Boolean Logic

Building a Modern Computer From First Principles

www.nand2tetris.org
Boolean algebra

Some elementary Boolean functions:

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

Boolean functions:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>( f(x, y, z) = (x + y) \overline{z} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A Boolean function can be expressed using a functional expression or a truth table expression.

Important observation:
Every Boolean function can be expressed using And, Or, Not.
### All Boolean functions of 2 variables

<table>
<thead>
<tr>
<th>Function</th>
<th>$x$</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$y$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Constant 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>And</td>
<td>$x \cdot y$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$x$ And Not $y$</td>
<td>$x \cdot \overline{y}$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$x$</td>
<td>$x$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Not $x$ And $y$</td>
<td>$\overline{x} \cdot y$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$y$</td>
<td>$y$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Xor</td>
<td>$x \cdot \overline{y} + \overline{x} \cdot y$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Or</td>
<td>$x + y$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Nor</td>
<td>$\overline{x + y}$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Equivalence</td>
<td>$x \cdot y + \overline{x} \cdot \overline{y}$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Not $y$</td>
<td>$\overline{y}$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>If $y$ then $x$</td>
<td>$x + \overline{y}$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Not $x$</td>
<td>$\overline{x}$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>If $x$ then $y$</td>
<td>$\overline{x} + y$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Nand</td>
<td>$x \cdot y$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Constant 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean algebra

**Given:** \( \text{Nand}(a,b), \text{false} \)

**We can build:**

- \( \text{Not}(a) = \text{Nand}(a,a) \)
- \( \text{true} = \text{Not}(\text{false}) \)
- \( \text{And}(a,b) = \text{Not}(\text{Nand}(a,b)) \)
- \( \text{Or}(a,b) = \text{Not}(\text{And}(\text{Not}(a),\text{Not}(b))) \)
- \( \text{Xor}(a,b) = \text{Or}(\text{And}(a,\text{Not}(b)),\text{And}(\text{Not}(a),b)) \)
- 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:**

  **Gate interface**

  ![Gate Interface Diagram]

  If \(a \land b \land c = 1\) then \(\text{out} = 1\) else \(\text{out} = 0\)

  **Gate implementation**

  ![Gate Implementation Diagram]

- **Important distinction:** Interface *(what)* VS implementation *(how).*
Gate logic

Interface

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>out</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

**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.
Proposed Implementation: based on Not, And, Or gates.
Example: Building an **And** gate

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

**Example: Building an And gate**

**And.hdl**

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

**And.tst**

```plaintext
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;
```

---

**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:** \( \text{And}(a,b) = 1 \) exactly when \( a=b=1 \)

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

And.hdl

```
CHIP And
{  IN  a, b;
    OUT out;
    // implementation missing
}
```
Building an **And** gate

Implementation: \( And(a,b) = \text{Not}(\text{Nand}(a,b)) \)

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

And.hdl
Building an And gate

Implementation: \( \text{And}(a,b) = \text{Not}(\text{Nand}(a,b)) \)

\[
\begin{array}{c}
\text{a} \\
\text{b}
\end{array} \rightarrow \begin{array}{c}
\text{Nand} \\
\text{Not}
\end{array} \rightarrow \begin{array}{c}
\text{out}
\end{array} \rightarrow \begin{array}{c}
\text{out}
\end{array}
\]

And.hdl

```
CHIP And
{
    IN a, b;
    OUT out;
    // implementation missing
}
```
Building an **And** gate

Implementation: \( \text{And}(a, b) = \text{Not}(\text{Nand}(a, b)) \)

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

```haskell
// Xor (exclusive or) gate
// if a=0 b=1 then out=1, else out=0
CHIP Xor |
  ID a,b;  
  OUT out;  
  ENTS:
    Not (ins,out=not);  
    Not (in=b,out=notb);  
    And (a=b,notb,out=x);  
    And (a=notb,b,out=y);  
    Or (a=x,b=y,out=out);  

// HDL program
// Load Xor,  
// output-file Xor.out,  
// compare-to Xor.exp,  
// output-list eqB3.1.3 b=B3.1.3 out=B3.1.3;
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;

Script restarted
```
Hardware simulator

```
// Xor {exclusive or| gate
// if a<>b output=1 else output=0
CHIP Xor |
| a,b: |
OUT: out;
PARTS:
Not (ins=x, out=notx);
Not (ins=y, out=noty);
and (a=x, b=x, out=x);
and (a=x, b=y, out=x);
OR (a=x, b=x, out=x);
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;
```

End of script - Comparison ended successfully

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Project 1: Logic Gates

| 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 are the elementary building blocks from which 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 |

<table>
<thead>
<tr>
<th>Chip (HDL)</th>
<th>Function</th>
<th>Test Script</th>
<th>Compare File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nand</td>
<td>Nand gate</td>
<td>Nand.tst</td>
<td>Nand.cmp</td>
</tr>
<tr>
<td>Not</td>
<td>Not gate</td>
<td>Not.tst</td>
<td>Not.cmp</td>
</tr>
<tr>
<td>And</td>
<td>And gate</td>
<td>And.tst</td>
<td>And.cmp</td>
</tr>
<tr>
<td>Or</td>
<td>Or gate</td>
<td>Or.tst</td>
<td>Or.cmp</td>
</tr>
<tr>
<td>Xor</td>
<td>Xor gate</td>
<td>Xor.tst</td>
<td>Xor.cmp</td>
</tr>
<tr>
<td>Mux</td>
<td>Mux gate</td>
<td>Mux.tst</td>
<td>Mux.cmp</td>
</tr>
<tr>
<td>DMux</td>
<td>DMux gate</td>
<td>DMux.tst</td>
<td>DMux.cmp</td>
</tr>
<tr>
<td>Not16</td>
<td>16-bit Not</td>
<td>Not16.tst</td>
<td>Not16.cmp</td>
</tr>
</tbody>
</table>

And.tst, 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 $\text{Not}(a) \ And \ (m \ Or \ w)$ is true.

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

<table>
<thead>
<tr>
<th>$a$</th>
<th>$m$</th>
<th>$w$</th>
<th>minterm $m_i$</th>
<th>$\text{suspect}(a, m, w) = \text{not}(a)$ and $(m \ or \ w)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$m_0 = \overline{a} \overline{m} \overline{w}$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$m_1 = \overline{a} \overline{m} w$</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$m_2 = \overline{a} m \overline{w}$</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$m_3 = \overline{a} m w$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$m_4 = a \overline{m} \overline{w}$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$m_5 = a \overline{m} w$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$m_6 = a m \overline{w}$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$m_7 = a m w$</td>
<td>0</td>
</tr>
</tbody>
</table>

**Canonical form:** $s(a, m, w) = \overline{a} \overline{m} w + \overline{a} m \overline{w} + \overline{a} m w$
End notes: Canonical representation (cont.)

\[ s(a, m, w) = \overline{a} \cdot (m + w) \]

\[ s(a, m, w) = \overline{a}m + \overline{a}m\overline{w} + \overline{a}mw \]
End notes: Programmable Logic Device for 3-way functions

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

(the on/off states of the fuses determine which gates participate in the computation)
End notes: universal building blocks, unique topology

![Diagram of logic gates](image1)

![Diagram of artificial neuron](image2)