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

Nand Atoms

Synthesize a Boolean function you say?
How?
Get to its disjunctive normal form
With And, Or, and Not navigating the storm

But that's not all
There's one gate to rule them all
Our atom, our one-man band
Nand

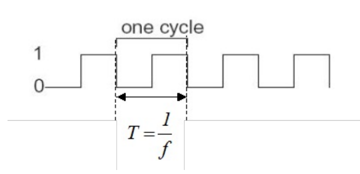
As you look on in awe
Look, there's De Morgan's Law

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

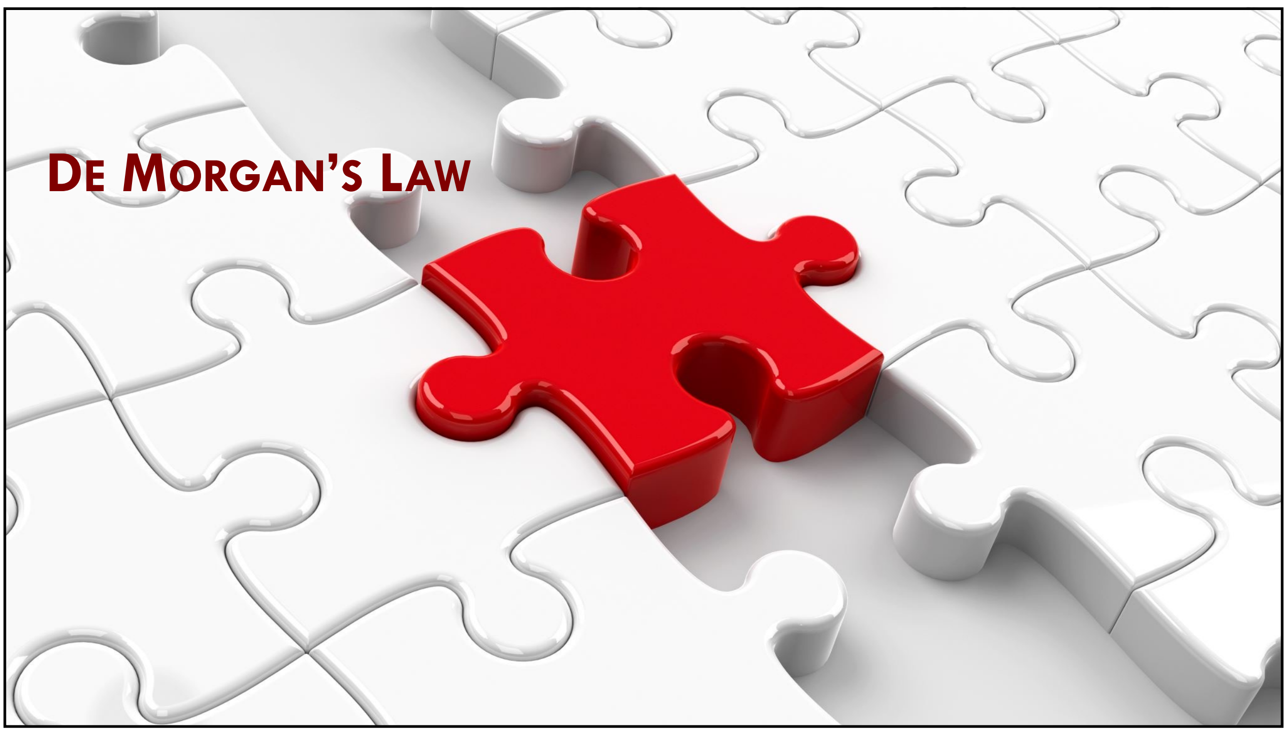
- Do manufacturers produce circuitry that have a mix of And, Not, Xor, etc. gates?
- How do you represent And, Or, and, Not with Nand?
- Why does the CPU care about these gates and boolean logic? Can't it just "do" the operations?
- Do truth tables have to be complete?
- What is a clock cycle?



