CS250: Foundations of Computer Systems
[Boolean Logic & Algebra]

The Janus-faced Boolean Function
One side a truth table
The other an expression

Like snowflakes
Every truth table is unique
Expressions? Dime a dozen

Fueled by Boolean algebra
Many an expression
Gets you to your truth table destination

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- How do we know where the fractional places begin?
- Why should we care about the IEEE 754 floating point standard?
- Why do we multiply by 2 for the decimal side during conversions of numbers like 0.350 etc
- How does the circuitry know its deal with a floating point number vs a vanilla whole number?
- Why use hexadecimal?
- How should I study?

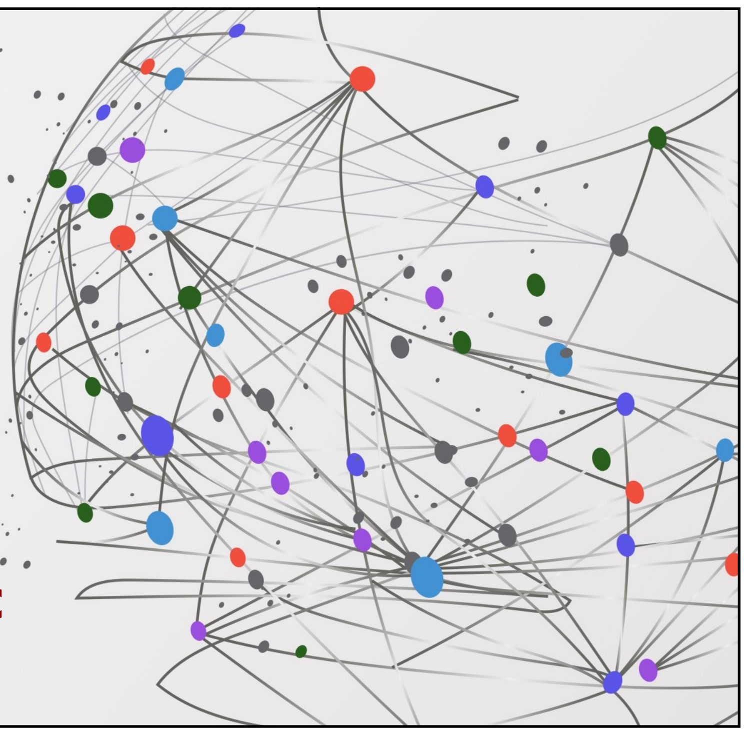
2

Topics covered in this lecture

- Boolean Algebra
  - Not, Or, and And
  - Xor
  - Nand

3

Such simple things, and we make of them
something so complex it defeats us, Almost.
—John Ashbery (1927–2017)



BINARY LOGIC

4

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 an 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

