Utrecht University

Logic for Computer Science

04 – Boolean algebra

Wouter Swierstra

University of Utrecht

Last time

Naive set theory

This lecture

Boolean algebra

Computer circuits

Binary arithmetic

Organization

Several students pointed out mistakes in (the solutions) the book.

We have a collection of errata here:

`https://github.com/jaccokrijnen/modelling-computing-systems-errata/blob/master/errata.md`

If you think there's a mistake, you may want to check this page.

Please open an issue or pull request if you find any new mistakes!

Boolean algebra

Similarities

We have seen the same equivalences between **sets** and **propositions** – for example, the following two equations hold:

1. $p \lor q \Leftrightarrow q \lor p$
2. $A \cup B = B \cup A$

The similarity extends far beyond this equation...

Challenge

Can we find a general definition, describing the operations and equalities of **both** sets and propositional logic?

With such a definition, we can prove an property once for **both** settings.

Abstract algebra

In mathematics, the field of *algebra* (or abstract algebra more specifically) involves studying mathematical structures.

These structures typically consist of a set, operations on the elements of this set, and the equations that these operations must satisfy.

Sound abstract?

Example: monoid

A **monoid** consists of:

- a set A
- an element $e \in A$
- a binary operator $\oplus$

That satisfy the following three laws, for all x, y and z:

1.	$e \oplus x = x$	(*Ident1*)
2.	$x \oplus e = x$	(*Ident2*)
3.	$x \oplus (y \oplus z) = (x \oplus y) \oplus z$	(*Associativity*)

We sometimes say that e is the **unit** of $\oplus$;

The third law states that the operator $\oplus$ is **associative**.

Monoids are everywhere

- the set of propositions using $\bigvee$ and F;
- the set of propositions using $\bigwedge$ and T;
- natural numbers using + and 0;
- natural numbers using × and 1;
- natural numbers using 0 and max;
- strings using the empty string and string concatenation;
- Imperative programs using `;` and `skip`;
- ...

Monoids pop up everywhere!

So what?

Claim: There unit element of any monoid is unique.

So what?

Claim: There unit element of any monoid is unique.

Proof: Suppose there are two candidate unit elements, e and e'. Then we know:

$$
\begin{aligned}
e &= e \oplus e' && (\textit{Ident1}) \\
  &= e' && (\textit{Ident2})
\end{aligned}
$$

And hence e and e' must be equal.

So what?

Claim: There unit element of any monoid is unique.

Proof: Suppose there are two candidate unit elements, e and e'. Then we know:

$$
\begin{aligned}
e &= e \oplus e' && (\mathit{Ident1}) \\
  &= e' && (\mathit{Ident2})
\end{aligned}
$$

And hence e and e' must be equal.

This proof works for propositions, natural numbers, strings, and **any** monoid in general.

Boolean algebra

Abstract algebra

Monoids are a very simple example of an algebraic structure – there are plenty of other structures such as groups, rings, or fields that are studied extensively.

But what kind of structure generalizes the algebraic structure on propositions and sets?

Boolean algebra

A **Boolean algebra** consists of:

- a set B;
- two elements, $0 \in B$ and $1 \in B$, called the **zero** and **unit** respectively;
- two binary operators $+$ and $\cdot$, called the **sum** and **product** respectively;
- a unary operator $^{-1}$ called the **inverse** (written as x' rather than x^{-1} in the book).

What laws should these satisfy?

Laws of boolean algebras

Commutativity

$x + y = y + x$

$x \cdot y = y \cdot x$

Associativity

$(x + y) + z = x + (y + z)$

$(x \cdot y) \cdot z = x \cdot (y \cdot z)$

Distributivity

$x + (y \cdot z) = (x + y) \cdot (x + z)$

$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$

Laws of boolean algebras

Identity

$x + 0 = x$

$x \cdot 1 = x$

Complement

$x + x^{-1} = 1$

$x \cdot x^{-1} = 0$

These laws generalize the versions we saw for propositional logic and sets.

Notes on these laws

A law such as the following:

Commutativity

$x + y = y + x$

Here we (implicitly) state that this equality holds *for all possible choices of* x and y.

For example:

- $a + b = b + a$
- $y + x = x + y$
- $(c \cdot x^{-1}) + 1 = 1 + (c \cdot x^{-1})$
- ...

