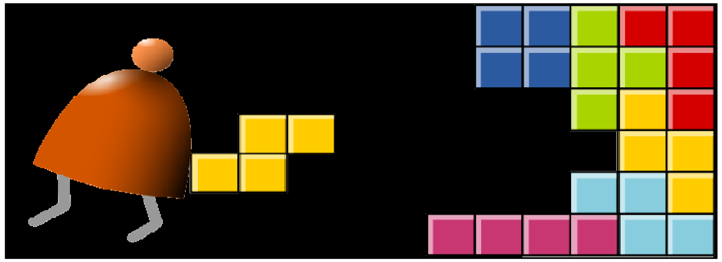
Boolean Logic



Building a Modern Computer From First Principles

www.nand2tetris.org

Boolean algebra

<u>Some elementary Boolean functions:</u>

- Not(x)
- And(x,y)
- Or(x,y)
- Nand(x,y)

x	Not(x)
0	1
1	0

x	y	And(x,y)
0	0	0
0	1	0
1	0	0
1	1	1

x	y	Or(x,y)
0	0	0
0	1	1
1	0	1
1	1	1

x	y	Nand(x,y)
0	0	1
0	1	1
1	0	1
1	1	0

<u>Boolean functions:</u>

x	y	z	$f(x,y,z)=(x+y)\bar{z}$
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

- A Boolean function can be expressed using a functional expression or a truth table expression
- <u>Important observation:</u>
  Every Boolean function can be expressed using And, Or, Not.

All Boolean functions of 2 variables

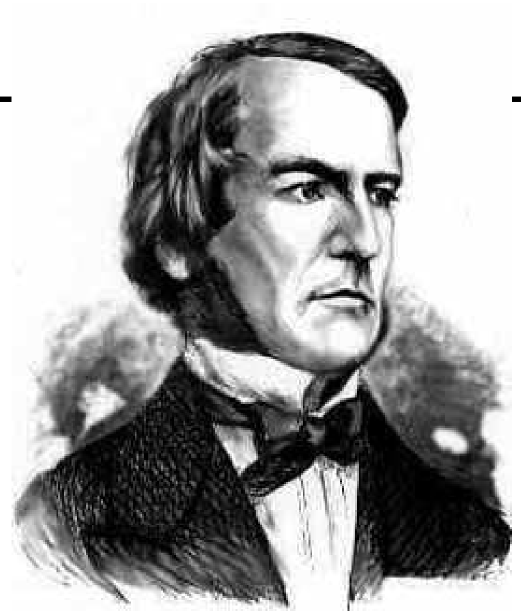
Function	x	**0**	**0**	**1**	**1**
	y	**0**	**1**	**0**	**1**
Constant 0	0	0	0	0	0
And	$x \cdot y$	0	0	0	1
x And Not y	$x \cdot \bar{y}$	0	0	1	0
x	x	0	0	1	1
Not x And y	$\bar{x} \cdot y$	0	1	0	0
y	y	0	1	0	1
Xor	$x \cdot \bar{y} + \bar{x} \cdot y$	0	1	1	0
Or	$x + y$	0	1	1	1
Nor	$\overline{x+y}$	1	0	0	0
Equivalence	$x \cdot y + \bar{x} \cdot \bar{y}$	1	0	0	1
Not y	$\bar{y}$	1	0	1	0
If y then x	$x + \bar{y}$	1	0	1	1
Not x	$\bar{x}$	1	1	0	0
If x then y	$\bar{x} + y$	1	1	0	1
Nand	$\overline{x \cdot y}$	1	1	1	0
Constant 1	1	1	1	1	1

Boolean algebra

Given: **`Nand(a,b), false`**

We can build:

- **`Not(a) = Nand(a,a)`**
- **`true = Not(false)`**
- **`And(a,b) = Not(Nand(a,b))`**
- **`Or(a,b) = Not(And(Not(a),Not(b)))`**
- **`Xor(a,b) = Or(And(a,Not(b)),And(Not(a),b)))`**
- Etc.



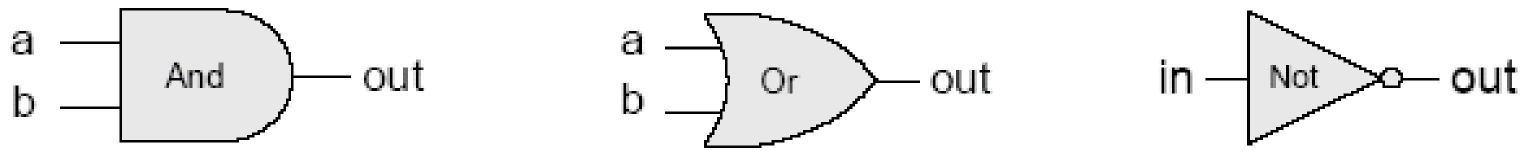
George Boole, 1815-1864

("*A Calculus of Logic*")

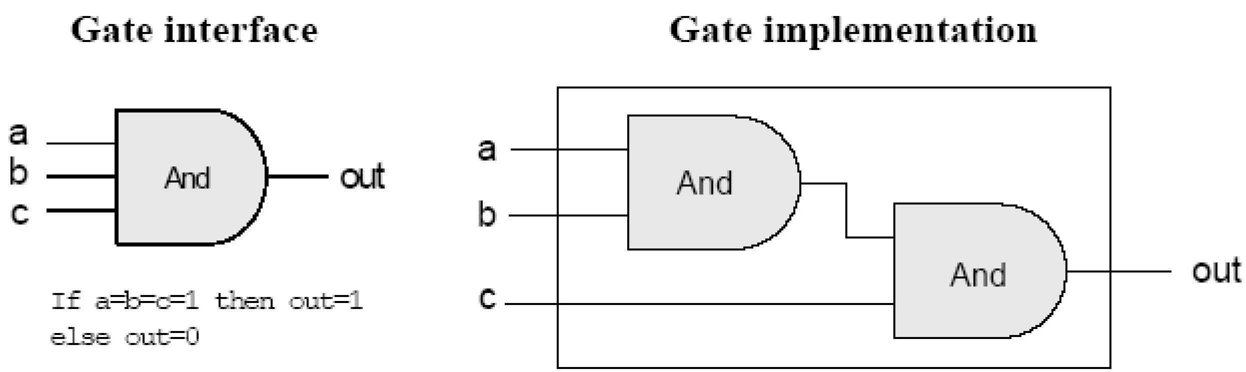
Gate logic

- Gate logic – a gate architecture designed to implement a Boolean function

- Elementary gates:



