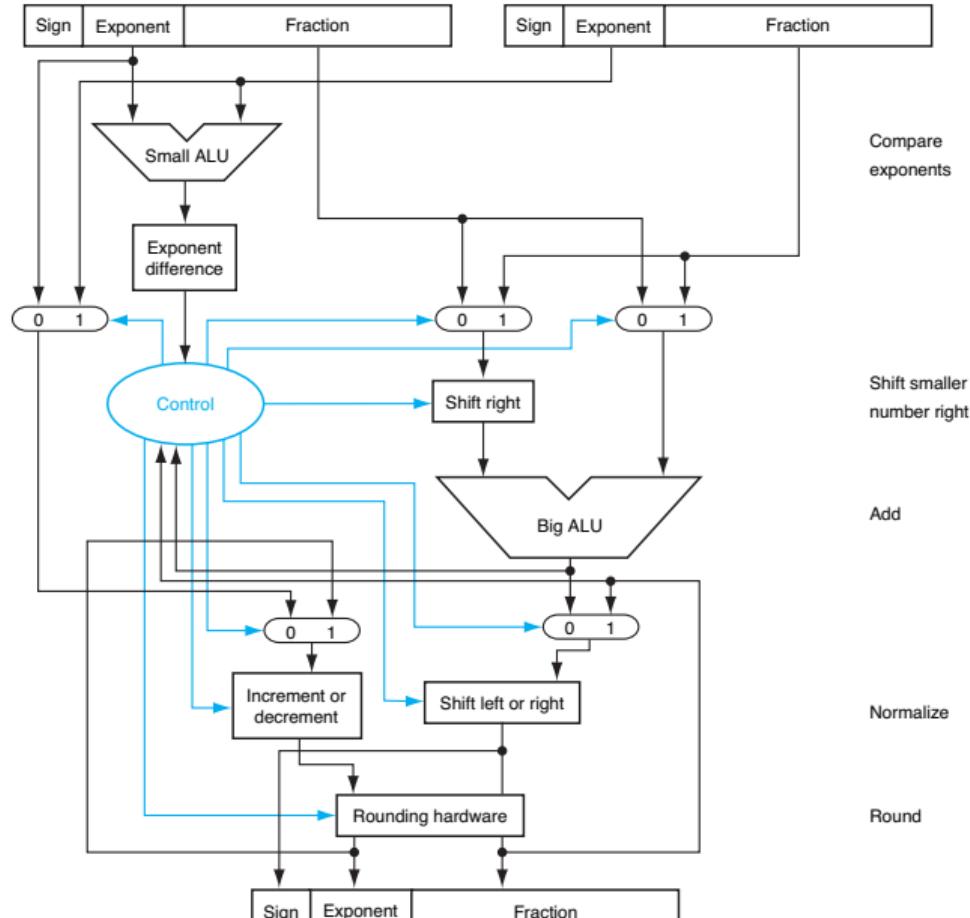
Final result:

- After completion, the **lower word** contains the 32 bit signed product

Addition of floating point numbers

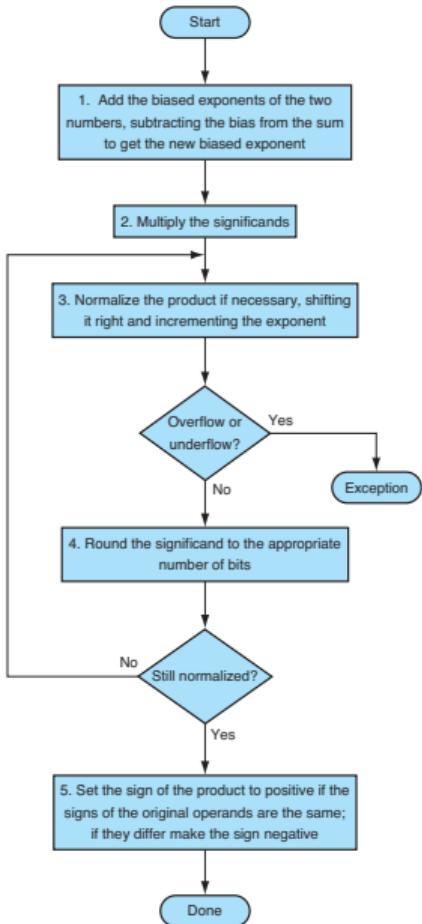


Addition of Floats



Subtraction of floating point numbers

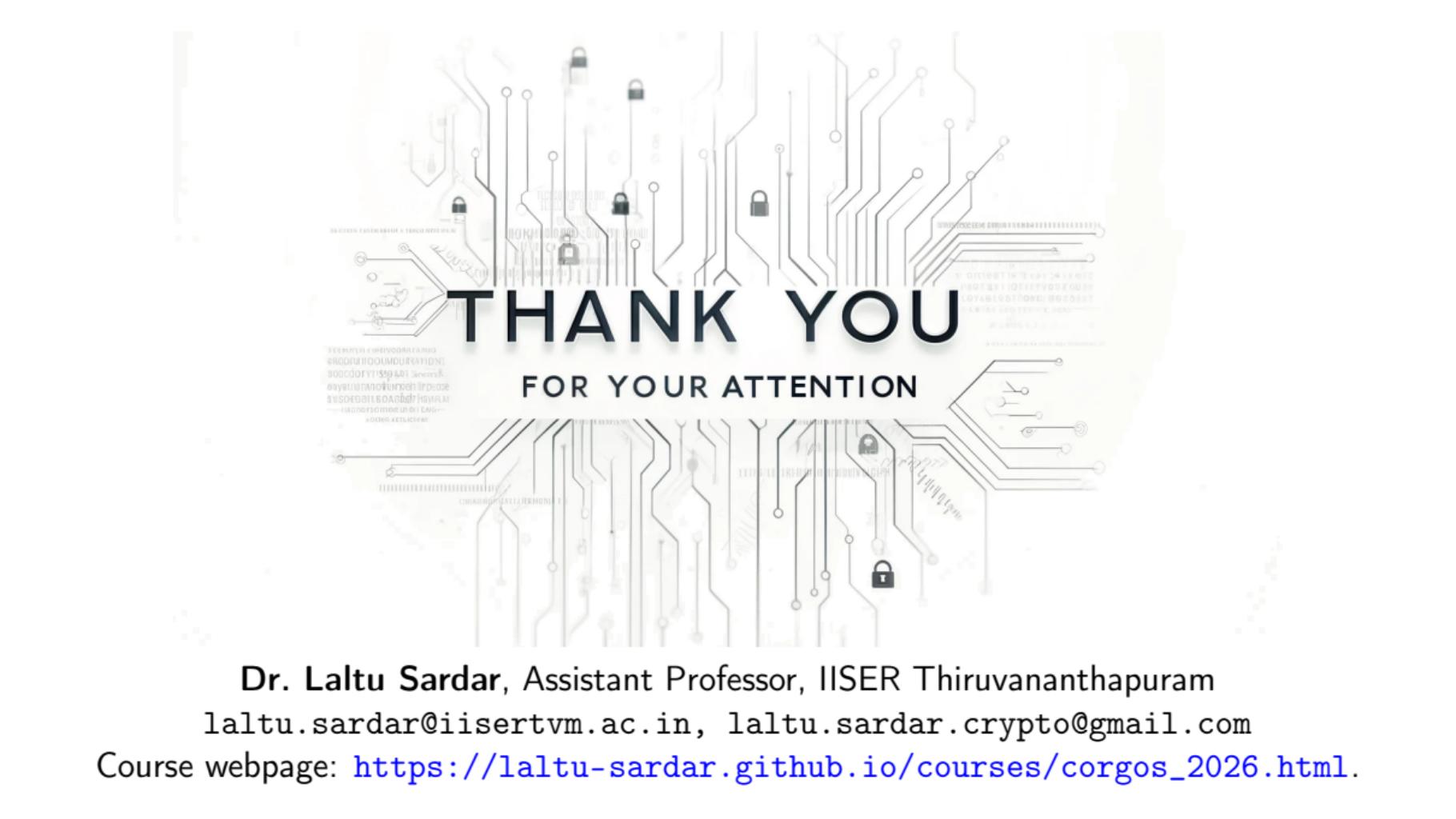
Multiplication of floating point numbers



Division of floating point numbers

Algorithm for Unsigned Binary Addition

- 1 Initialize the carry to 0.
- 2 Start from the least significant bit of both binary numbers.
- 3 Add the corresponding bits and the carry.
- 4 The sum bit is the result modulo 2.
- 5 Update the carry as follows:
 - Carry = 1, if the sum is 2 or 3
 - Carry = 0, otherwise
- 6 Move to the next higher bit and repeat until all bits are processed.
- 7 If a carry remains after the most significant bit, append it to the result.



THANK YOU

FOR YOUR ATTENTION

Dr. Laltu Sardar, Assistant Professor, IISER Thiruvananthapuram

laltu.sardar@iisertvm.ac.in, laltu.sardar.crypto@gmail.com

Course webpage: https://laltu-sardar.github.io/courses/corgos_2026.html.

-  Carl Hamacher, Zvonko Vranesic, and Safwat Zaky.
Computer Organization.
McGraw-Hill, 6th edition, 2012.
-  John L. Hennessy and David A. Patterson.
Computer Architecture: A Quantitative Approach.
Morgan Kaufmann, 6th edition, 2017.
-  Vincent P. Heuring and Harry F. Jordan.
Computer System Design and Architecture.
Pearson, 2nd edition, 2007.
-  David A. Patterson and John L. Hennessy.
Computer Organization and Design: The Hardware/Software Interface.
Morgan Kaufmann, 4th edition, 2014.
-  William Stallings.
Computer Organization and Architecture: Designing for Performance.
Pearson, 10th edition, 2016.