CS 250: Foundation of Computer Systems
[Binary Representations & Operations]

A number in context
Look closely …
a 1 or a 0 in the right place
The leftmost bit to be precise
Is also a sign

Break a sequence of bits here or there
And it gives you powers
To size up who-ville or
the universe's atoms

The only clarity that matters? Context
The interpretation that follows? Unambiguous

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- N! = 1 x 2 x 3 x … x N
- Did long variables exist on 32-bit systems?
- Twos complement: Conversion to decimal?

2

Topics covered in this lecture

- Signed numbers
  - Two's complement
  - One's complement
- Floating point numbers
- Hexadecimal numbers

Announcements

- Recitations are moving to CSB-315 [Unix Lab]
- Quiz etiquette

Summarizing two's complement for an n-bit binary system

- The system codes 2^n signed numbers, ranging from -2^{n-1} to $(2^{n-1} - 1)$
- The code of any *nonnegative* number begins with a **0**
- The code of any *negative* number begins with a **1**
- Representation of $-x$ ➔ $2^n - x$
- To obtain the binary code of $-x$ from the binary code of x?
  - **Flip all the bits of x and add 1 to the result**

5

Let's consider an 8-bit binary system and two's complement

- The system codes 2^n signed numbers, ranging from -2^{n-1} to $(2^{n-1} - 1)$
  - i.e., from -2^{8-1} to $(2^{8-1} - 1)$ or -2^7 to (2^7-1)
- Let's look at the number -42
  - Method-1: -42 ➔ $2^8 - 42 = 214 =$ 0b11010110
  - Method-2: Flip all bits of x and add 1 to the result
    - 42 =0b00101010
    - Flips the bits: 0b11010101
    - Add 1: 0b11010110

6

Converting 2s complement to decimal (or denary)

- 0b11010110
  - The weight for the leftmost digit is negative
  - $1 \times -2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
  - **– 128** + 64 + 0 + 16 + 0 + 4 + 2 + 0
  - **– 128** + 86
  - $-\mathbf{42}_{10}$

ONE'S COMPLEMENT

Another useful concept to know … one's complement

- A not so **successful** attempt to represent signed numbers
- Get to negative numbers by taking positive numbers and flipping all the bits (i.e., 1 becomes 0 and 0 becomes 1)

One's complement

- Flipping each bit of 0111 (+7) yields 1000 (–7)
- Has **two** different representations for zero

Sign	2^2	2^1	2^0	Decimal
0	1	1	1	+7
0	1	1	0	+6
0	1	0	1	+5
0	1	0	0	+4
0	0	1	1	+3
0	0	1	0	+2
0	0	0	1	+1
0	0	0	0	+0
1	1	1	0	–1
1	1	0	1	–2
1	1	0	0	–3
1	0	1	1	–4
1	0	1	0	–5
1	0	0	1	–6
1	0	0	0	–7

One's complement: Addition is a little more complicated

1

0	0	1	0	+2
1	1	1	0	–1
0	0	0	0	0
				1
				1

End-around carry

11

12

General-purpose computers are built to solve general-purpose problems

- Which involve a wide range of numbers
- You can get an idea of this range by skimming a physics textbook
  - There are **tiny numbers** such as Planck's constant (6.63×10^{-34} joule-seconds) and
  - **Huge numbers** such as Avogadro's constant (6.02×10^{23} molecules/mole)
  - This is a range of 10^{57}, which comes out to about 2^{191}
  - That's almost 200 bits!
- Bits just aren't cheap enough to use a few hundred of them to represent every number, so we need a different approach

Look to what we have done in the past

- **Scientific notation** represents a large range of numbers by (how else?) creating a *new context* for interpretation
  - It uses a number with a **single digit** to the left of the decimal point, called the **mantissa**, multiplied by 10 raised to some power, called the **exponent**

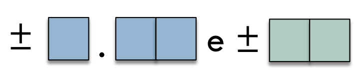
$$\underbrace{6.63}_{\text{Mantissa}} \times \underbrace{10^{-34}}_{\text{Exponent}}$$

Floating points

- Computers use the same system, except that the mantissa and exponent are binary numbers and 2 is used instead of 10
- This is called the **floating point representation**

15

More on the scientific notation: Let's come up with a representation



- A value like 9,876,543,210 would be approximated with 9.88×10^9 (or 9.88e+9 in programming language notation)

16

Scientific notation complicates arithmetic somewhat

- When adding and subtracting two numbers in scientific notation, you must **adjust** the two values so that their exponents are the same
- For example, when adding 1.23e1 and 4.56e0, you could convert 4.56e0 to 0.456e1 and then add them
  - The result (1.23e1 + 0.456e1 = 1.686e1), does not fit into the three significant digits of our current format (in the previous slide)
  - So, we must either **round** or **truncate** the result to three significant digits
    - Rounding generally produces the more accurate result, so let's round to obtain 1.69e1

The implications of such adjustments?