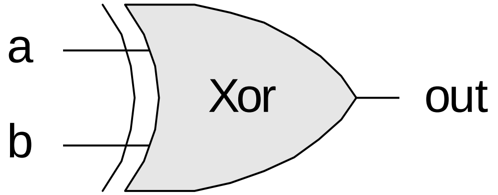
- Composite gates:



- <u>Important distinction:</u> Interface (*what*) VS implementation (*how*).

Gate logic

Interface

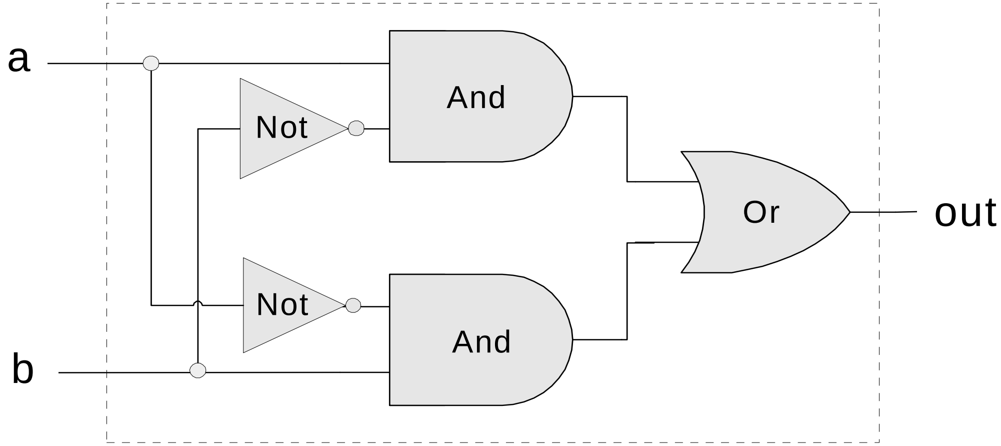


a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Claude Shannon, 1916-2001

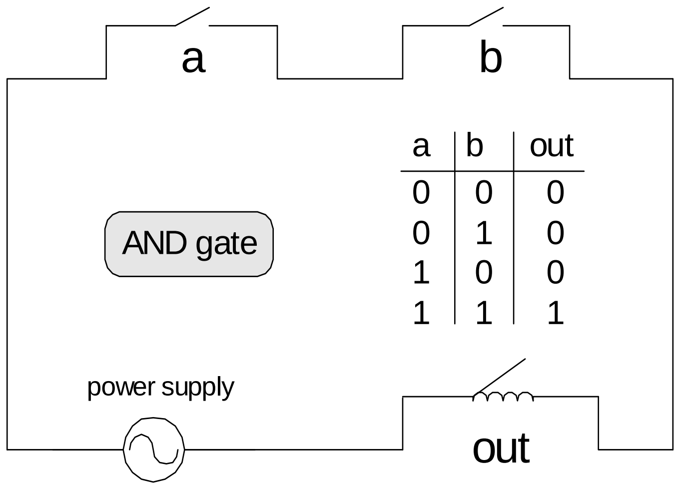
("Symbolic Analysis of Relay and Switching Circuits")

