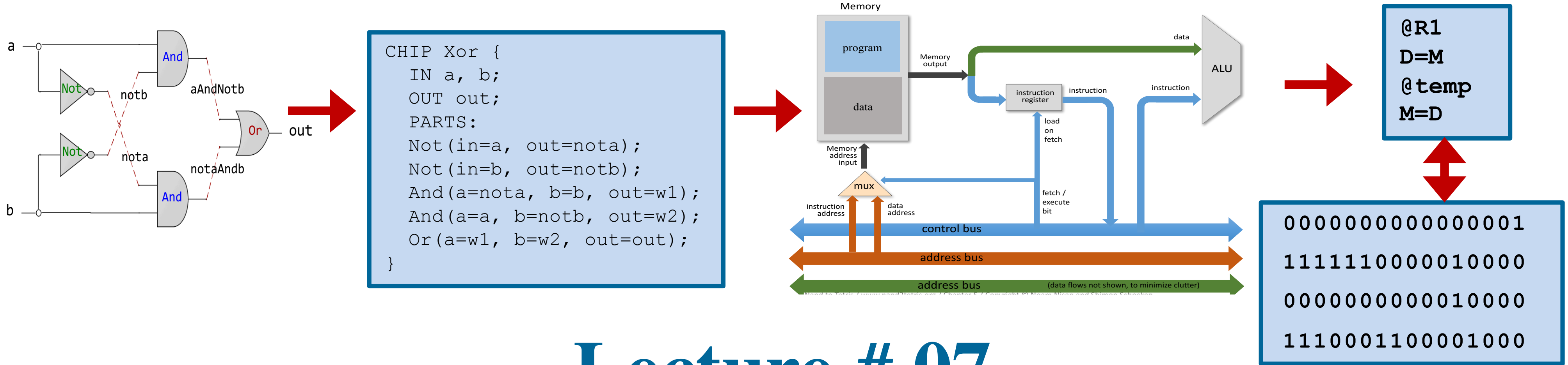
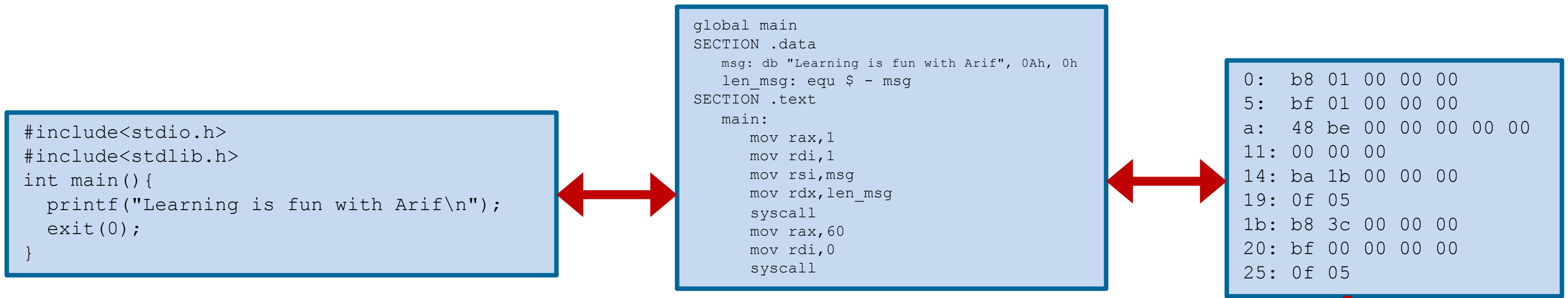
Computer Organization & Assembly Language Programming

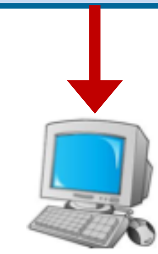


Lecture # 07

Data Storage - II



Slides of first half of the course are adapted from:
https://www.nand2tetris.org
Download s/w tools required for first half of the course from the following link:
https://drive.google.com/file/d/0B9c0BdDJz6XpZUh3X2dPR1o0MUE/view