- The lack of **precision** (the number of digits or bits maintained in a computation) affects the **accuracy** (the correctness of the computation)

Floating point representation: **Binary version** of the scientific notation to represent a wide range of numbers

- The naming convention is confusing because the binary (or decimal) point is always in the same place
  - Between the ones and halves (tenths in decimal)
  - The "float" is just another way of saying "scientific notation," which allows us to write 1.2×10^{-3} instead of 0.0012
- By **separating** the significant digits from the exponents
  - The floating-point system allows us to represent very small or very large numbers without having to store all those zeros

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA | COMPUTER SCIENCE DEPARTMENT | BINARY REPRESENTATIONS | L4.19

19

IEEE 754 Floating standard [1/2]

- When Intel planned to introduce a floating-point unit (FPU) for its original 8086 microprocessor?
  - Intel knew their electrical engineers and solid-state physicists didn't have the numerical analysis background to design a good floating-point representation
  - Went out and hired the best numerical analyst they could find to design a floating-point format for its 8087 FPU
- That person then hired two other experts in the field, and the three of them (Kahan, Coonen, and Stone) designed the KCS Floating-Point Standard
  - They did such a good job that the IEEE organization used this format as the basis for the **IEEE Std 754** floating-point format

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA | COMPUTER SCIENCE DEPARTMENT | BINARY REPRESENTATIONS | L4.20

20

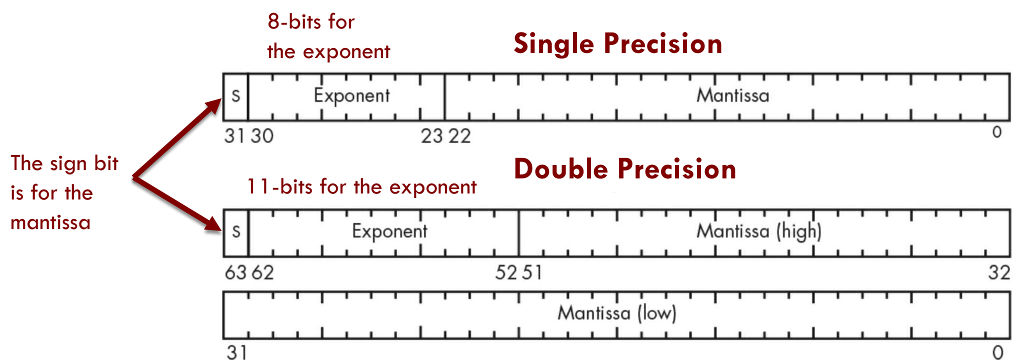
IEEE 754 Floating standard [2/2]

- The floating-point system is the standard way to represent real numbers in computing
- There are two signs:
  - One for the **mantissa** and one for the **exponent** (*hidden*)
- There are also a lot of tricks to make sure that things like rounding work as well as possible and to minimize the number of wasted bit combinations

The mantissa and the exponent

- The mantissa is a base value that usually falls within a **limited range** (for example, between 0 and 1)
- The exponent is a **multiplier** that, when applied to the mantissa, produces values outside this range
- The big advantage of the mantissa/exponent configuration
  - Floating-point format can represent values across a **wide range**

IEEE Floating point number formats



A couple of the tricks that are used in the standard

- **Normalization**, which adjusts the mantissa so that there are no leading (that is, on the left) zeros
- A second trick, from Digital Equipment Corporation (DEC), doubles the accuracy
  - By throwing away the leftmost bit of the mantissa since we know that it will always be 1, which makes room for one more bit
- Exponent values of all 0s and all 1s would have special meaning