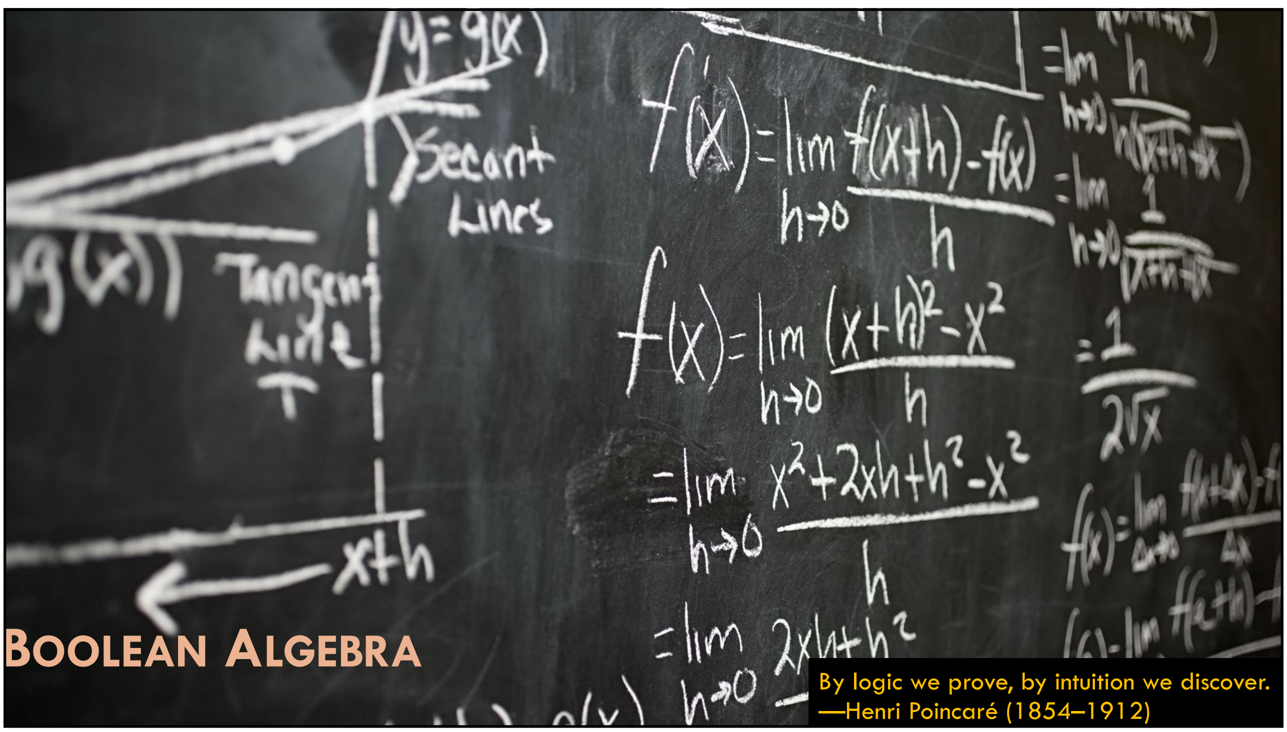
Let's look at another example: **OR**

- 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

11



12

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"

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

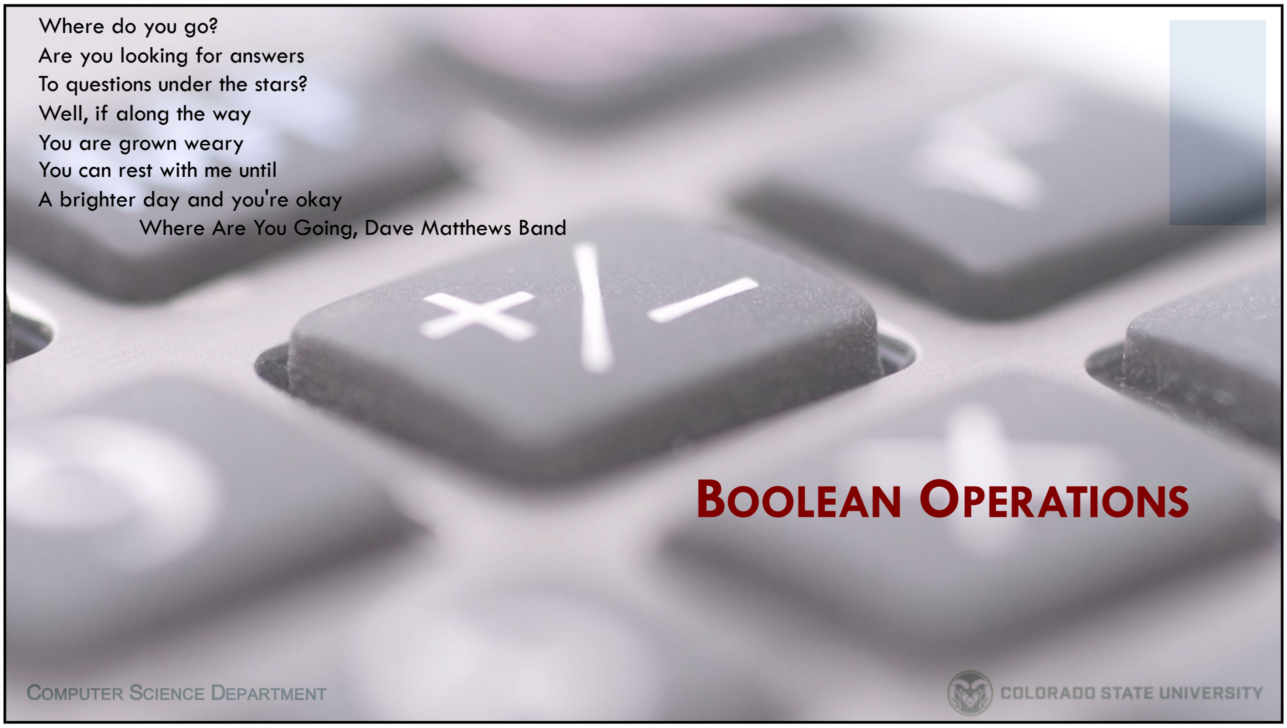
- First, we can define the function using a **truth table**
  - For each one of the 2^n possible tuples of input variable values, the table lists the value of $f(v_1, v_2, \ldots, v_n)$
  - Can be thought of as a data-driven definition
- In addition to this data-driven definition, we can also define 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