Notes on these laws

A law such as the following:

Commutativity

$x + y = y + x$

Not only holds for all *x* and *y*, it may be used in any bigger expression. For example:

- $z + (x + y) = z + (y + x)$
- $(a \cdot b + c) \cdot d = (c + a \cdot b) \cdot d$
- $0 + 1 = 1 + 0$

Notes on boolean algebras

The operations in a boolean algebra such as + seem like the familiar operations on numbers – but they are not!

The distributivity property that states:

$x + (y \cdot z) = (x + y) \cdot (x + z)$

for example does not hold for numbers, addition and multiplication.

There are many other properties that we can prove that do not make any sense when you read these formulas as numbers.

Example proof

Lemma for all x, $x + x = x$.

Example proof

Lemma for all x, $x + x = x$.

$$
\begin{aligned}
x + x &= (x + x) \cdot 1 && (\textit{Ident}) \\
&= (x + x) \cdot (x + x^{-1}) && (\textit{Compl}) \\
&= x + (x \cdot x^{-1}) && (\textit{Distr}) \\
&= x + 0 && (\textit{Compl}) \\
&= x && (\textit{Ident})
\end{aligned}
$$

This is another equality does not hold for numbers!

Boolean algebras are not numbers

Lemma for all x, $x + x = x$.

This does not hold for numbers.

Although boolean algebras assume that there are a 0 and 1 element – these are not ‘the number 0’ and ‘the number 1’.

There need not be an element 2 (or 3, or 4, or 5).

In general, we do not know anything about the *elements* of the set underlying a boolean algebra.

They could be sets, numbers, propositions, or anything at all!

Proofs using boolean algebra

Lemma for all x, $x + x = x$.

$$
\begin{aligned}
x + x &= (x + x) \cdot 1 && (\textit{Ident}) \\
&= (x + x) \cdot (x + x^{-1}) && (\textit{Compl}) \\
&= x + (x \cdot x^{-1}) && (\textit{Distr}) \\
&= x + 0 && (\textit{Compl}) \\
&= x && (\textit{Ident})
\end{aligned}
$$

The proof starts from the left-hand side $x + x$.

Each step applies one of the properties of boolean algebras (or possibly another lemma).

Each step is *annotated* with the law being used.

Lemma for all x, $x + x = x$.

$x + x$	$= (x + x) \cdot 1$	(*Ident*)
	$= (x + x) \cdot (x + x^{-1})$	(*Compl*)
	$= x + (x \cdot x^{-1})$	(*Distr*)
	$= x + 0$	(*Compl*)
	$= x$	(*Ident*)

We then chain these steps together until we reach the desired right-hand side of the lemma we're proving – in this example, x.

This gives a proof showing that x and $x + x$ are equal, going through several intermediate steps.

We can use each property in either direction – from left to right or right to left.

These proofs are built a bit differently than the usual ‘solving of equations’ that you may be familiar with from high school.

Equations

$x^2 - x = 6$

$\Leftrightarrow$

$x^2 - x - 6 = 0$

$\Leftrightarrow$

$(x + 2)(x - 3) = 0$

From which we can conclude that $x = 3$ or $x = -2$.

Boolean algebras

$x + x$	$= (x + x) \cdot 1$	(*Ident*)
	...	...
	$= x$	(*Ident*)

Here *prove an equality*, rather than *solve an equation*.

To prove this equality, we connect each step by proving it is equal to the previous one.

When we solve an equation, we (implicitly) connect each step using the logical equivalence operator (if-and-only-if).

Proving theorems over boolean algebras

$$
\begin{aligned}
x + x &= (x + x) \cdot 1 && (\textit{Ident}) \\
&\ldots && \ldots \\
&= x && (\textit{Ident})
\end{aligned}
$$

A proof of a *theorem* over boolean algebras consists of a series of equalities chained together. To prove the theorem A = B:

- start with A, apply a law of boolean algebras to prove A = A′;
- continue the proof by showing A′ = B, until you are done.

Every step is labelled with the property or axiom being used to show both sides of the equation are equal. Chaining together all these equalities gives the complete proof.

Remember: You are not solving an equation (high school) but proving an equality!

Remainder of lecture

Applications and examples of boolean algebra

- **Examples** – powersets and booleans
- **Derived equations** – if we can prove derived properties of boolean algebras in terms of the laws, we know that these properties hold for **every** boolean algebra.
- **Applications** – design and optimization of digital circuits

Boolean algebras - examples

Powersets

Given any set U, the powerset P(U) forms a boolean algebra with:

- the empty set as 0
- U as 1
- union as +
- intersection as ·
- complement as $^{-1}$

Exercise: convince yourself that all the laws hold as you would expect using a Venn diagram.

