Boolean Logic

Friday, January 21, 2022 8:26 PM

"Such Simple things, and we make them something so complex it defeats us, Almost."
- John Ashbery (1927-2017)

Every computer is made up of chips (no matter how complex it is). The job of these chips is to process binary data (0 or 1)

These chips are made up of *logic gates.*

This chapter will focus on one primitive logic gate Nand (NOT, AND, OR, XOR)

1.1 Boolean algebra

Boolean algebra is used to manipulate two-state binary values

Binary is denoted as -

- True/false
- 1/0
- Yes/no
- On/off

Boolean functions take in binary inputs and also output binary

Boolean operators:

Operation	1	2
And	$x \cdot y$	$x \wedge y$
Or	$x + y$	$x \vee y$
Not	$\bar{x}$	$\neg x$

$Nand - N(ot)\ And$

$Nand(x, y) = Not\big(And(x, y)\big)$

Boolean Functions:

Types of Boolean Functions:

	$x =$	0	0	1	1
	$y =$	0	1	0	1
	Notation				
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

One way to define a Boolean function is to use a truth table.

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:

Given any Truth Table you can derive the Boolean function from it (Proof in Appendix 1).

The ability to move from one expression to the other is one of the most fundamental practices of hardware design

The ability to optimize Boolean expressions is the first step to optimizing hardware.

Logic Gates:

A gate is a physical device that represents a simple Boolean function.

There have been many ways engineers have implemented Boolean expressions such as magnetic, optical, biological, pneumatic, hydraulic, quantum-based, and even domino-based mechanisms.

The most common (in computing) is the electric representation. Transistors etched into silicon, packaged as *chips*.

Computer Scientists aren't worried about the physical representation of these Boolean Algebra concepts. Most are fine with the abstract concept of them and leave the hardware to the physicists and electrical engineers.

Primitive and Composite Gates:

Since all logic gates have the same input and output type (0's or 1's) they can be combined in any fashion or complexity to achieve new gates.

If asked to construct the Boolean function $And(a, b, c)$ we can see using Boolean Algebra that

$$a \cdot b \cdot c = (a \cdot b) \cdot c$$

Is equivalent and thus the representation of both will produce the same output.

Now consider $Xor(a, b)$

Xor is one only when either $a = 1 \; and \; b = 0 \; or \; a = 0 \; and \; b = 1$

Note that each individual interface is unique, there is only one way to specify it. This is usually done via a truth table, Boolean expression or a verbal expression.

Each individual interface can be realized in multiple ways, and some will be more efficient and/or elegant.

Basically there are multiple ways to write a Boolean expression or draw a gate representation of that expression, but the gate/expression will always have the same outputs for its corresponding inputs.

Hardware Construction:

This section goes into the garage approach to designing circuits.

Imagine you need to build 100 Xor gates. You had to use 3 (of each) - Not, And, and Or gates to fabricate it. You could impliment the circuit and then cover it up except for the two input and one output pin. Next you can label them all Xor and put them in a box labeled Xor for future use as a black-box gate.

This is great and holds a lot to be desired, except not applicable for real life use. The circuits in production today are much more complicated and this method could likely cause for errors as you may have made a mistake with the production of one of the hundred gates. If that is the issue then you would have to trouble shoot the individual black-box circuit (which is often a lot harder than it is worth it).

Hardware Description Language:

HDL is a language that Hardware Designers use to design hardware on a computer with circuits.

A special *Hardware Simulation* tool is then used to take the HDL code as input and put it through rigorous tests to see if the implementation was a success.

The hardware designer will also look at certain specifications that are also calculated such as speed of computation, energy consumption, and the overall cost implied by the proposed chip implementation,

The designer will then alter the chip to fit their (or the customers) desired cost and performance levels.

HDL can thus be used to plan, debug, and optimize an entire chip before a single penny is spent on physical production!

The HDL code (once the chip has passed all the checks) can then be used as a blueprint to print the chip into silicon.

Example: Building an Xor Gate:

HDL consists of two sections, a *header* section and a *parts* section.

The header section contains the name of the chip and the names of its input and output pins

The parts section describes the chips parts from which the chip is made.

Sample HDL code

```
/* Xor (exclusive or) gate:
   if a!=b out=1 else out=0. */
Chip Xor {
     IN a, b;
     OUT out;
```

```
    PARTS:
    Not (in=a  out=nota);
    Not (in=b, out=notb);
    And (a=a,    b=notb, out=w1);
    And (a=nota, b=b,    out=w2);
    Or  (a=w1,   b=w2,   out=out);
}
```

The API's of all the chips used in Nand to Tetris are specified in Appendix 4

Firstly, internal pins are created "automatically" in code the first time they appear i.e. "nota", "notb" etc.