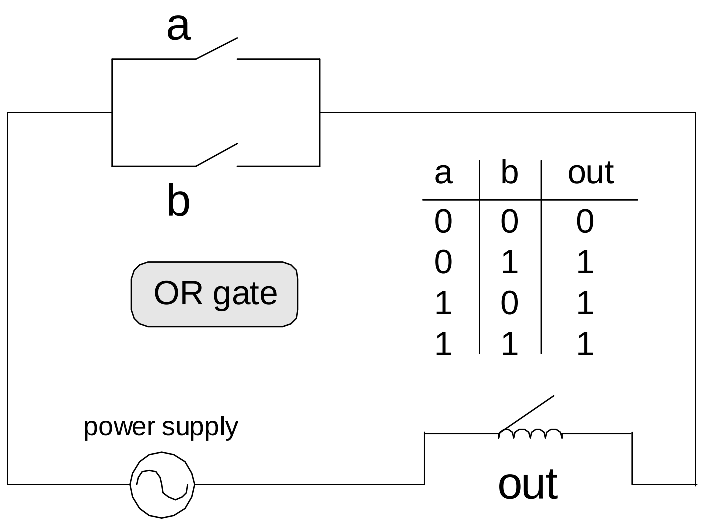
Implementation

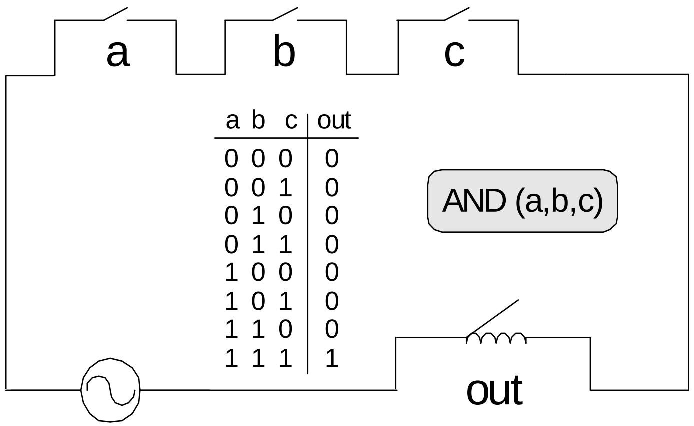


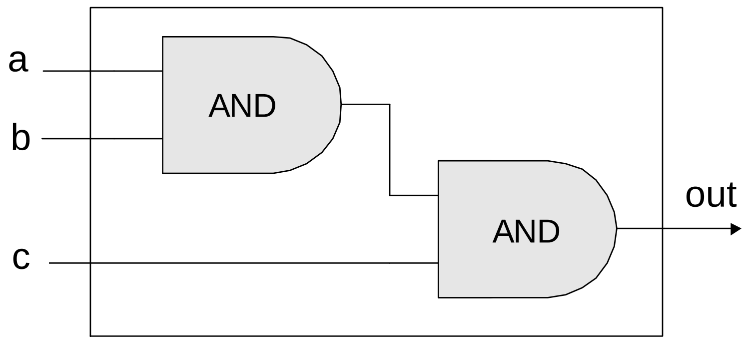
Xor(a,b) = Or(And(a,Not(b)),And(Not(a),b)))

Circuit implementations









- From a computer science perspective, physical realizations of logic gates are irrelevant.

Project 1: elementary logic gates

Given: `Nand(a,b), false`

a	b	Nand(a,b)
0	0	1
0	1	1
1	0	1
1	1	0

Build:

- `Not(a) = ...`
- `true = ...`
- `And(a,b) = ...`
- `Or(a,b) = ...`
- `Mux(a,b,sel) = ...`
- Etc. - 12 gates altogether.

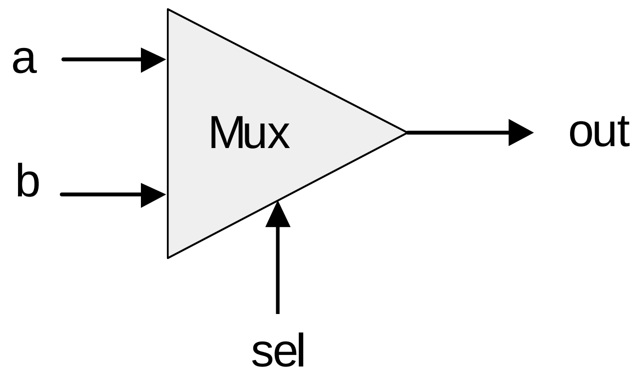
Q: Why these particular 12 gates?

A: Since …

- They are commonly used gates
- They provide all the basic building blocks needed to build our computer.

Multiplexer

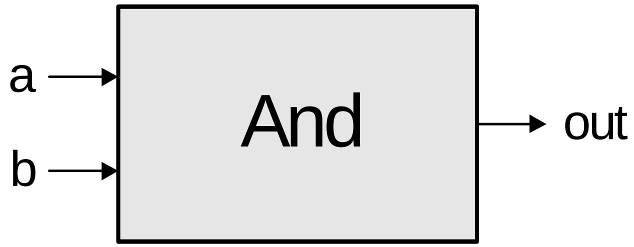
a	b	sel	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



sel	out
0	a
1	b

Proposed Implementation: based on Not, And, Or gates.

Example: Building an **And** gate



And.cmp

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

Contract:

When running your **.hdl** on our **.tst**, your **.out** should be the same as our **.cmp**.

And.hdl

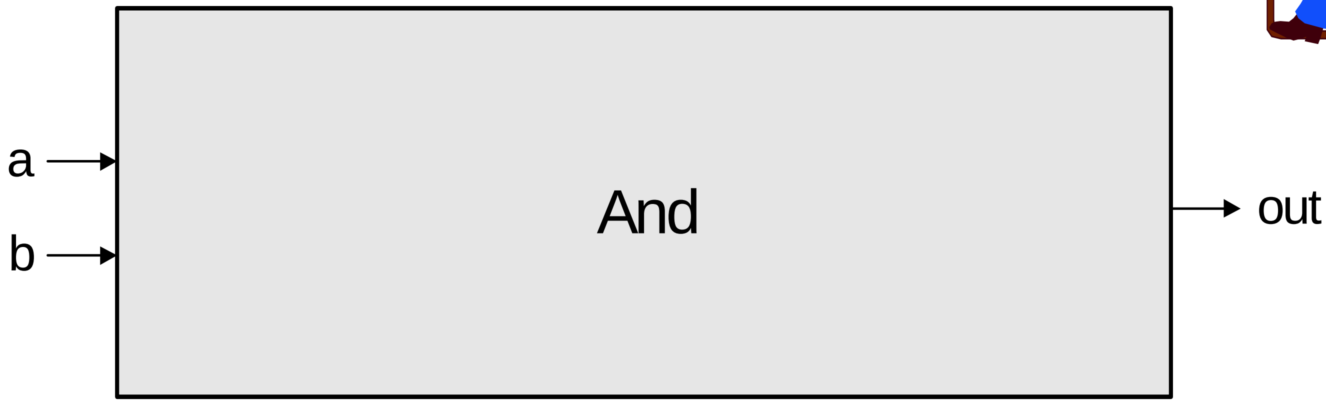
```
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```

And.tst