(The laws that you need to check can be found in Figure 3.1 in the book)

0-1

If we take **B** to be the set {0,1}.

B forms a boolean algebra, with the operators:

- $\bigvee$ (for +)
- $\bigwedge$ (for ·)
- and ¬ (for $^{-1}$).

In fact, we can see familiar behaviour from numbers show up in a very different context…

Sum

x	y	$x + y$
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

Disjunction ($\vee$) ‘behaves the same as addition’.

Product

x	y	$x \cdot y$
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

Conjunction ($\wedge$) ‘behaves the same as multiplication’.

Derived theorems

Given the following three laws:

1. $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
2. $x + y = y + x$
3. $x \cdot y = y \cdot x$

Question:

Prove $(x + y) \cdot z = (x \cdot z) + (y \cdot z)$

Derived theorems

Given the following three laws:

1. $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
2. $x + y = y + x$
3. $x \cdot y = y \cdot x$

Question:

Prove $(x + y) \cdot z = (x \cdot z) + (y \cdot z)$

Any equation derived from the laws is known as a **theorem**

Derived theorems

We have seen the proof that for all x, we have $x + x = x$.

$x + x$	$= (x + x) \cdot 1$	(*Ident*)
	$= (x + x) \cdot (x + x^{-1})$	(*Compl*)
	$= x + (x \cdot x^{-1})$	(*Distr*)
	$= x + 0$	(*Compl*)
	$= x$	(*Ident*)

This statement says something about sets and unions, propositions and disjunction, and *any* other structure that satisfies the properties of a Boolean algebra.

Proving this once – using only the laws of boolean algebras – guarantees this property holds **in every boolean algebra**.

Derived theorems

Many of these derived theorems follow use the following property.

For all x, y, and z, if $x + y = x + z$ and $x \cdot y = x \cdot z$, then $y = z$.

In other words, if y and z 'behave the same' on every element x, we can conclude that y and z are equal.

Derived theorems

Many of these derived theorems follow use the following property.

For all x, y, and z, if $x + y = x + z$ and $x \cdot y = x \cdot z$, then $y = z$.

In other words, if y and z 'behave the same' on every element x, we can conclude that y and z are equal.

We can use this to show that $(x^{-1})^{-1} = x$.

De Morgan, revisited

We can even generalize de Morgan's laws to work over any boolean algebra:

- $(x + y)^{-1} = x^{-1} \cdot y^{-1}$
- $(x \cdot y)^{-1} = x^{-1} + y^{-1}$

I'll refer to the book for the proofs.

Another example proof

Question

Show that the following equation holds for all x in every boolean algebra:

$$x + 1 = 1$$

Using the idempotence law: $x + x = x$.

Another example proof

Question

Show that the following equation holds for all x in every boolean algebra:

$$x + 1 = 1$$

Using the idempotence law: $x + x = x$.

$x + 1$	$= x + (x + x')$	(*Compl*)
	$= (x + x) + x'$	(*Assoc*)
	$= x + x'$	(*Idempotence*)
	$= 1$	(*Compl*)

Another example proof

Question

Show that the following equation holds for all x in every boolean algebra:

$$x + 1 = 1$$

Using the idempotence law: $x + x = x$.

$x + 1$	$= x + (x + x')$	(*Compl*)
	$= (x + x) + x'$	(*Assoc*)
	$= x + x'$	(*Idempotence*)
	$= 1$	(*Compl*)

What about proving $x \cdot 0 = 0$?

Duality

Given any expression or equation in a Boolean algebra, we can construct a new one by 'reversing' all operations – that is:

- replace 0 by 1
- replace 1 by 0
- replace + by ·
- replace · by +

For example, the dual of $x + (y^{-1} \cdot z) = 1$ is $x \cdot (y^{-1} + z) = 0$.

Exercise: What is the dual of $x \cdot (y^{-1} + z) = 0$?

Duality

Given any expression or equation in a Boolean algebra, we can construct a new one by ‘reversing’ all operations – that is:

- replace 0 by 1
- replace 1 by 0
- replace + by ·
- replace · by +

For example, the dual of $x + (y^{-1} \cdot z) = 1$ is $x \cdot (y^{-1} + z) = 0$.