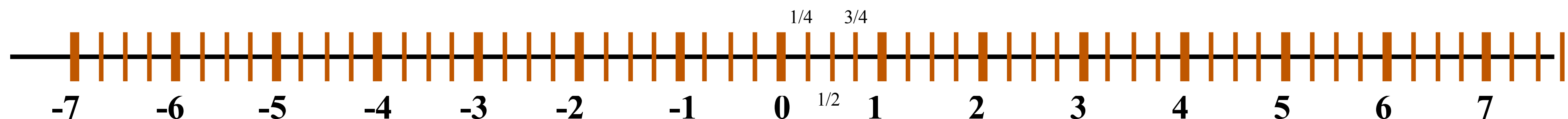
Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

- Encoding Real Numbers
- Fixed Point Representation
- Floating Point Representations (IEEE-754)
  - Storage layout
  - Conversion Examples
  - Range and Precision
  - Arithmetic Operations
  - Overflow and Underflow
  - IEEE-754 Special Values

Encoding Real Numbers

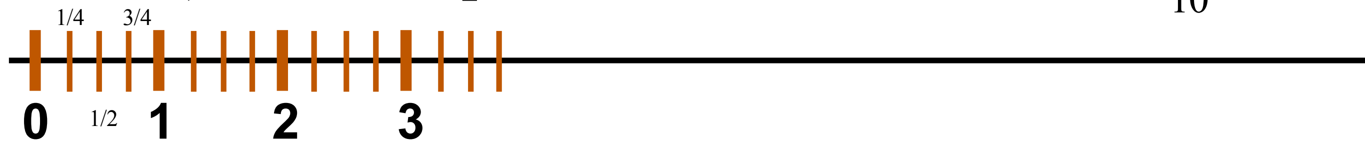
- Most scientific computations are performed using real numbers, i.e., numbers with a fractional part. In order to represent real numbers in computers, we have to ask two questions:
  - ➢ How many bits are needed to encode a real number?
  - ➢ How to represent a real numbers using these bits?
- There are two ways to encode the real numbers:
  - ➢ Fixed Point Representation
  - ➢ Floating Point Representation



Fixed Point Representation

Unsigned Fixed Point Representation (5 bits)

- Fixed point representation fixes the position of binary point within a register. Therefore fixes the sizes of the whole number and the fractional part of any real number stored inside it
- Let us use **five** bits to represent unsigned real numbers by keeping **three** bits for the integral part and **two** bits for the fractional part (as shown in table)
- **Range** (max/min values) for this representation is 0 to 7
- **Precision** (smallest distance between two successive numbers) for this representation is $2^{-2} = \frac{1}{4} = 0.25_{10}$



- Conversion:

  $001.01_2 = 0x2^2 + 0x2^1 + 1x2^0 + 0x2^{-1} + 1x2^{-2} = 1.25_{10}$

  $6.75_{10} = 110.11_2$

- Integer Arithmetic Work, and there is no need to align binary point!

```
    001.10          1.50
+)  010.01      +)  2.25
    ------          ----
    011.11          3.75
```

Decimal	Binary
0.0	000.00
0.25	000.01
0.5	000.10
0.75	000.11
1.0	001.00
1.25	001.01
1.5	001.10
1.75	001.11
2.0	010.00
…	…
…	…
6.75	110.11
7.0	111.00
7.25	111.01
7.5	111.10
7.75	111.11

Signed Fixed Point Representation (16 bits)

- Let us use 16 bits to represent signed fractional numbers by keeping one bit for the sign, 11 bits for the integer part and 4 bits for the fractional part

S	Integer part (11)	Fractional part (4)

- **Range** for this representation:
  - From: $-[(2^{11} - 1) + (1 - 2^{-4})] = -2047.9375_{10}$

1	11111111111	1111

  - To: $(2^{11} - 1) + (1 - 2^{-4}) = +2047.9375_{10}$

0	11111111111	1111

- **Precision** for this representation: $2^{-4} = 1/16 = 0.0625_{10}$

0	00000000000	0001

- **Advantage:** Arithmetic and logical operations can be performed on fixed point numbers using integer arithmetic (Performance). No FPU required

Limitations of Fixed Point Representation

- To represent very large numbers, we need to give more bits to the integer part and less bits to the fractional part. This will make it hard to represent small numbers
- Similarly to represent very small numbers, we need to give more bits to the fractional part and less bits to the integer part. This will make it hard to represent large numbers

S	Integer part (11)	Fractional part (4)