```
load And.hdl,
output-file And.out,
compare-to And.cmp,
output-list a b out;
set a 0,set b 0,eval,output;
set a 0,set b 1,eval,output;
set a 1,set b 0,eval,output;
set a 1, set b 1, eval, output;
```

Building an **And** gate

Interface: And(a,b) = 1 exactly when a=b=1

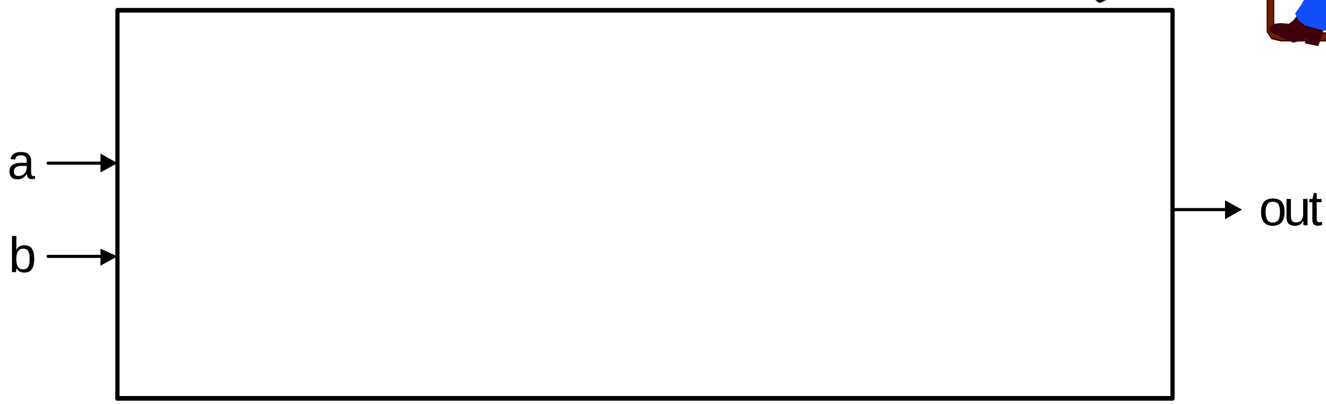


And.hdl

```
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```

Building an **And** gate

Implementation: And(a,b) = Not(Nand(a,b))

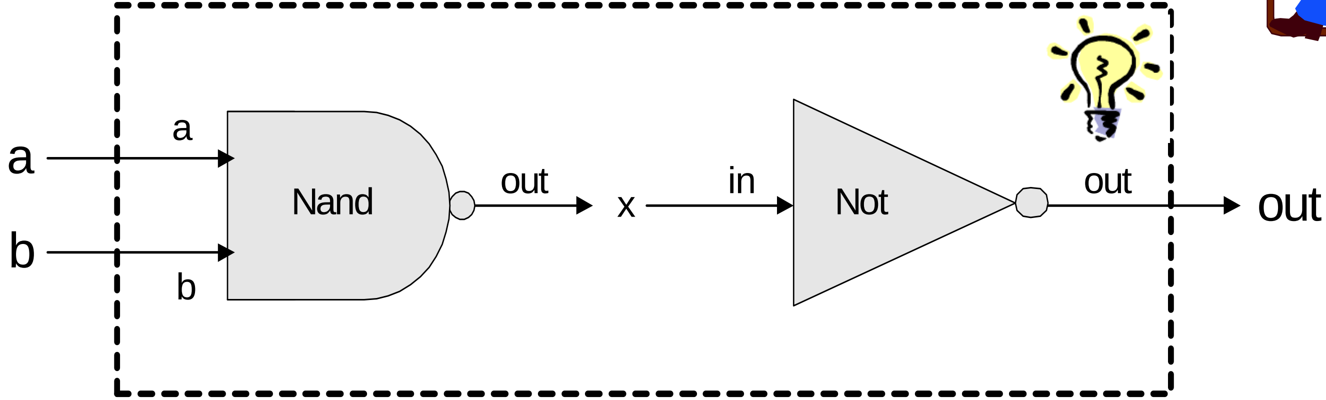


And.hdl

```
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```

Building an **And** gate

Implementation: And(a,b) = Not(Nand(a,b))

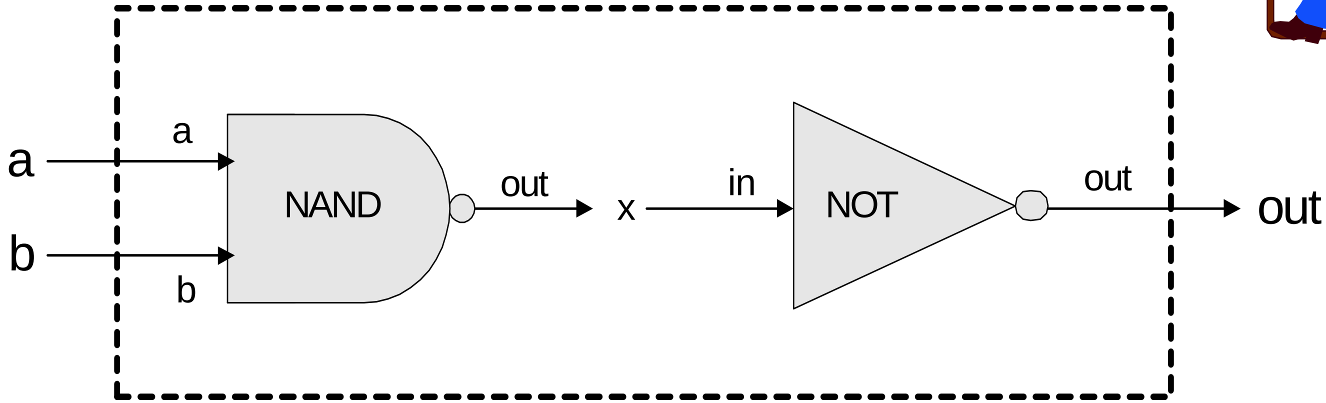


And.hdl