Exercise: What is the dual of $x \cdot (y^{-1} + z) = 0$?

Essentially, this is ‘mirroring’ every operation and constant.

Duality

Duality theorem

The dual of every theorem in a Boolean algebra is also a theorem.

Duality

Duality theorem

The dual of every theorem in a Boolean algebra is also a theorem.

Proof:

For every law of a boolean algebra, its dual is also a law – hence we can mirror thet proof of the original theorem to produce a proof of the dual theorem.

Duality

Duality theorem

The dual of every theorem in a Boolean algebra is also a theorem.

Proof:

For every law of a boolean algebra, its dual is also a law – hence we can mirror thet proof of the original theorem to produce a proof of the dual theorem.

Hence, it suffices to prove either one of the two de Morgan laws:

- $(x + y)^{-1} = x^{-1} \cdot y^{-1}$
- $(x \cdot y)^{-1} = x^{-1} + y^{-1}$

Duality example

We have shown that $x + 1 = 1$.

$$
\begin{aligned}
x + 1 &= x + (x + x') && (\textit{Compl}) \\
&= (x + x) + x' && (\textit{Assoc}) \\
&= x + x' && (\textit{Idempotence}) \\
&= 1 && (\textit{Compl})
\end{aligned}
$$

The dual proof shows that $x \cdot 0 = 0$

$$
\begin{aligned}
x \cdot 0 &= x \cdot (x \cdot x') && (\textit{Compl}) \\
&= (x \cdot x) \cdot x' && (\textit{Assoc}) \\
&= x \cdot x' && (\textit{Idempotence}) \\
&= 0 && (\textit{Compl})
\end{aligned}
$$

Abstraction

A week ago we were discussing fire alarms and evacuating the class room…

Now we've transitioned to the formal manipulation of funny symbols.

It's important to develop an **intuition** for what these symbols mean – and practice makes perfect.

And even though it seems like we've lost all connection with reality, boolean algebra has many, many applications.

Including in the design of the computer hardware…

Computer circuits

Hardware 101

Now - we show how to apply boolean algebra in the optimization of hardware, as you learned in the last period.

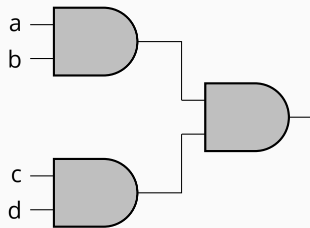
Computer hardware is constructed from *logical gates*.

We will work with the following *model* of hardware:

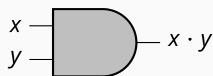
- A circuit takes a collection of *inputs* to produce *outputs*;
- Each input and output takes the form of a wire, carrying a binary value—i.e. 0 or 1.
- These inputs are fed into logical gates;
- Each logical gate behaves according to a specific boolean function.

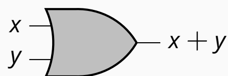
Note that we're ignoring all kinds of aspects of hardware design: physics, timing, current, resistance, magnetic interference, etc.

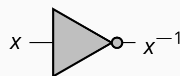
Example: (a·b) · (c · d)



Other gates?







- There are many different logical gates, corresponding to the familiar boolean operations.
- For example, there are gates for conjunction (AND), disjunction (OR), and negation (NOT).

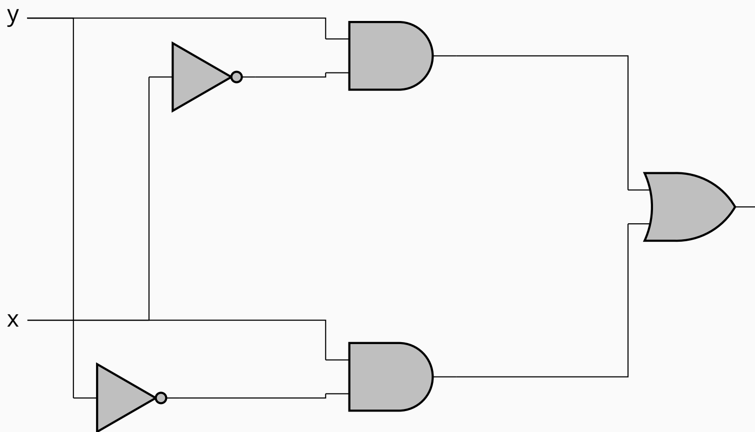
Implementing XOR

x	y	XOR x y
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**

Question:

Implement a circuit that takes the ‘exclusive or’ of two inputs using the AND, OR and NOT gates.