1. **Limited integer range.** It is not possible to represent very large and very small numbers with the same representation
2. **Small precision,** e.g., keeping 4 bits for the fractional part, the smallest fraction that can be represented is 1/16
3. It is easy for an arithmetic operation to produce an overflow and underflow
   - **Overflow** will occur when the result is large to fit in the representation (+/- 2047.9375 for 16 bit representation)
   - **Underflow** will occur when the result is too small to fit in the representation, e.g., if the result of the arithmetic is less than 0.0625. It can't be represented in above scheme

Floating Point Representation

Floating Point Representation

- The concept of floating point binary numbers was introduced in the mid 1950s. It uses scientific notation for storing the real numbers, thus increasing the range as well as precision of real numbers

$$+/- M \times 2^{+/-E}$$

Mantisssa (8)	Exponent (8)

- One of the possible 16 bit representation is shown above. The mantissa with its sign, and the exponent with its sign were saved in 2s complements format in 8 bits each
- There was no uniformity in the formats used to represent floating point numbers and programs were not portable from one manufacturer's computer to another
- To resolve the issue, a Standards Committee was formed by the Institute of Electrical and Electronics Engineers (IEEE) to standardize how floating point binary numbers would be represented in computers
- This standard, called IEEE Standard 754 for floating point numbers, was adopted in 1985 by all computer manufacturers. It allowed porting of programs from one computer to another without the answers being different. The standard was updated in 2008 and then in 2019. The current standard is IEEE 754-2019

IEEE-754 Standard for Floating Point Representation

IEEE 754 Standard for FP Representation

- A 32 bit floating point representation using IEEE-754 standard is shown below:

- The sign of the real number is stored in the MSb, followed by 8 bit exponent and then a 23 bit mantissa
- The mantissa presents the precision/accuracy. To increase precision, IEEE 754 Standard uses a normalized mantissa which implies that its most significant bit is always 1. So the mantissa actually has 24 bit precision, but only 23 bits need to be stored
- The exponent bits represent the range. To store the exponent and its sign, one can use 2's complement encoding. But the designers of IEEE-754 used some thing known a biased notation. WHY?

$100 Question: Why exponent is saved in bias Notation?

0000	0	0	-5	-10	-7
0001	1	1	-4	-9	-6
0010	2	2	-3	-8	-5
0011	3	3	-2	-7	-4
0100	4	4	-1	-6	-3
0101	5	5	0	-5	-2
0110	6	6	1	-4	-1
0111	7	7	2	-3	0
1000	8	-8	3	-2	1
1001	9	-7	4	-1	2
1010	10	-6	5	0	3
1011	11	-5	6	1	4
1100	12	-4	7	2	5
1101	13	-3	8	3	6
1110	14	-2	9	4	7
1111	15	-1	10	5	8

S	Exponent & its sign (8)	Mantissa (23)

Why does IEEE-754 designers used biased exponent representation?

$$2^{(n-1)} - 1$$

$$2^{(4-1)} - 1 = 7$$

$$2^{(8-1)} - 1 = 127$$

$100 Question: Why exponent bits are before Mantissa?

Why does IEEE 754 standard use biased exponent representation and place it before the mantissa?

True/Real Exponent		Biased Exponent
-127		0
-126		1
-125		2
-124	+127 →	3
…		…
…		…
-1		126
0		127
1		128
…		…
…	-127	…
125	←	252
126		253
127		254
128		255

- The first advantage of use of biased exponent is that, it allows to have different numbers of positive and negative exponents (as discussed on previous slide)
- Keeping this biased exponent bits before the mantissa bits allows two floating point numbers to be compared easily. As you can see in the table the biased exponents are in lexical order. This would not have been possible if exponent is saved in 2's complement. So, this way sorting real numbers may also be carried out by IU as biased exponent are the most significant bits

Conversion Examples

Example 1

Convert **0.09375**$_{10}$ to 32 bit IEEE-754 floating point representation in Hex format

S	Exponent & its sign (8)	Mantissa (23)

1. **Determine the sign bit:** Positive, so 0
2. **Convert to pure binary:** 0.00011_2
3. **Normalize (to get mantissa and true exponent):** $0.00011 = 1.100 \times 2^{-4}$
4. **Determine biased exponent:** $-4 + 127 = 123_{10} = 01111011_2$
5. **Remove leading 1 from mantissa:** $1.100 = 10000000000000000000000$
   (Implied 1 on the left of the radix point is not stored)
6. **Assemble the result:**