```
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```

Building an **And** gate

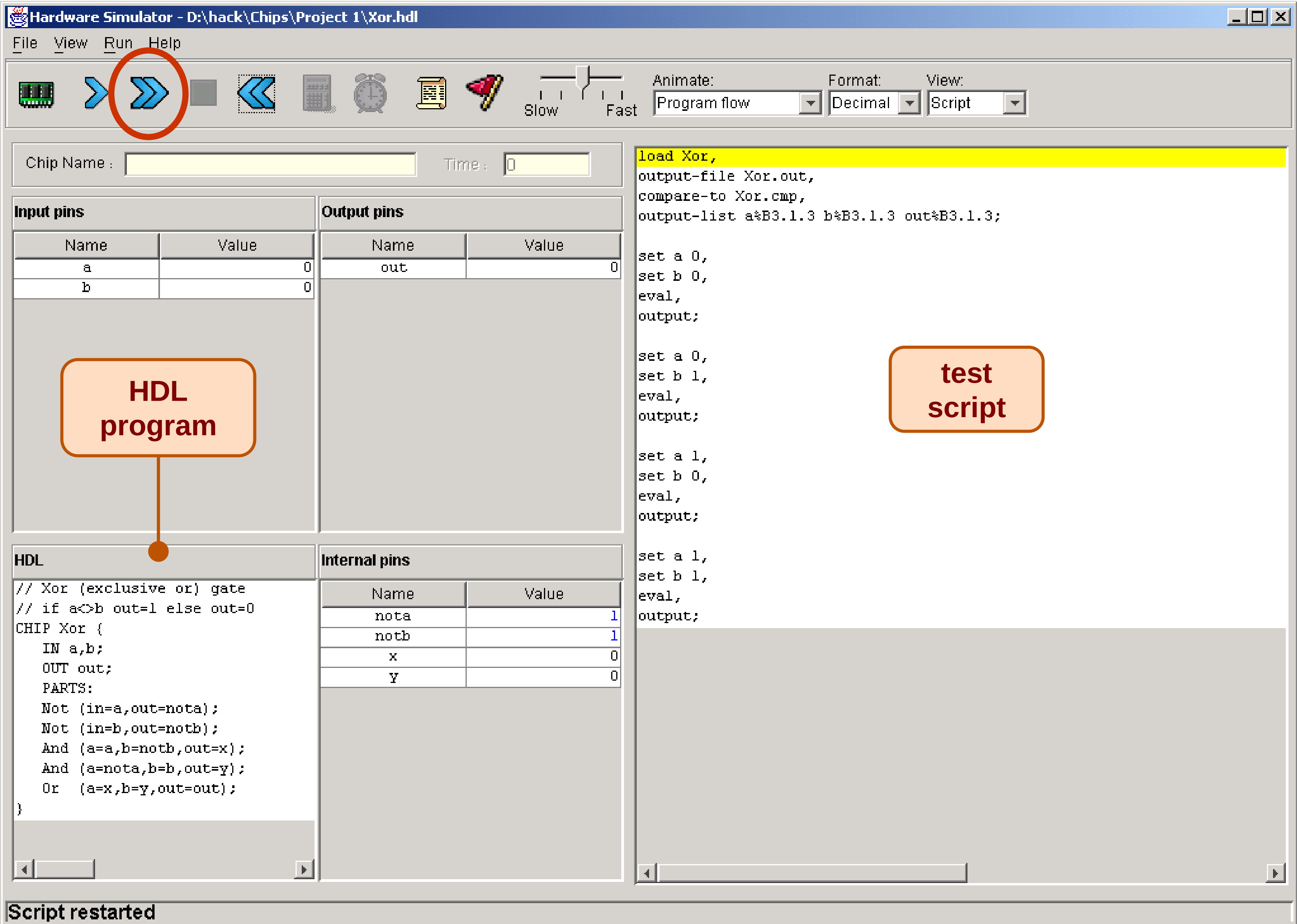
Implementation: And(a,b) = Not(Nand(a,b))



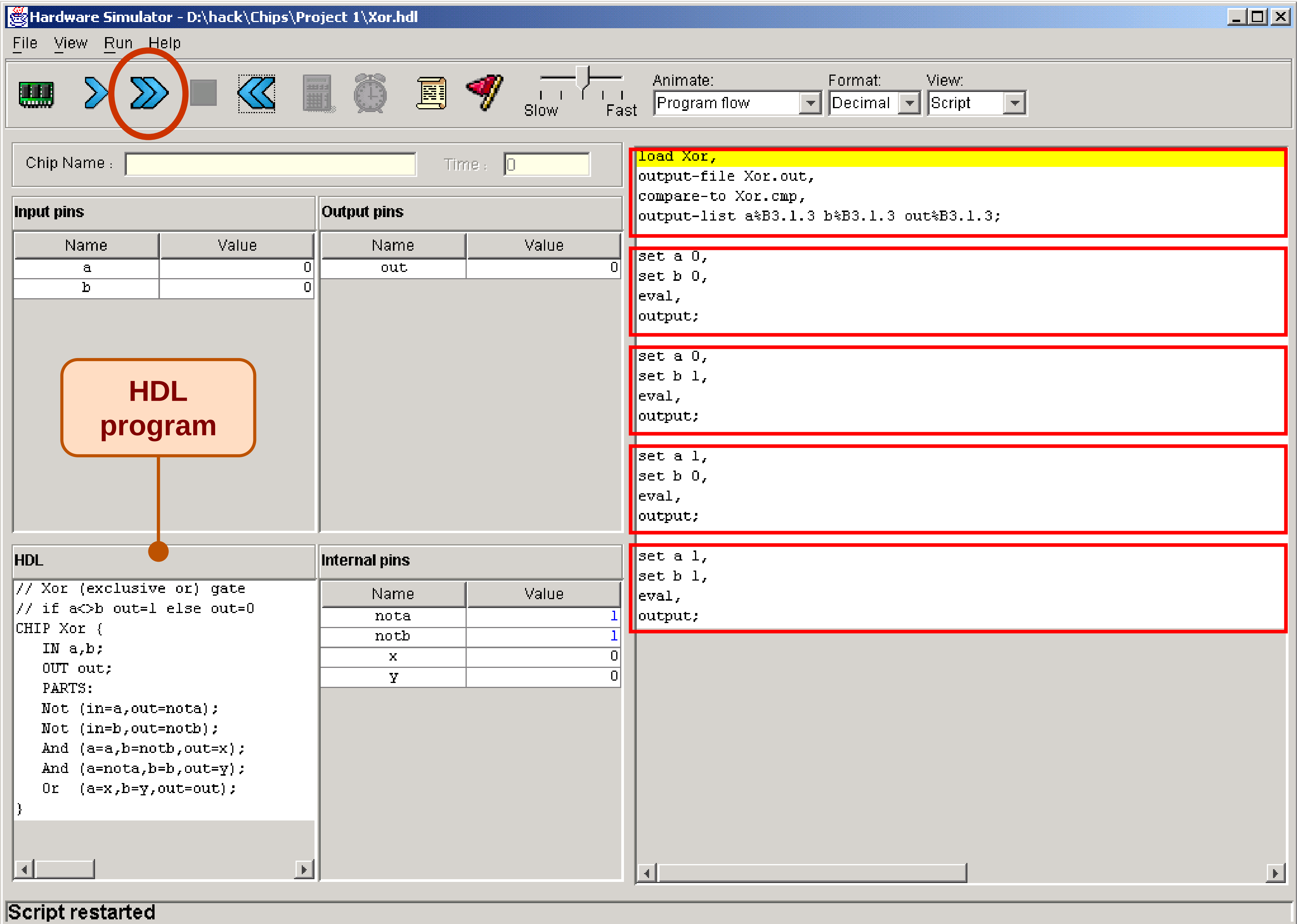
And.hdl

```
CHIP And
{   IN  a, b;
    OUT out;
    Nand(a = a,
         b = b,
         out = x);
    Not(in = x, out = out)
}
```