Solution



But many alternatives exist...

Finding a solution

x	y	XOR x y
0	0	0
0	**1**	**1**
1	**0**	**1**
1	1	0

One way to find this solution is by reading off the lines of the truth that return **1** and taking their disjunction:

$(x \cdot y^{-1}) + (x^{-1} \cdot y)$

This works for **any** truth table.

Another problem

x	y	??? x y
0	**0**	**1**
0	**1**	**1**
1	0	0
1	1	0

We can implement this mystery gate as follows:

$(x^{-1} \cdot y^{-1}) + (x^{-1} \cdot y)$

But is this a good choice? It requires 1 OR gate, 2 AND gates, and 3 NOT gates...

Boolean algebra to the rescue

$(x^{-1} \cdot y^{-1}) + (x^{-1} \cdot y)$	$= x^{-1} \cdot (y^{-1} + y)$	(*Distr*)
	$= x^{-1} \cdot 1$	(*Compl*)
	$= x^{-1}$	(*Ident*)

We only need a single gate – the problem was much simpler than we thought initially!

Boolean algebra to the rescue

$(x^{-1} \cdot y^{-1}) + (x^{-1} \cdot y)$	$= x^{-1} \cdot (y^{-1} + y)$	(*Distr*)
	$= x^{-1} \cdot 1$	(*Compl*)
	$= x^{-1}$	(*Ident*)

We only need a single gate – the problem was much simpler than we thought initially!

Of course this is an artificial example – but in realistic circuits such optimizations can really matter!

Logical gates

Built from these primitive gates, there are plenty of bigger circuits:

- *multiplexers* are similar to if-then-else statements, that produce one of two inputs depending on a selection bit;
- *demultiplexers* takes an input and selection bit, and sends the input to one of two possible outputs, depending on the selection bit.

Sequential circuits introduce loops, allowing the circuit to store information from one clock cycle to the next – this is enables you to have *memory*.

Binary arithmetic

Binary arithmetic

We can do much more with logical gates than just manipulate boolean formulas.

We can perform all kinds of *arithmetic*.

Next up, we'll study how to model integers and write circuits that manipulate them.

This forms the heart of the Arithmetic Logical Unit (ALU) – an important part of any modern CPU.

Binary numbers

We're used to writing a number in *base 10*. A sequence of digits, like 642, is read as

$$642 = (6 \times 10^2) + (4 \times 10^1) + (2 \times 10^0)$$

We can interpret a sequence of binary digits (0's and 1's) in a similar fashion.

$$\begin{aligned} 10011 &= (1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\ &= 16 + 0 + 0 + 2 + 1 \\ &= 19 \end{aligned}$$

Claim: We can write **any** number as a sum of powers of two.

Negative numbers

If we only represent positive numbers in this fashion, we call these binary numbers **unsigned**.

But we can also represent positive and negative numbers as **signed** integers.

The most common representation of **signed** integers is the so-called **two's complement** method.

We interpret five bits, such as 10011, as follows.

- the **first** bit is interpreted as -2^5;
- the **last** four bits are interpreted as normally;

You can observe this yourself: if you add two huge numbers (in most programming languages), this can sometimes produce a negative result.

Negative numbers

Bits (positive)	value	Bits (negative)	value
00000	0	10000	-16
00001	1	10001	-15
00010	2	10010	-14
00011	3	10011	-13
00100	4	10100	-12
00101	5	10101	-11
...	...	...	...
01110	14	11110	-2
01111	15	11111	-1

If you have ever played *Civilization*, you're very aware that Gandhi tends to be the first to use nuclear weapons, and spares no expense on wiping your civilization off the map. You probably always thought you were crazy — how could a series that prides itself on historical accuracy portray Gandhi so wrong? Well, you'll be happy to know that both your sanity and *Civilization's* historical integrity aren't at fault. Instead, a **bug's to blame**.

In the earlier *Civs*, leaders are given a set of attributes that dictate their behavior. One such attribute is a number scale associated with aggressiveness. Gandhi was given the lowest number possible, a rating of 1. However, when a civilization adopted democracy, it granted a civilization -2 to opponent aggression levels. This sent Gandhi's rating of 1 into the negative, which swung it back around to 255 — the highest possible rating available, and thus, the infamous warmonger Gandhi was born.

Adding binary numbers

We won’t be too concerned with negative numbers for the remainder of the lecture.