Two types of floating point numbers: Single and double-precision

- **Single-precision** numbers use 32 bits and can represent numbers approximately in the range $\pm 10^{\pm 38}$ with about 7 digits of accuracy
  - Although there is an infinite number of values between 1 and 2, we can represent only 8 million (2^{23}) of them because we use a 23-bit mantissa
    - Therefore, have only 23 bits of precision
- **Double-precision** numbers use 64 bits and can represent a wider range of numbers, approximately $\pm 10^{\pm 308}$, with about 15 digits of accuracy
  - Number of atoms in the known universe is between $10^{78} \sim 10^{82}$

IEEE 754 also uses some special bit patterns

- To represent things like division by zero, which evaluates to positive or negative infinity
- It also specifies a special value called NaN, which stands for "not a number"
  - If you find yourself in the NaNny state, it probably means that you did some illegal arithmetic operation

Let's look at a number with a decimal point

- 0.15625
- We will compute its binary representation

	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
Place-value	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125

Unlike the division (by 2) that we did for number left of the decimal point ….

- During conversions, for numbers to the right of the decimal, we will **multiply** 2
- 0.15625_{10}

	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
Place-value	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125
						0	0	1	0	1

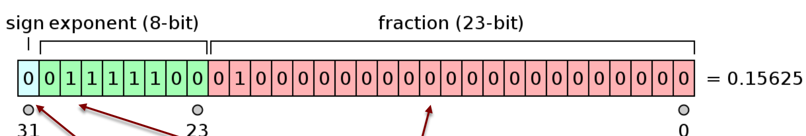
0.15625 x 2 = **0**.3125 → 0
0.3125 x 2 = **0**.625 → 0
0.625 x 2 = **1**.250 → 1
0.250 x 2 = **0**.500 → 0
0.500 x 2 = **1**.000 → 1

Top to bottom is left-to-right

$0.15625_{10} = 0.00101_2$

- $0.00101_2 = 1.01 \times 2^{-3}$
- Fraction is .01 and the exponent is -3
- The number is positive
- In the IEEE-754 standard, the exponent is written as a **biased exponent**
  - In single-precision, you need to add 127
    - -3 would be written as $-3 + 127 = 124_{10} = 01111100_2$
  - In double-precision you need to add 1023

Representation of $0.15625_{10} = 0.00101_2$ in IEEE-754 single precision



- $0.00101_2 = 1.01 \times 2^{-3}$
- The number is positive
- Exponent is written as a **biased exponent**: 01111100_2
- We only write the **fractional part** (the 1in 1.01 is implied) i.e., 01 and then pad zeros all the way to the right

NaN and Infinity

- The biased-exponent field is filled with all 1 bits to indicate either
  - Infinity mantissa field = 0
  - NaN mantissa field ≠ 0

31

HEXADECIMAL

32

Hexadecimal

- Hexadecimal is **base 16**!
- Given what we've already seen so far, you probably know what that means
- Hexadecimal, or just **hex** for short, is a place-value system where
  - Each place represents a power of 16
  - and each place can be one of 16 symbols

Hexadecimal number representation

- As in all place-value systems, the rightmost place will still be the ones place
- The next place to the left will be the sixteens place, then the 256s (16 × 16) place, then the 4,096s (16 × 16 × 16) place, and so on
- Simple enough!

But what about the other requirement that each place can be one of 16 symbols? [1/2]

- We usually have ten symbols to use to represent numbers, 0 through 9
- We need to add **six more** symbols to represent the other values

But what about the other requirement that each place can be one of 16 symbols? [2/2]

- We could pick some random symbols like & @ #, but these symbols have no obvious order
- Instead, the standard is to use A, B, C, D, E, and F
  - Either uppercase or lowercase is fine!
- In this scheme, A represents ten, B represents eleven, and so on, up to F, which represents fifteen

Symbols in the hexadecimal or hex number system

- **A** represents ten, **B** represents eleven, and so on, up to **F**, which represents fifteen …

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

Consider the number **0x**1A5 in hexadecimal

- What's the value of this number in decimal?
- The rightmost place is worth 5
- The next place has a weight of 16, and there's an A there, which is 10 in decimal, so the middle place is worth 16 × 10 = 160
- The leftmost place has a weight of 256, and there's a 1 in that place, so that place is worth 256
- The total value then is 256 + 160 + 5 = 421 in decimal