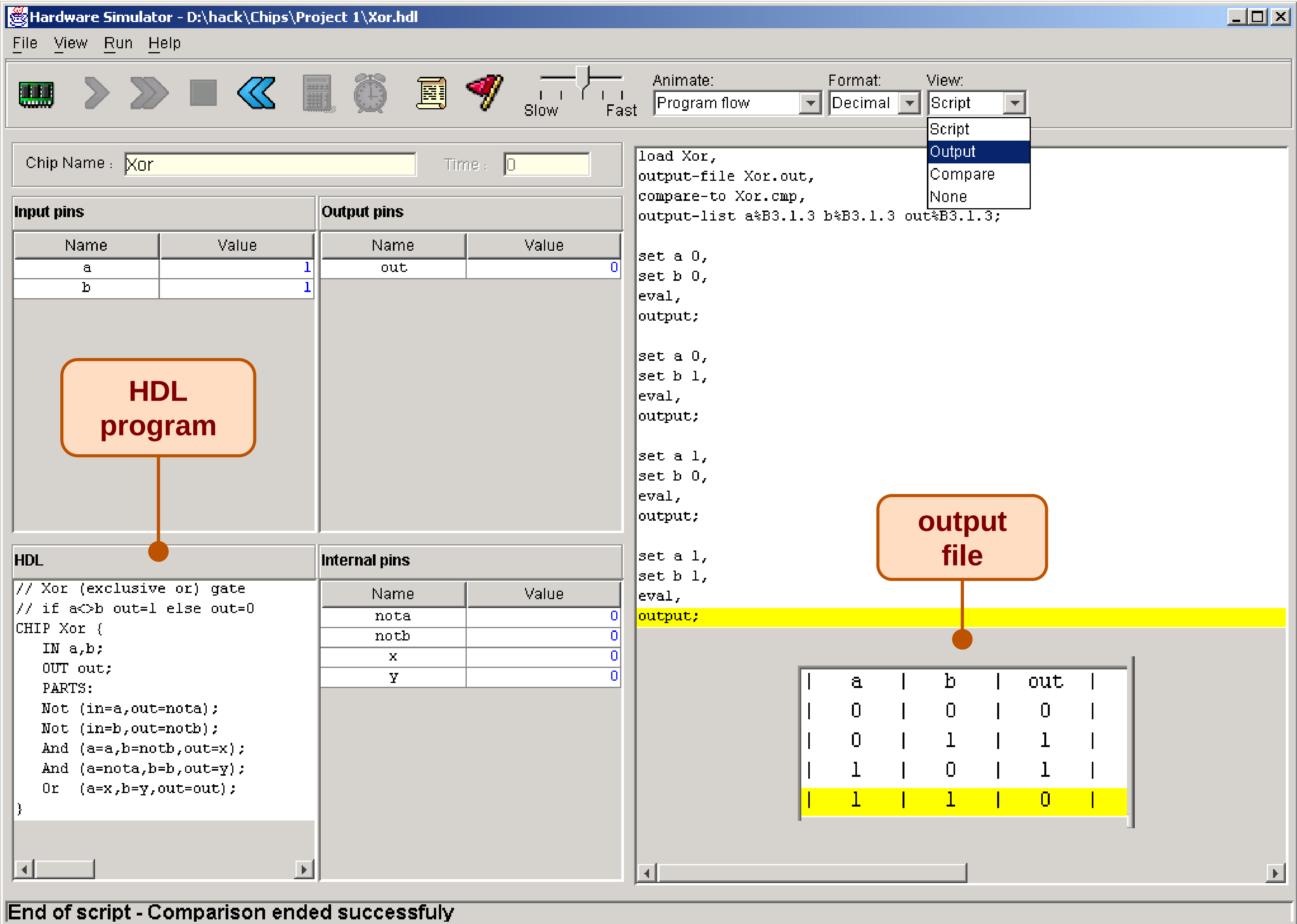
Hardware simulator (demonstrating Xor gate construction)



Hardware simulator



Hardware simulator



Project materials: www.nand2tetris.org

From NAND to Tetris

Building a Modern Computer From First Principles

- Home
- Projects
- Book
- Software
- Media
- Cool Stuff
- Terms
- Q&A
- About

Project 1: Logic Gates

Project 1 web site

Background

A typical computer architecture is based on a set of elementary logic gates like `And`, `Or`, etc., as well as their bit-wise versions `And16`, `Or16`, etc. (in a 16-bit machine). This project engages you in the construction of a typical set of elementary gates. These gates form the elementary building blocks from which more complex chips will be later constructed.