0	01111011	10000000000000000000000

7. **Write Result in Hex:** $0011\ 1101\ 1100\ 0000\ 0000\ 0000\ 00000000_2$ **3DC00000**$_{16}$

Calculation	Bit
0.09375 x 2 = 0.1875	0
0.1875 x 2 = 0.375	0
0.375 x 2 = 0.75	0
0.75 x 2 = 1.5	1
0.5 x 2 = 1.0	1

Example 2

Convert **-123.3**$_{10}$ to 32-bit IEEE-754 floating point representation in Hex format

S	Exponent & its sign (8)	Mantissa (23)

1. **Determine the sign bit:** Negative, so 1
2. **Convert to pure binary:** $1111011.0100110011\ldots_2$
3. **Normalize (to get mantissa and true exponent):** $= 1.1110110100110011\ldots \times 2^{+6}$
4. **Determine biased exponent:** $+6 + 127 = 133_{10} = 10000101_2$
5. **Remove leading 1 from mantissa:** $= .11101101001100110011001\ldots$
   (Implied 1 on the left of the radix point is not stored)

✓ **Round mantissa up or down if necessary:** $= .11101101001100110011010$

6. **Assemble the result:**

1	10000101	11101101001100110011010

7. **Write Result in Hex:** $1100\ 0010\ 1111\ 0110\ 1001\ 1001\ 1001\ 1010_2$ **C2F6999A**$_{16}$

$0.3 \times 2 = 0.6$ 0
$0.6 \times 2 = 1.2$ 1
$0.2 \times 2 = 0.4$ 0
$0.4 \times 2 = 0.8$ 0
$0.8 \times 2 = 1.6$ 1
$0.6 \times 2 = 1.2$ 1
$0.2 \times 2 = 0.4$ 0
$0.4 \times 2 = 1.8$ 0
….

Example 3

Given the 32-bit IEEE-754 floating point representation $\mathbf{426A0000_{16}}$ find the decimal value

1. **Convert to pure binary:** $0100\ 0010\ 0110\ 1010\ 0000\ 0000\ 0000\ 0000_2$
2. **Split to components:**

3. **Determine sign:** 0, so positive
4. **Determine biased exponent:** $10000100_2 = 132_{10}$
5. **Get true exponent:** $132 – 127 = +5$
6. **Write the result:** $+1.11010100000000000000000 \times 2^{+5} = +111010.1000000 = \mathbf{+58.5}$

(Don't forget to write the implicit one before the significand bits)

Example 4

Given the 32-bit IEEE-754 floating point representation $\mathbf{41440000}_{16}$ find the decimal value

1. **Convert to pure binary:** $0100\ 0001\ 0100\ 0100\ 0000\ 0000\ 0000\ 0000_2$

2. **Split to components:**

3. **Determine sign:** 0, so positive

4. **Determine biased exponent:** $10000010_2 = 130_{10}$

5. **Get true exponent:** $130 - 127 = +3$

6. **Write the result:** $+1.10001000000000000000000 \times 2^3 = +1100.01000 = \mathbf{+12.25}$

(Don’t forget to write the implicit one before the significand bits)

Range/Precision of IEEE FP Representation

S	Exponent & its sign (8)	Mantissa (23)

Range (32 bit):

0/1	11111110	11111111111111111111111

- Exponent: 254 - 127 = +127 An exponent of all 1s is reserve (more on it later)
- Significand: $1.111\ldots1 = 1 + (1 - 2^{-23}) = 2 - 2^{-23}$
- Range: $\pm(2 - 2^{-23}) \times 2^{+127} = \pm 3.403 \times 10^{+38}$

Precision (32 bit):

0/1	00000001	00000000000000000000000

- Exponent: 1 - 127 = -126 An exponent of all 0s is reserve (more on it later)
- Significand: $1.000\ldots0 = 1$
- Precision: $\pm 1 \times 2^{-126} = \pm 1.1755 \times 10^{-38}$

Storage Layout for IEEE FP Representation

- Single Precision (32 bits):