2

Topics covered in this lecture

- De Morgan's Laws
- Synthesizing Boolean functions
- The expressive power of Nand gates
- Adder circuits



De Morgan's Law

- In the 1800s, British mathematician Augustus De Morgan added a law that applies only to Boolean algebra
  - The eponymous De Morgan's law
- This law states that the operation
  - Not (x And y) = Not(x) Or Not(y)
  - Not (x Or y) = Not(x) And Not(y)

Another way of stating this

- Not (A Or B) = Not(A) And Not(B) $\overline{A \cup B} = \overline{A} \cap \overline{B}$
- Not (A And B) = Not(A) Or Not(B) $\overline{A \cap B} = \overline{A} \cup \overline{B}$

Not (*x* And *y*) = Not(*x*) Or Not(*y*)

- Replacing And operations with Or
- Also: x And y = Not (Not(x) Or Not(y))

x	y	x And y	Not(x And y)
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

x	y	Not x	Not y	Not(x) Or Not (y)
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

7

Not (*x* Or *y*) = Not(*x*) And Not(*y*)

- Replacing Or operations with And
- x Or y = Not (Not(x) And Not(y))

x	y	x Or y	Not(x Or y)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

x	y	Not x	Not y	Not(x) And Not(y)
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

8

De Morgan's Law: Implications

- This means that with enough NOT operations, we can **replace** AND operations with OR operations (and vice versa)
- This is useful because computers operate on real-world input that's not under their control
- De Morgan's law is a tool that lets us operate on these negative logic propositions in addition to the positive logic that we've already seen
  - Similar to double negatives in languages such as English ("We didn't not go skiing")

While it would be nice if inputs were of the form cold or raining, they're often NOT cold or NOT raining

cold	raining	wear-coat
F	F	F
F	T	T
T	F	T
T	T	T

not-cold	not-raining	not-wear-coat
F	F	F
F	T	F
T	F	F
T	T	T

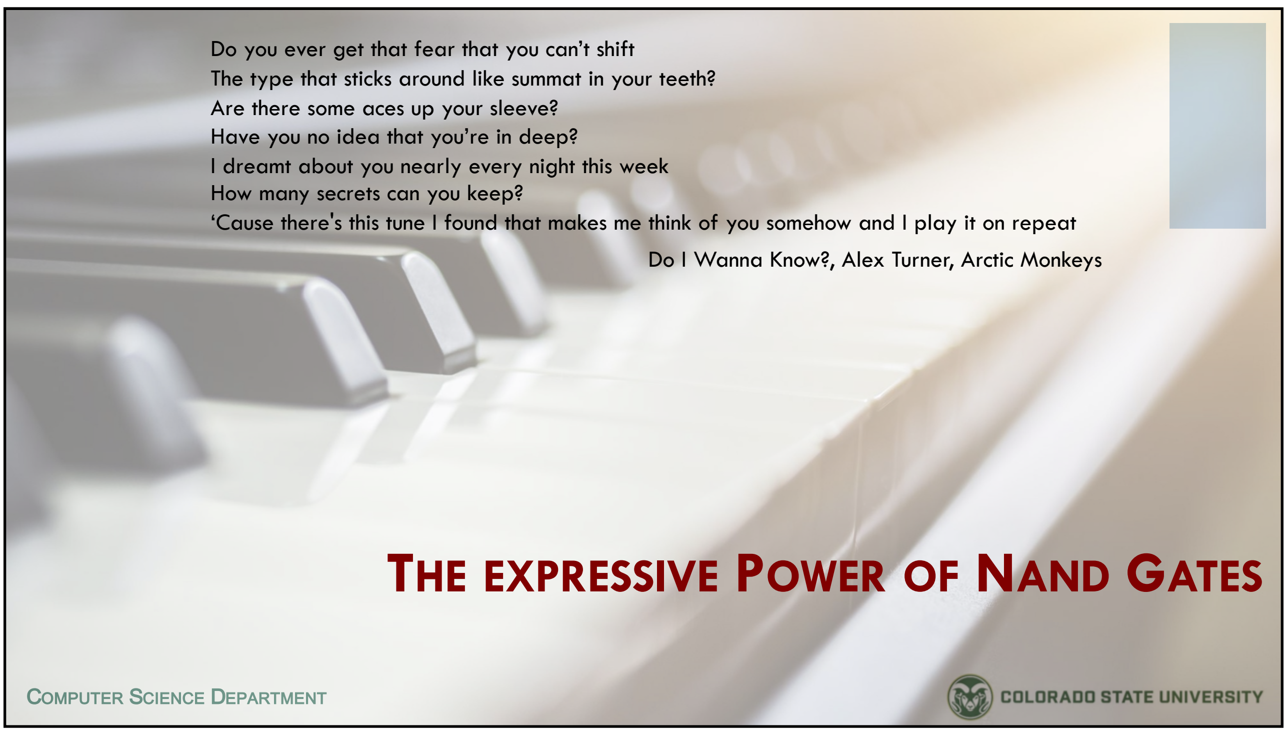
- On the left (positive logic) side, we can make our decision using a single OR operation
- On the right (negative logic) side, De Morgan's law allows us to make our decision using a single AND operation