Suppose we want do design a circuit that adds two binary numbers.

How might we go about doing this?

Adding binary numbers on paper

Question:

What is the result of 01101 + 00111?

Adding binary numbers on paper

Question:

What is the result of 01101 + 00111?

As humans, we can convert to the usual decimal notation add the two numbers and convert back.

1 + 4 + 8 + 1 + 2 + 4 = 20

But that’s certainly no the best way to perform binary addition.

Adding binary numbers

```
   01101
+  00111
---------
   ?????
```

Adding binary numbers

```
   1111
   01101
+  00111
---------
   10100
```

10100 is 20 in decimal – the familiar addition principles of addition work just as well on binary numbers

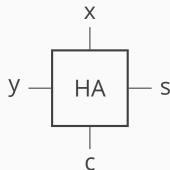
Designing an adder

To construct an **adder** that adds two n-bit binary numbers, we'll proceed in a few steps:

- Design **half-adder** – a circuit to add two bits, producing a sum and a carry bit;
- Combine such half-adders to a **full-adder** that computes the sum of two n-bit words.

In what follows, we'll focus on 4-bit adders – but the techniques can be generalized to arbitrary width adders.

Half-adder



The half-adder takes two bits, x and y, as inputs to produce a sum and a carry bit.

How should it behave?

Half-adder behaviour

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Suppose we want to add x and y to produce a sum bit s and carry c.

The sum is just the XOR we saw previously; the carry is computed by taking the AND of x and y.

Half-adder

We build a bigger circuit, implementing XOR as we saw previously and taking the AND of both inputs to compute the carry.

Half-adder

We build a bigger circuit, implementing XOR as we saw previously and taking the AND of both inputs to compute the carry.

Alternatively, I claim we can write:

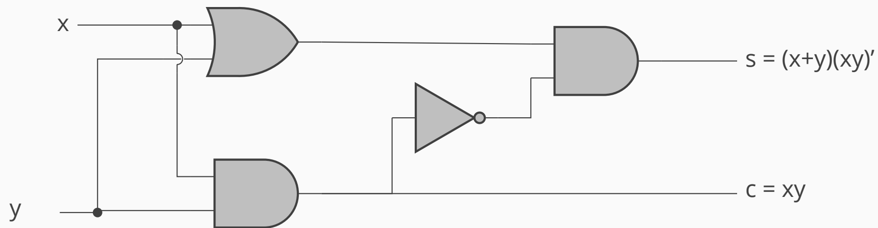
$s = (x + y) \cdot (x \cdot y)^{-1}$

$c = (x \cdot y)$

Now the subexpression $(x \cdot y)$ can be **shared** in both computations, resulting in fewer gates necessary.

The idea is to compute $(x \cdot y)$ once, but use the result in both s and c.

Half-adder



But...

But we'd still need to check that our previous version of XOR and current formulation are equal.

Put differently, does the following equality hold?

$$(x \cdot y^{-1}) + (x^{-1} \cdot y) = (x + y) \cdot (x \cdot y)^{-1}$$

But...

But we'd still need to check that our previous version of XOR and current formulation are equal.

Put differently, does the following equality hold?

$(x \cdot y^{-1}) + (x^{-1} \cdot y) = (x + y) \cdot (x \cdot y)^{-1}$

We still need to find a proof!

Proof

$(x \cdot y^{-1}) + (x^{-1} \cdot y)$

$= ((x \cdot y) + x^{-1} \cdot y^{-1})^{-1}$

$= (x \cdot y)^{-1} \cdot (x^{-1} \cdot y^{-1})^{-1}$

$= (x \cdot y)^{-1} \cdot (x^{-1\,-1} + y^{-1\,-1})$

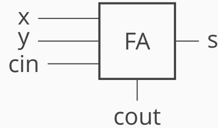
$= (x \cdot y)^{-1} \cdot (x + y)$

$= (x + y) \cdot (x \cdot y)^{-1}$

Homework

Identify all the laws that have been used here.

Full-adders



A half-adder lets us add two input bits to produce a sum and a carry.

A **full-adder** takes two input bits and a carry-in bit and produces a sum and carry-out bit.

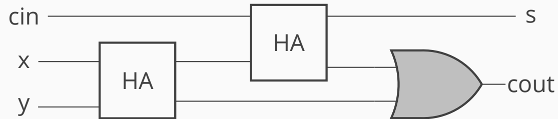
We can create a full-adder from two half-adders.