Another view of the number 0x1A5 in hexadecimal

1	A	5
256s place	16s place	1s place
16^2	16^1	16^0
16 x 16 = 256	16	1

= 1*256 + 10 * 16 + 1*5
= 256 + 160 + 5
= 421 in decimal

Conversion from decimal to hex

- The hex number is constructed from **right to left**
- Step 1: Divide the decimal number by 16 and note down the remainder
- Step 2: Divide the obtained quotient by 16, and note remainder again
- Step 3: Repeat the above steps until you get **0** as the quotient
  - **Stopping criteria**

Example: Decimal number 421

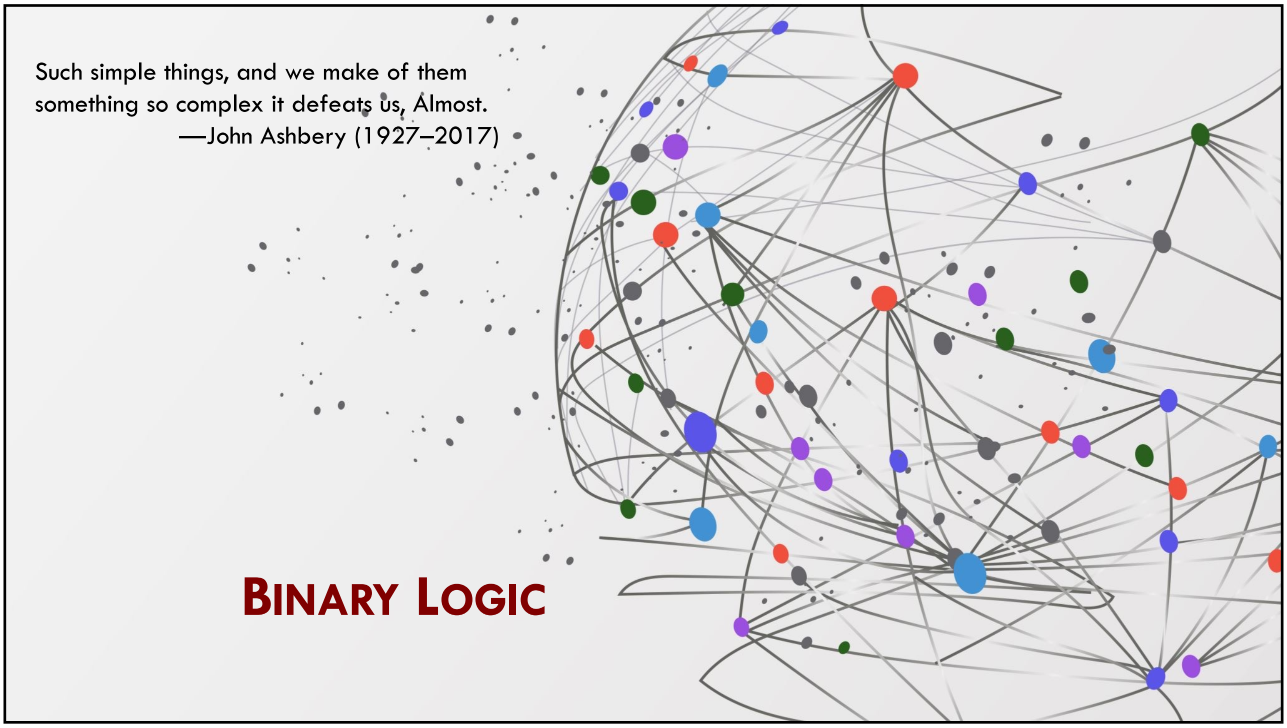
- 421 ÷ 16 = **26** (Remainder 5)
- **26** ÷ 16 = **1** (Remainder 10 or **A**)
- **1** ÷ 16 = **0** (Remainder 1)

Top to Bottom
is
Right to Left
{LSB to MSB}

- Hex representation: 0x1A5
  - = 1*256 + 10 * 16 + 1*5
  - = 256 + 160 + 5
  - = 421 in decimal

Some conversions across number systems

	Example 1	Example 2
Binary	1111 0000 0000 1111	1000 1000 1000 0001
Hexadecimal	F00F	8881
Decimal	61,455	34,945



43

Why binary logic matters

- Every digital device — be it a PC, a cell phone, or a network router — is based on chips designed to store and process binary information
- Although these chips come in different shapes and forms, they are all made of the same building blocks: **elementary logic gates**
- The gates can be physically realized using many different hardware technologies
  - But their logical behavior, or abstraction, is consistent across all implementations

44

We've looked at using binary to represent data, but computers do more than simply store data

- Binary allows us to work with data as well
- Computers give us the capability to process data using hardware that we can program to execute a sequence of simple instructions
  - Instructions like "add two numbers together" or "check if two values are equal"
- Computer processors that **implement these instructions** are fundamentally based on **binary logic**
  - A system for describing logical statements where variables can only be one of two values — `true` or `false`

Why is binary a natural fit for logic?