19

Boolean Operations

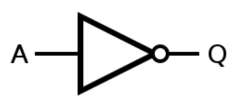
- There are three basic Boolean operations
  - **Not**, **And**, and **Or**
  - Composite operations: **Xor** (short for "exclusive-or"), **Nand**, and **Nor**

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT BOOLEAN LOGIC & ALGEBRA L5.20

20

NOT: This operation means “the **opposite**”

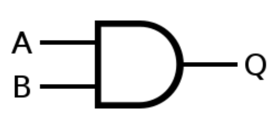
- For example, if a bit is false, NOT that bit would be true
  - If a bit is true, NOT that bit would be false



x	Not x
0	1
1	0

AND: This operation involves 2 or more bits

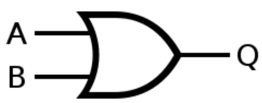
- In a 2-bit operation, the result is true only if **both** the first AND second bit are true
- When more than 2 bits are involved, the result is true only if **all** bits are true



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

OR: This operation also involves 2 or more bits

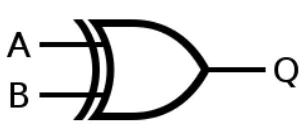
- In a 2-bit operation, the result is true if the first OR second bit is true; otherwise, the result is false
- With more than 2 bits, the result is true if **any** bit is true



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

Xor: Also known as Exclusive-OR

- The result of an exclusive-or operation is true if the first and second bits have **different values**
  - It's either but not both
- Because "exclusive-or" is a mouthful, we often use the abbreviation Xor (pronounced "ex-or")



x	*y*	*x* **Xor** *y*
0	0	0
0	1	1
1	0	1
1	1	0

Nand: This operation involves 2 or more bits

- The name of the Nand operator is shorthand for Not-And, coming from the observation that Nand (x, y) is equivalent to Not (And (x, y))
  - Pipes the output of the And gate through a Not gate

x	*y*	*x* And *y*	*x* Nand *y*
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

What makes And, Or, and Not more interesting, or privileged, than any other subset of Boolean operators?

- The short answer is that indeed there is nothing special about And, Or, and Not
- A deeper answer is that various subsets of logical operators can be used for expressing **any** Boolean function, and {And, Or, Not} is one such subset

What makes And, Or, and Not more interesting, or privileged, than any other subset of Boolean operators?

- If you find this claim impressive, consider this: any one of these three basic operators can be expressed using yet another operator—**Nand**
  - The name of the Nand operator is shorthand for Not-And, coming from the observation that Nand (x, y) is equivalent to Not (And (x, y))
- Now, that's impressive!
- It follows that **any boolean function can be realized using Nand gates only**

27

28

Representing a Boolean function using truth tables and Boolean expressions

x	y	z	f(x,y,z) = (x Or y) And Not(z)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Truth Tables and Boolean Expressions [1/2]

- Given a Boolean function of n variables represented by a Boolean expression, we can always construct from it the function's truth table
- We simply compute the function for every set of values (row) in the table
  - This construction is laborious, and obvious

Truth Tables and Boolean Expressions [2/2]

- At the same time, the dual construction is not obvious at all:
- Given a truth table representation of a Boolean function, can we always **synthesize** from it a Boolean expression for the underlying function?
  - The answer to this intriguing question is **Yes!**

When it comes to building computers

- The truth table representation, the Boolean expression, and the ability to construct one from the other are all highly relevant

Suppose that we are called to build some hardware for sequencing DNA data

- Our domain expert biologist wants to describe the sequencing logic using a truth table
- Our job is to **realize this logic in hardware**
- With the truth table data as a point of departure, we can synthesize from it a Boolean expression that represents the underlying function
  - After *simplifying the expression* using Boolean algebra, we can proceed to implement it using logic gates