Practical Implications of the previous example and De Morgan's law in general

- Without De Morgan's law, we'd have to implement the negative logic case as NOT not-cold OR NOT not-raining
- Although that works, there is a cost in price and performance to each operation, so minimizing operations minimizes costs

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT BOOLEAN LOGIC & ALGEBRA L6.11

11



12

We have made the following claims without proof

- Given a truth table representation of a Boolean function, we can *synthesize* from it a Boolean expression that realizes the function
- Any Boolean function can be expressed using only And, Or, and Not operators
- Any Boolean function can be expressed using only Nand operators

Commutative and Idempotent Laws

- **Commutative** Laws
  - x And $y = y$ And x
  - x Or $y = y$ Or x
- **Idempotent** Laws
  - x And $x = x$
  - x Or $x = x$

Associative and Distributive Laws

- **Associative** Laws
  - *x* And (*y* And *z*) = (*x* And *y*) And *z*
  - *x* Or (*y* Or *z*) = (*x* Or *y*) Or *z*
- **Distributive** Laws
  - *x* And (*y* Or *z*) = (*x* And *y*) Or (*x* And *z*)
  - *x* Or (*y* and *z*) = (*x* Or *y*) And (*x* Or *z*)

De Morgan's Laws

- Not (*x* And *y*) = Not(*x*) Or Not(*y*)
- Not (*x* Or *y*) = Not(*x*) And Not(*y*)

Simplifying Boolean Functions

- The algebraic laws we considered could be used to ***simplify*** Boolean functions
- For example, consider the function: Not (Not (*x*) And Not (*x* Or *y*))
  - Can we reduce it to a simpler form?

Not (Not (*x*) And Not (*x* Or *y*))

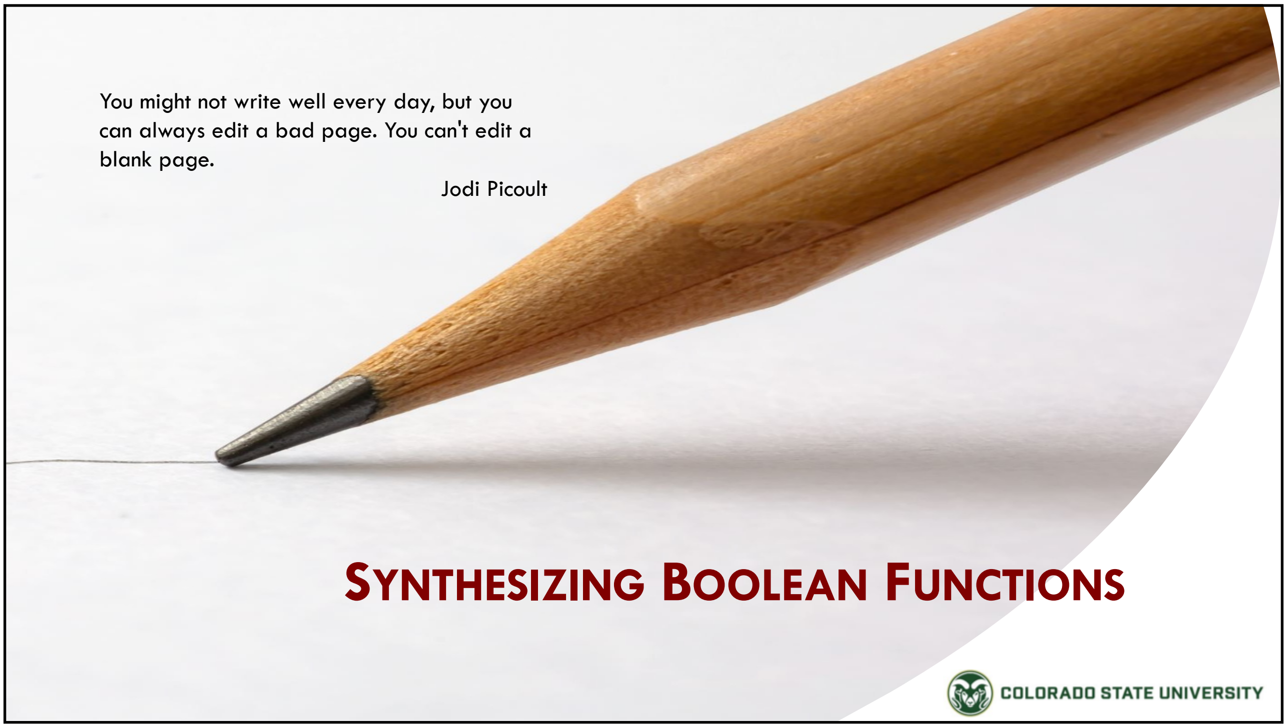
- = Not (Not (*x*) And (Not(*x*) And Not (*y*))) … By De Morgan's Law
- Not (Not (*x*) And (Not(*x*)) And Not (*y*)) … By the associative Law
- Not (Not (*x*) And Not (*y*)) … By the idempotent law
- Not(Not (*x*)) Or Not(Not (*y*)) … By De Morgan's Law
- *x* Or *y* … By double negation