- Typically, when someone speaks of logic, they mean **reasoning**, or thinking through *what is known* in order to **arrive at a valid conclusion**
- When presented with a set of facts, logic allows us to determine whether *another related statement* is also **factual**
- Logic is all about **truth** — what is true, and what is false
- Likewise, a bit can only be one of two values, 1 or 0
  - Therefore, a single bit can be used to represent a logical state of true (1) or false (0)

Let's consider the logical statements for a rectangle

- If the shape does not have four sides **and** does not have four right angles
  - It is not a rectangle
- If the shape does not have four sides **and** does have four right angles
  - It is not a rectangle
- If the shape does have four sides **and** does not have four right angles
  - It is not a rectangle
- If the shape does have four sides **and** does have four right angles
  - It is a rectangle!

Let's put those statements in table

Four sides	Four right angles	Is a rectangle
False	False	False
False	True	False
True	False	False
True	True	True

What's so special about this table?

- This type of table is known as a **truth table**
- A truth table shows **all the possible combinations** of
  - **Conditions** (inputs) and
  - Their **logical conclusions** (outputs)
- Our previous table was written specifically for our statement about a rectangle, but …
  - The same table **applies to any logical statement joined with AND**

Let's represent false as 0 and true as a 1

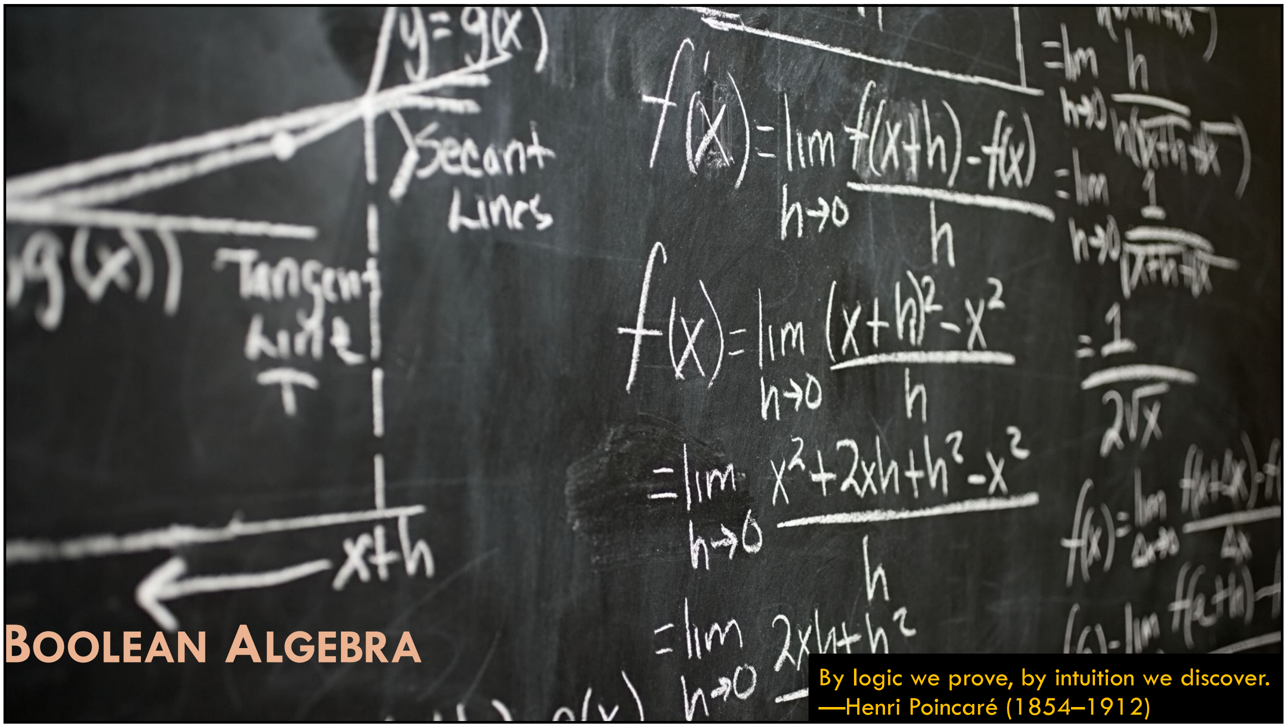
x	y	x And y
0	0	0
0	1	0
1	0	0
1	1	1

Let's look at another example: **OR**

- Say you work at a shop that gives a discount to only two types of customers:
  - Children
  - People wearing sunglasses
- No one else is eligible for a discount
- If you wanted to state the store's policy as a logical expression, you could say the following:
  - GIVEN the customer is a child
  - OR GIVEN the customer is wearing sunglasses
  - I CONCLUDE that the customer is eligible for a discount

The OR Truth Table

x	*y*	*x* Or *y*
0	0	0
0	1	1
1	0	1
1	1	1



BOOLEAN ALGEBRA

By logic we prove, by intuition we discover.
—Henri Poincaré (1854–1912)

53

Logic Operations

- One use of bits is to represent the answers to yes/no questions such as "Is it cold?" or "Do you like my hat?"
  - We use the terms `true` for yes and `false` for no
- Questions like "Where's the party?" don't have a yes/no answer and can't be represented by a single bit
- We often combine several yes/no clauses into a single sentence
  - We might say, "Wear a coat if it is cold or if it is raining" or "Go skiing if it is snowing and it's not a school day"

54

Logic Operations