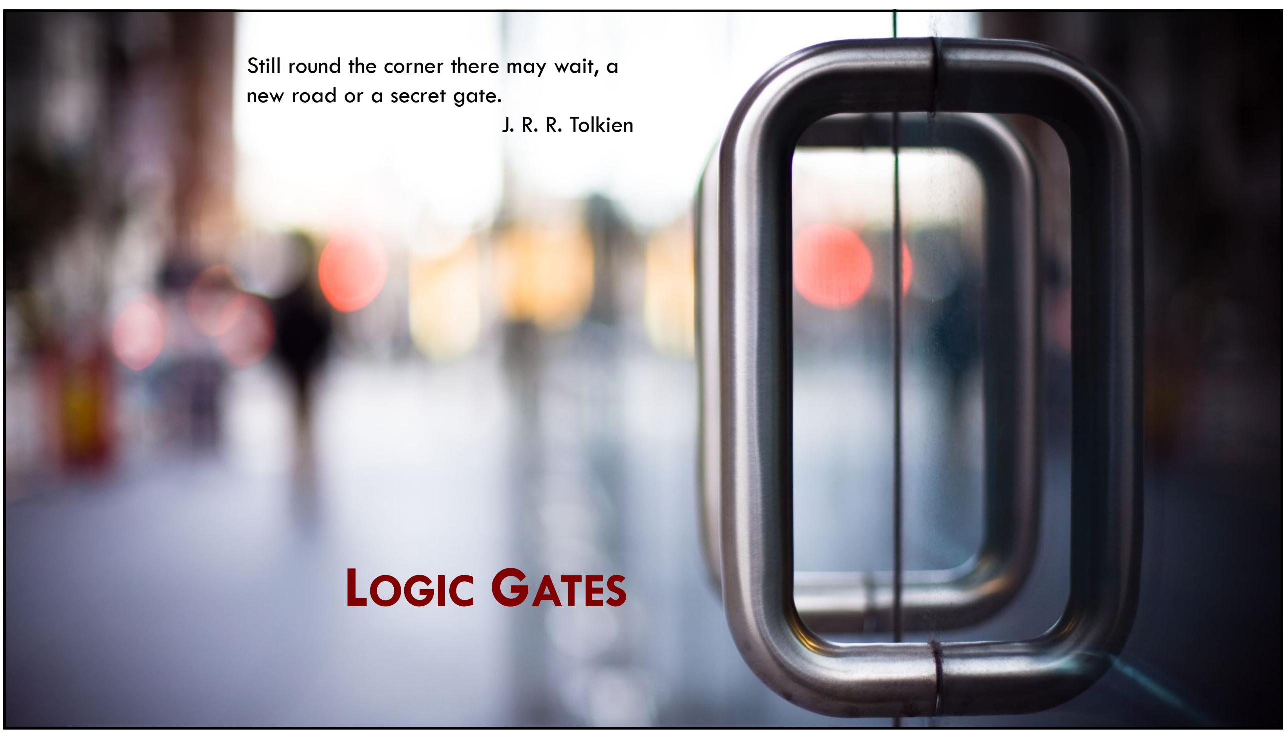
Truth table vs Boolean Expression

- A truth table is often a convenient means for describing some states of nature
- Whereas a Boolean expression is a **convenient formalism** for realizing this description in silicon
- The ability to move from one representation to the other is one of the most important practices of hardware design

Although the truth table representation of a Boolean function is unique

- Every Boolean function can be represented by **many** different yet equivalent Boolean expressions
  - And some will be shorter and easier to work with
- For example, the expression:
  - (Not (x And y) And (Not (x) Or y) And (Not (y) Or y))
  - Is equivalent to the expression Not (x)
- The ability to simplify a Boolean expression is the first step toward **hardware optimization**

35



36

Gates

- A **gate** is a physical device that implements a simple Boolean function
- Most digital computers today use electricity to realize gates and represent binary data
  - Today, gates are typically **implemented as transistors** etched in silicon, packaged as chips

Lots of "can do" implementations of gates also exist alongside practical ones

- Any alternative technology permitting switching and conducting capabilities can be employed
- Over the years, many hardware implementations of Boolean functions were created
  - Including magnetic, optical, biological, hydraulic, pneumatic, quantum-based, and even domino-based mechanisms
    - Many of these implementations are whimsical "can do" feats

Implication of switching technologies and Boolean algebra [1/2]