Boolean simplifications like the one we just looked at have significant practical implications

- For example, the original Boolean expression Not (Not (x) And Not (x Or y)) can be implemented in hardware using five logic gates
- Whereas the simplified expression x Or y can be implemented using a single logic gate
  - Both expressions deliver the **same functionality**
  - But the latter (i.e., x Or y) is five times more efficient in terms of cost, energy, and speed of computation

Reducing a Boolean expression into a simpler one is an **art requiring experience and insight**

- Various reduction tools and techniques are available, but the problem remains challenging
- In general, reducing a Boolean expression into its simplest form is an **NP-hard** problem



21

Synthesizing Boolean Functions

- Given a truth table of a Boolean function, how can we construct, or synthesize, a Boolean expression that represents this function?
  - We will look at a **constructive** algorithm to do this
- And, come to think of it, are we guaranteed that every Boolean function represented by a truth table can also be represented by a Boolean expression?
  - Yes!

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT BOOLEAN LOGIC & ALGEBRA L6.22

22

Let's look at a truth table definition of some three-variable function f(x,y,z)

- Our goal is to **synthesize** from these data a Boolean expression that represents this function
- We start by *focusing only* on the truth table's rows in which the function's value is **1**
  - This happens in rows 3, 5, and 7
- For each such row i, we define a Boolean function f_i that returns 0 for all the variable values except for the variable values in row i, for which the function returns 1

x	y	z	$f(x, y, z)$
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

Let's look at a truth table definition of some three-variable function f(x,y,z)

- For each such row i, we define a Boolean function f_i that returns 0 for all the variable values **except** for the variable values in row i, for which the function returns 1
- Each of these functions f_i can be represented by a **conjunction** (And-ing) of three terms, one term for each variable x, y, and z
  - Each being either the variable (or its negation), depending on whether the value of this variable is 1 (or 0) in row i

x	y	z	$f(x, y, z)$
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

Let's look at a truth table definition of some three-variable function f(x,y,z).

- This process yields three such functions

$f_3(x, y, z)$ = (Not(x)) And y And (Not (z))

$f_5(x, y, z)$ = x And Not(y) And (Not (z))

$f_7(x, y, z)$ = x And y And (Not (z))

f (x, y, z) = $f_3(x, y, z)$ Or $f_5(x, y, z)$ Or $f_7(x, y, z)$

x	y	z	$f(x, y, z)$
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

Since these functions describe the only cases in which the Boolean function f evaluates to 1