Exponent bias = $2^{(8-1)} - 1 = 127$ Precision = 23 x $\log_{10} 2$ = 6 decimal digits

- Double Precision (64 bits):

Exponent bias = $2^{(11-1)} - 1 = 1023$ Precision = 52 x $\log_{10} 2$ = 15 decimal digits

- Quadruple Precision (128 bits):

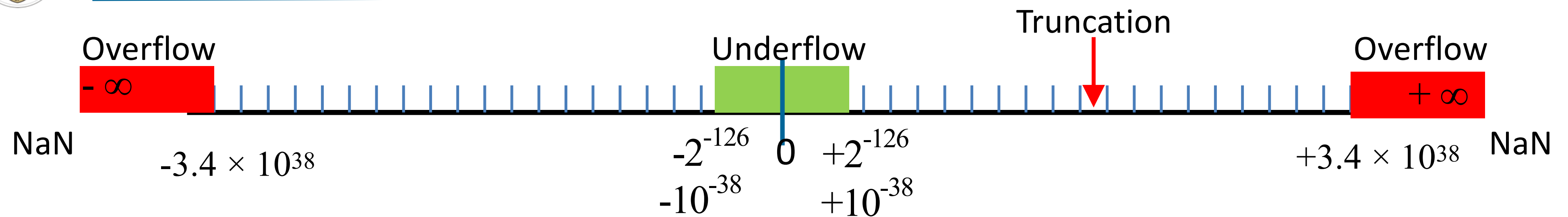
Exponent bias = $2^{(15-1)} - 1 = 16383$ Precision = 112 x $\log_{10} 2$ = 33decimal digits

- Octuple Precision (256 bits):

Exponent bias = $2^{(19-1)} - 1 = 262143$ Precision = 236 x $\log_{10} 2$ = 71 decimal digits

Floating Point Arithmetic

Floating Point Computation



- **Overflow:** An overflow occurs when a mathematical operation results in a value that falls outside the range of a data type. In case of 32 bit FP operations any result that falls outside +/-3.403 x 10^{+38} will be rounded to infinity. (e.g., `a=3.0 x 10`^{30} then `a*a` will overflow)
- **Underflow:** An underflow occurs when an operation results in a value that is smaller that the smallest number that can be stored as a floating point number. (Remember there is no underflow in integer arithmetic). In case of 32 bit FP operations any result that is less than +/- 1.1755 x 10^{-38} is an underflow. (e.g., `a=3.0 x 10`$^{-30}$ then `a*a` will underflow)
- So good programs must detect round-off errors as well as overflow/ underflow and raise alerts whenever they occur

IEEE 754: Special Values

- **Infinity:** All 1s in the exponent field and all 0s in the mantissa field represent infinity in the IEEE standard. The sign bit distinguishes between negative infinity and positive infinity. Operations with infinite values are well defined in IEEE floating point
- **Zero:** All 0s in the exponent field and all 0s in the mantissa field represent zero in the IEEE standard. Even though +0 and –0 have distinct representations, though they both compare as equal

Exponent Value	Mantissa	Represents
11111111	00000000	Infinity
00000000	00000000	Zero
11111111	Not all zeros	Not a Number (NaN)
00000000	Not all zeros	Subnormal (very small)

IEEE 754: Special Values

- **Not a Number (NaN):** When an operation is performed by a computer on a pair of operands which results in an indeterminate answer, it is called NaN in IEEE standard. All 1s in the exponent field and not all 0s in the mantissa field represent NaN
- **Subnormal Number:** If the exponent bits are all zeros, and the mantissa are not all zeros, the value being represented is very very small. This is known as a sub-normal or de-normalized number

Exponent Value	Mantissa	Represents
11111111	00000000	Infinity
00000000	00000000	Zero
11111111	Not all zeros	Not a Number (NaN)
00000000	Not all zeros	Subnormal (very small)

Things To Do

- Represent following decimal numbers using 32-bit IEEE-754 floating point representation:
  - 19.59375
  - 127.69
  - -68.34
- Given the 32-bit IEEE-754 floating point numbers, find their corresponding decimal values:
  - 0x4355AE14
  - 0xC18A0000
- Confirm your working by using online IEEE-754 converter:

https://www.binaryconvert.com/index.html

Coming to office hours does NOT mean you are academically week!