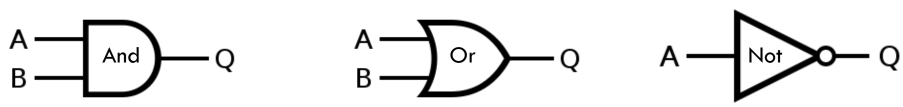
- The availability of alternative switching technologies, on the one hand, and the observation that Boolean algebra can be used to abstract the behavior of logic gates, on the other, is extremely important
- Implies that computer scientists don't have to worry about physical artifacts like electricity, circuits, switches, relays, and power sources

Implication of switching technologies and Boolean algebra [2/2]

- Allows computer scientists to be content with the **abstract** notions of Boolean algebra and gate logic
- Trusting blissfully that someone else—physicists and electrical engineers—will figure out how to actually realize them in hardware

Primitive Gates as black boxes

- Primitive gates can be viewed as **black box devices** that implement elementary logical operations

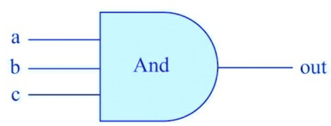


Composite gates

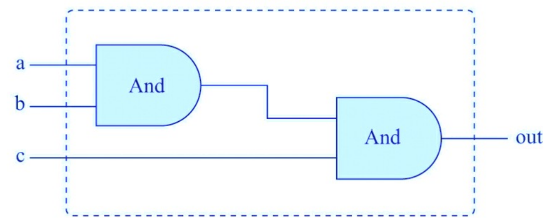
- Since all logic gates have the same input and output data types (0's and 1's), they can be combined, creating **composite gates** of arbitrary complexity
- For example, suppose we are asked to implement the three-way Boolean function And (a, b, c), which returns 1 when every one of its inputs is 1, and 0 otherwise
- Using Boolean algebra, we can begin by observing that a.b.c = (a.b).c

Next, we can use this result to construct the composite gate

Gate Interface



Gate Implementation

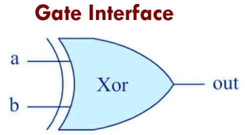



```
if (a==b==c==1) set out = 1
else set out = 0
```

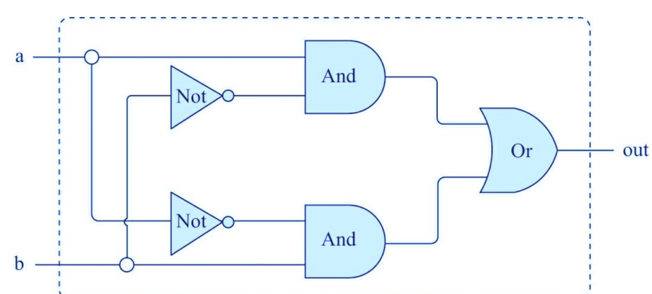

Let us consider another logic design example: Xor

- By definition, Xor (a, b) is 1 exactly when either a is 1 and b is 0 or a is 0 and b is 1
- Said otherwise, Xor (a,b) = Or (And (a, Not(b)), And (Not (a), b))

Gate Interface



a	*b*	*a* Xor *b*
0	0	0
0	1	1
1	0	1
1	1	0



Gate Implementation

Note that the **interface** of any given gate is **unique**: there is only one way to specify it

- This is normally done using a truth table, a Boolean expression, or a verbal specification
- This interface, however, can be **realized in many different ways**
  - Some will be more elegant and efficient than others
- For example, the Xor implementation we saw in the previous slide is one possibility
  - There are more efficient ways to realize Xor, using less logic gates and less inter-gate connections

Functionality vs Efficiency

- From a **functional** standpoint, the fundamental requirement of logic design is that the gate implementation will **realize its stated interface**
  - One way or another
- From an **efficiency** standpoint, the general rule is to try to use **as few gates as possible**, since fewer gates imply less cost, less energy, and faster computation

Art of Logic Design: Abstraction to Implementation

- Given a gate abstraction (also referred to as specification, or interface) …
- Find an efficient way to implement it using other gates that were already implemented

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. [Chapter 1-2, Appendix A]
- Jonathan E. Steinhart. *The Secret Life of Programs: Understand Computers -- Craft Better Code*. ISBN-10/ ISBN-13 : 1593279701/ 978-1593279707. No Starch Press. [Chapter 2]