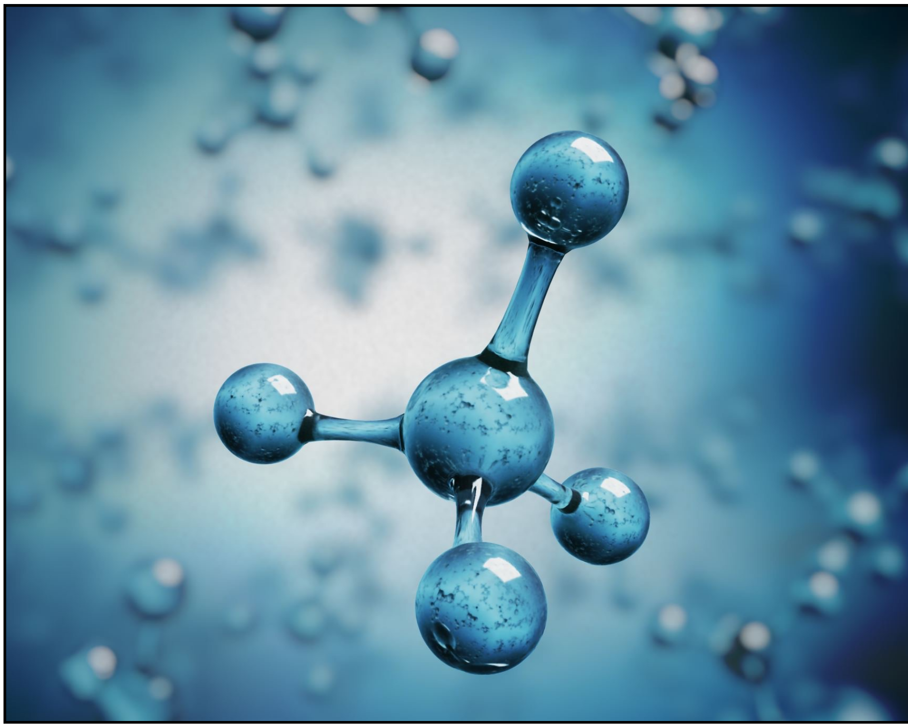
- We conclude that f can be represented by the Boolean expression
  - $f = f_3$ Or f_5 Or f_7
- Spelling it out: (Not (x) And y And Not (z)) Or (x And Not (y) And Not (z)) Or (x And y And Not (z)).

Avoiding tedious formality

- The preceding example suggests that *any* Boolean function can be systematically represented by a Boolean expression that has a very specific structure:
  - It is the **disjunction** (Or-ing) of all the **conjunctive** (And-ing) functions f_i whose construction was just described
- This expression, which is the Boolean version of a **sum of products**, is sometimes referred to as the function's **disjunctive normal form** (DNF)

What if the function has many variables?

- And, thus the truth table has exponentially many rows?
  - The resulting DNF may be long and cumbersome
- At that point, Boolean algebra and various reduction techniques can help transform the expression
  - Into a more efficient and workable representation



THE EXPRESSIVE POWER OF NAND

COLORADO STATE UNIVERSITY

29

Every computer can be built using nothing more than Nand gates

- There are two ways to support this claim
  - One is to actually build a computer from Nand gates only
  - Another way is to provide a formal proof, which is what we'll do next

30

Step One

- **Lemma 1**: Any Boolean function can be represented by a Boolean expression containing only And, Or, and Not operators
- PROOF
  - Any Boolean function can be used to generate a corresponding truth table
  - And, as we've just shown, any truth table can be used for synthesizing a DNF, which is an **Or**-ing of **And**-ings of variables and their **negations**
  - It follows that any Boolean function can be represented by a Boolean expression containing only And, Or, and Not operators

In order to appreciate the significance of this result

- Consider the infinite number of functions that can be defined over integer numbers (rather than binary numbers)
- It would have been nice if every such function could be represented by an algebraic expression involving only addition, multiplication, and negation
- As it turns out, the vast majority of integer functions cannot be expressed using a closed algebraic form
  - For example, $f(x) = 2x$ for $x \neq 7$ and $f(7) = 312$

In the world of binary numbers

- Due to the finite number of values that each variable can assume (0 or 1), we do have an attractive property
  - Every Boolean function can be expressed using nothing more than And, Or, and Not operators
- The practical implication is immense: any computer can be built from nothing more than And, Or, and Not gates

But, can we do better than this?

- **Lemma 2**: Any Boolean function can be represented by a Boolean expression containing only Not and And operators
- PROOF
  - According to De Morgan law, the Or operator can be expressed using the Not and And operators
  - Combining this result with Lemma 1, we get the proof.

Theorem: Any Boolean function can be represented by a Boolean expression containing only Nand operators

- Not (x) = Nand (x, x)
  - In words: If you set both the x and y variables of the Nand function to the same value (0 or 1), the function evaluates to the negation of that value
- And (x, y) = Not (Nand (x, y))
  - It is easy to show that the truth tables of both sides of the equation are identical
  - We've just shown that Not can be expressed using Nand
  - Combining these two results with Lemma 2, we get that any Boolean function can be represented by a Boolean expression containing only Nand operators

What are the implications? [1/2]

- This remarkable result, may well be called the **fundamental theorem of logic design**
- This stipulates that computers can be built from **one atom** only: a logic gate that realizes the Nand function

What are the implications? [2/2]