Objective

Build all the logic gates described in Chapter 1 (see list below), yielding a basic chip-set. The only building blocks that you can use in this project are primitive `Nand` gates and the composite gates that you will gradually build on top of them.

Chips

Chip (HDL)	Function	Test Script	Compare File
Nand	Nand gate (primitive)		
Not	Not gate	Not.tst	Not.cmp
And	And gate	And.tst	And.cmp
Or	Or gate	Or.tst	Or.cmp
Xor	Xor gate	Xor.tst	Xor.cmp
Mux	Mux gate	Mux.tst	Mux.cmp
DMux	DMux gate	DMux.tst	DMux.cmp
Not16	16-bit Not	Not16.tst	Not16.cmp

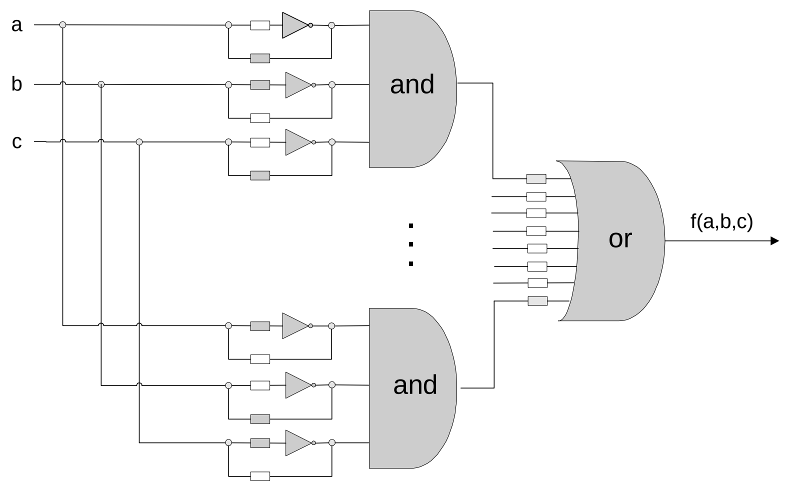
And.hdl ,
And.tst ,
And.cmp files

Project 1 tips

- Read the Introduction + Chapter 1 of the book
- Download the book’s software suite
- Go through the hardware simulator tutorial
- Do Project 0 (optional)
- You’re in business.

Perspective

- Each Boolean function has a canonical representation
- The canonical representation is expressed in terms of And, Not, Or
- And, Not, Or can be expressed in terms of Nand alone
- Ergo, every Boolean function can be realized by a standard PLD consisting of Nand gates only
- Mass production
- Universal building blocks, unique topology
- Gates, neurons, atoms, …



End notes: Canonical representation

Whodunit story: Each suspect may or may not have an alibi (*a*), a motivation to commit the crime (*m*), and a relationship to the weapon found in the scene of the crime (*w*). The police decides to focus attention only on suspects for whom the proposition Not(*a*) And (*m* Or *w*) is true.

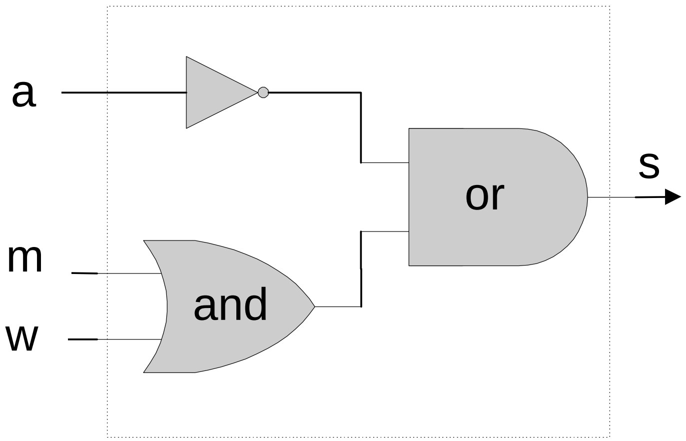
Truth table of the "suspect" function $s(a,m,w) = \bar{a} \cdot (m + w)$

a	m	w	*minterm*	suspect(a,m,w)= not(a) and (m or w)
0	0	0	$m_0 = \bar{a}\,\bar{m}\,\bar{w}$	0
0	0	1	$m_1 = \bar{a}\,\bar{m}\,w$	1
0	1	0	$m_2 = \bar{a}\,m\,\bar{w}$	1
0	1	1	$m_3 = \bar{a}\,m\,w$	1
1	0	0	$m_4 = a\,\bar{m}\,\bar{w}$	0
1	0	1	$m_5 = a\,\bar{m}\,w$	0
1	1	0	$m_6 = a\,m\,\bar{w}$	0
1	1	1	$m_7 = a\,m\,w$	0