Full-adders

cin	y	x	s	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- `s` is 1 when one or all three inputs are 1;
- `cout` is 1 when at least two inputs are 1.

Full-adders



Use a half-adder to add x and y;

Add the sum to the carry in;

Take the disjunction of both carry outs.

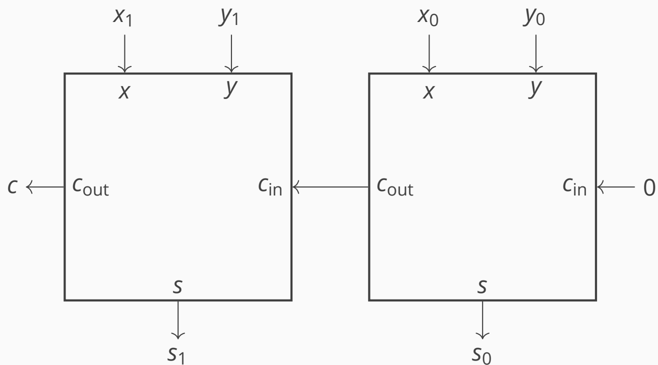
A 2-bit adder

From two full-adders, we can construct a 2-bit adder.

- Use a full-adder to add the least significant two bits, with the initial carry being 0;
- Use a second full-adder to add the most significant two bits, together with the carry from the first addition.

This gives us two sum bits and a final carry bit.

A 2-bit adder



From a 2-bit adder to a 32-bit adder

The 2-bit adder performs a single ‘column’ of addition of binary numbers.

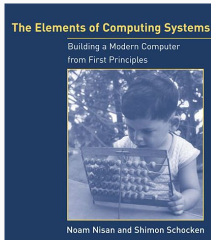
Generalising this to 32-bits is not hard – but essentially repeats a row of full-adders that are linked together appropriately.

In a similar fashion, all kinds of more complicated arithmetic circuits can be constructed for multiplication, division, modulo, and so forth.

Similarly, we can construct circuits to handle the memory access, caching, and handling all the typical operations from some binary instruction set.

Recap

- We can generalize the structure of both propositional logic and sets to a *boolean algebra*;
- Using such an algebra, we can prove theorems that hold for *every* algebra – such as duality.
- We can use the algebraic structure to reason about logical gates in hardware design.



Elements of computing systems: Building a modern computer from first principles

Covers a bit of everything from:

- bits to binary arithmetic;
- computer architecture;
- assembler and programming languages;
- operating systems

Every new abstraction is built up from the previous pieces.

Learning the laws

Do you need to know the laws of boolean algebras by heart for the exam?

Learning the laws

Do you need to know the laws of boolean algebras by heart for the exam?

Yes.

Although I will not ask you to write down all ten laws – I will almost certainly ask you to do a proof; or show that some structure is a boolean algebra; or ask whether a given operator is commutative or not.

I want you to *understand* concepts like commutativity, distributivity, associativity, etc.

So yes, this means study and practice to learn what these words mean.

Proofs about boolean algebras

On the exam, I may ask you to prove a theorem about boolean algebras.

I'll typically give you any auxiliary theorems that you'll need, and possibly hint at one direction to take the proof.

I find many of the proofs quite 'unintuitive':

How do I know that I should change x to $x + 0$ to $x + (x \cdot x')$?

Proofs about boolean algebras

On the exam, I may ask you to prove a theorem about boolean algebras.

I'll typically give you any auxiliary theorems that you'll need, and possibly hint at one direction to take the proof.

I find many of the proofs quite 'unintuitive':

How do I know that I should change x to $x + 0$ to $x + (x \cdot x')$?

I don't know of a 'good' way to find these proofs. Practice helps you recognise common patterns – but often it requires a bit of fumbling around.

Proofs about boolean algebras

On the exam, I may ask you to prove a theorem about boolean algebras.

I'll typically give you any auxiliary theorems that you'll need, and possibly hint at one direction to take the proof.

I find many of the proofs quite 'unintuitive':

How do I know that I should change x to $x + 0$ to $x + (x \cdot x')$?

I don't know of a 'good' way to find these proofs. Practice helps you recognise common patterns – but often it requires a bit of fumbling around.

When it comes to *proof strategies* that we'll see next week will give us a much clearer toolbox to find and write proofs in a structured fashion.

Material

- Modelling Computing Systems Chapter 3