- In other words, if we have as many Nand gates as we want, we can **wire them in patterns of activation** that implement any Boolean function
  - All we have to do is figure out the right wiring
- Indeed, most computers today are based on hardware infrastructures consisting of billions of Nand gates (or Nor gates, which have similar generative properties)

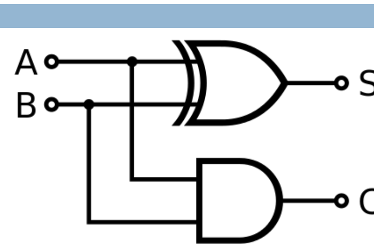
In practice, though, we don't have to limit ourselves to Nand gates only

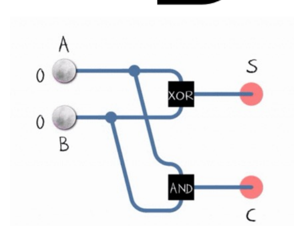
- If electrical engineers and physicists can come up with efficient and low-cost physical implementations of other elementary logic gates, we will happily use them directly as primitive building blocks
- This pragmatic observation does not take away anything from the theorem's importance

39

Logic diagram for a half-adder

A	*B*	C_{out}	*S*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

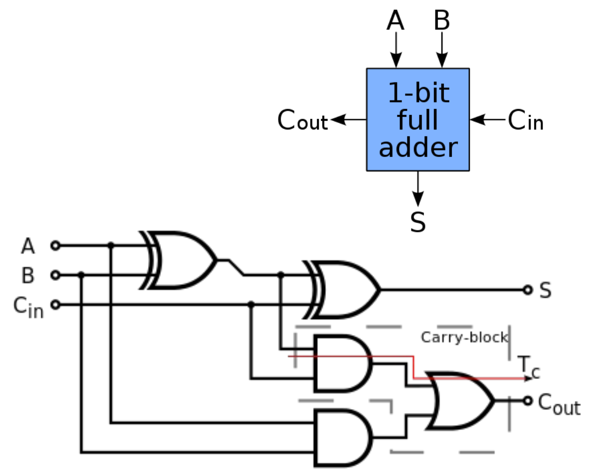




40

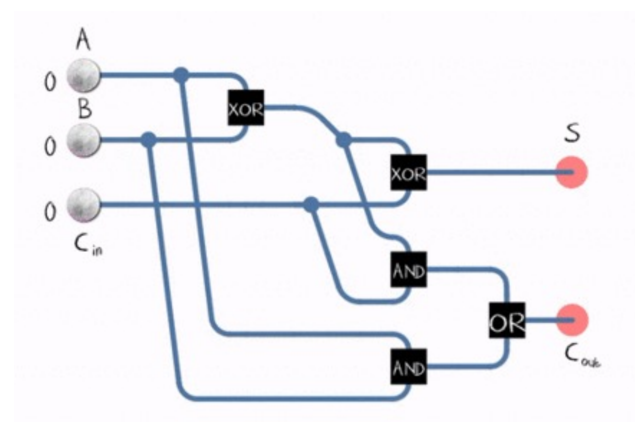
Logic diagram for a full-adder [1/2]

A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

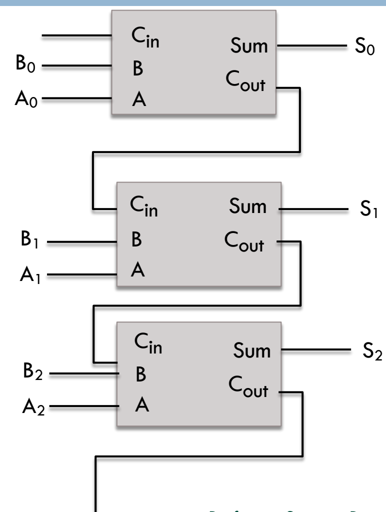


Logic diagram for a full-adder [2/2]

A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Ripple adder: Adding multiple bits [1/2]



Ripple adder: Adding multiple bits [2/2]

- This ripple-carry adder gets its name from the way that **the carry ripples from one bit to the next**
  - It's like doing the wave
- This works fine, but you can see that there are delays per bit, which adds up fast if we're building a 32- or 64-bit adder
- These delays are substantially alleviated in the carry lookahead adder

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]
- Randall Hyde. Write Great Code, Volume 1, 2nd Edition: Understanding the Machine 2nd Edition. ASIN: B07VSC1K8Z. No Starch Press. 2020. [Chapter 2]
- https://en.wikipedia.org/wiki/Adder_(electronics)