- Another way of saying those things might be
  - "Wear coat is `true` if cold is `true` **or** raining is `true`" and "Skiing is `true` if snowing is `true` **and** school day is not `true`"
  - These are logic operations that each produce a new bit based on the contents of other bits

Boolean Functions

- A **boolean function** is a function that operates on binary inputs and returns binary outputs
- Since computer hardware is based on representing and manipulating binary values …
  - **Boolean functions play a central role** in the specification, analysis, and optimization of hardware architectures

Every Boolean function can be defined using **two** alternative representations

- First, we can define the function using a **truth table**
  - For each one of the 2^n possible tuples of input variable values, the table lists the value of $f(v_1, v_2, \ldots, v_n)$
  - Can be thought of as a data-driven definition
- In addition to this data-driven definition, we can also define Boolean functions using **Boolean expressions**
  - For example: (x Or y) And Not (z)

Boolean Algebra [1/2]

- Algebra is a set of rules for operating on numbers
- **Boolean algebra** manipulates **two-state binary values** that are typically labeled true/false, 1/0, yes/no, on/off, and so forth
- We will use 1 and 0

Boolean Algebra [2/2]

- **Boolean algebra**, invented in the 1800s by English mathematician George Boole, is a set of rules that we use to operate on bits
- As with regular algebra: the associative, commutative, and distributive rules also apply
  - x * y = y * x — Commutative
  - (x * y) * z = x * (y * z) — Associative
  - x * (y + z) = x*y + x*z — Distributive

COLORADO STATE UNIVERSITY — Professor: SHRIDEEP PALLICKARA, COMPUTER SCIENCE DEPARTMENT — BINARY REPRESENTATIONS — L4.59

59

The contents of this slide-set are based on the following references

- Noam Nisan and Shimon Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles.* 2nd Edition. ISBN-10/ ISBN-13: 0262539802 / 978-0262539807. MIT Press. [Preface, Chapter 1-2]
- Jonathan E. Steinhart. *The Secret Life of Programs: Understand Computers -- Craft Better Code.* ISBN-10/ ISBN-13 : 1593279701/ 978-1593279707. No Starch Press. [Chapter 1]
- Randall Hyde. Write Great Code, Volume 1, 2nd Edition: Understanding the Machine 2nd Edition. ASIN: B07VSC1K8Z. No Starch Press. 2020. [Chapter 2]
- Matthew Justice. *How Computers Really Work: A Hands-On Guide to the Inner Workings of the Machine.* ISBN-10/ISBN-13 : 1718500661/ 978-1718500662. No Starch Press. 2020. [Chapter 1]
- https://en.wikipedia.org/wiki/IEEE_754

COLORADO STATE UNIVERSITY — Professor: SHRIDEEP PALLICKARA, COMPUTER SCIENCE DEPARTMENT — BINARY REPRESENTATIONS — L4.60

60