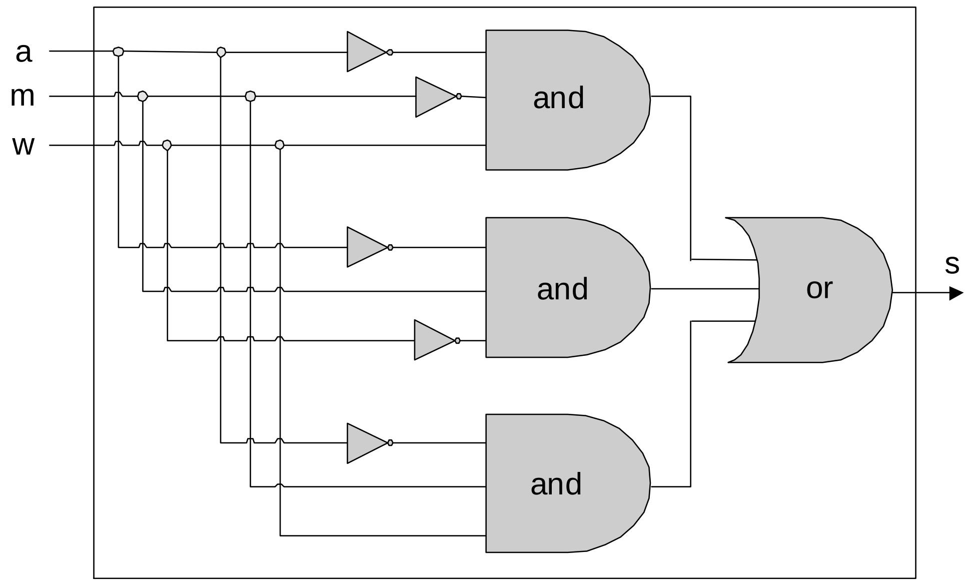
Canonical form: $s(a,m,w) = \bar{a}\,\bar{m}\,w + \bar{a}\,m\,\bar{w} + \bar{a}\,m\,w$

End notes: Canonical representation (cont.)

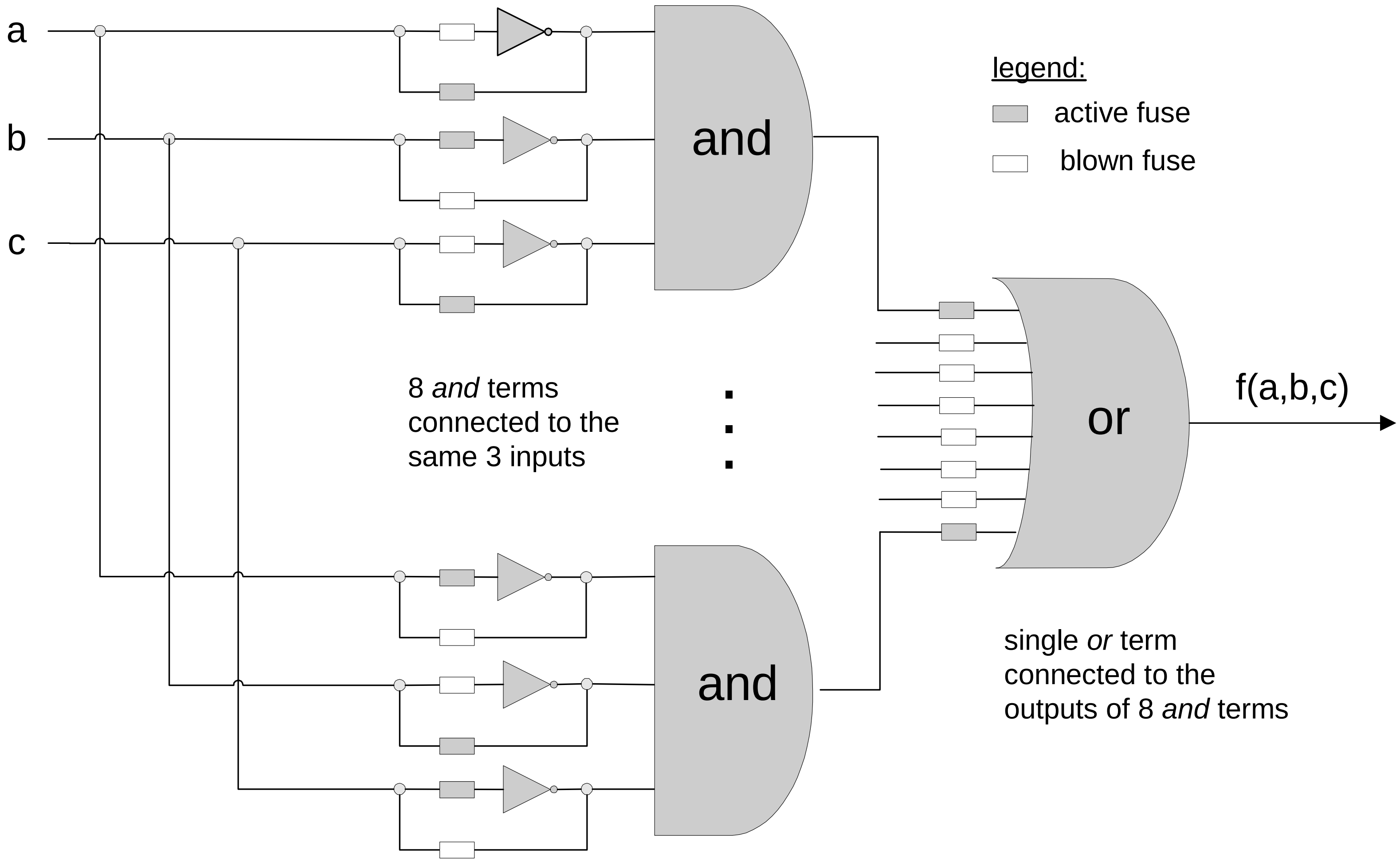
$$s(a,m,w) = \bar{a} \cdot (m + w)$$



$$s(a,m,w) = \bar{a}\,\bar{m}\,w + \bar{a}\,m\,\bar{w} + \bar{a}\,m\,w$$



End notes: Programmable Logic Device for 3-way functions



PLD implementation of $f(a,b,c)= a\,\overline{b}\,c + \overline{a}\,b\,\overline{c}$

(the on/off states of the fuses determine which gates participate in the computation)

End notes: universal building blocks, unique topology