Secondly, in circuit building, multiple forks are allowed. In Circuit diagrams, this is specified by drawing multiple connections, while in code this is just inferred.

- For example the input "a" is found to go into both
  - Not (in=a,…);
  - And (a=a,…);
- These connections are inferred to be a fork as a is going into two separate gates.

Testing:

Testing these HDL designed chips is mandated to be very rigorous and thorough.

Sample test code:

```
load Xor.hdl
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;
```

Sample Output

```
 a | b | out
-------------
 0 | 0 |  0
 0 | 1 |  1
 1 | 0 |  1
 1 | 1 |  0
```

Hardware Simulation:

Writing HDL code is similar to writing code for software. Instead of compiling and running the software code you only need to test the HDL code.

Specification:

Nand:

The starting point of our computer architecture is the Nand gate From which all other chips will be built. The Nand gate realizes the following Boolean Function:

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

Using API Style

```
Chip name: Nand
Input:     a, b
Output:    out
Function:  if ((a==1) and (b==1)) then out = 0, else out = 1
```

a	b	AND	NAND
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

Basic Logic Gates

Not:

```
Chip name: Not
Input:     in
Output     out
Function:  if (in==0) then out = 1, else out = 0
```

```
Not = Nand(a, a) 1 1 0 0
```

1
1
0
0

And:

```
Chip name: And
Input:     a, b
Output:    out
Function:  if ((a==1) and (b==1)) then out = 1, else out = 0
```

```
And = And(a, b)
    = Not( Nand(a, b) )
    = Nand(          0 0 0 1
        Nand(a, b), 1 1 1 0
        Nand(a, b)  1 1 1 0
      )
```

0
0
0
1

Or:

```
Chip name: Or
Input:     a, b
Output:    out
Function:  if ((a=0) and (b=0)) then out = 0, else out = 1
```

```
Or = Or(a, b)
   = Not( And( Not(a), Not(b) )
   = Nand(          0 1 1 1
       Nand(a, a), 1 1 0 0
       Nand(b, b)  1 0 1 0
     )
```

0
1
1
1

Xor:

```
Chip name: Xor
Input:     a, b
Output:    out
Function:  if (a!=b) then out = 1, else out = 0
```

Xor = Xor(a, b) = Or(And(Not(a), b), And(a, Not(b))) = Nand(0 1 1 0 Nand(Nand(a, a), b), 1 0 1 1 Nand(a, Nand(b, b)) 1 1 0 1)
0
1
1
0

Multiplexer (Mux):

Three input gate, where two (a, b) are the input bits, and the other (sel) is the selection bit. The sel bit is used to select either input a or b to be the output.

Demultiplexer (Demux):

Performs the opposite function of a mux. Takes in a single bit and routes it to the 2 output pins based on where the sel tells it too.

Multi-Bit Version of Basic Gates

For our computer design we will be dealing with the 16-bit version of these gates. Although the number of bits (n) does not specifically matter (e.g., 16, 32, or 64 bits).

HDL programs treat multi-bit values as single-bit values. Except for the fact that individual values can be indexed to get individual bits.

For example:

16-bit gate has a 16-bit in and out

out[3] = in[5]

Will set the third bit of the out value to the fifth bit of the in value.

Multi-Bit Logic Gates:

Multi-bit Not:

```
Chip name: Not16
Input:     in[16]
Output     out[16]
Function:  for i = 0..15 out[i] = Not(in[i])
```

Multi-bit And:

```
Chip name: And16
Input:     a[16], b[16]
Output:    out[16]
Function:  for i = 0..15 out[i] = And(a[i], b[i])
```

Multi-bit Or:

```
Chip name: Or16
Input:     a[16], b[16]
Output:    out[16]
Function:  for i = 0..15 out[i] = Or(a[i], b[i])
```

Multi-bit Multiplexer:

```
Chip name: Mux16
Input:     a[16], b[16], sel
Output:    out[16]
Function:  if (sel==0) the for i = 0..15 out[i] = a[i],
           else for I = 0..15 out[i] = b[i]
```

Multi-Way Versions of basic Gates:

```
Chip name: Or8Way
Input:     in[8]
Output:    out
Function:  out = Or(in[0], in[1],… , in[7])

Chip name: Mux4Way16
Input:     a[16], b[16], c[16], d[16], sel[2]
Output:    out[16]
Function:  if (sel 00, 01, 10, 11) then out = a, b, c, d
```

```
Chip name: Mux8Way16
Input:     a[16], b[16], c[16], d[16], e[16], f[16], g[16], h[16], sel[3]
Output:    out[16]
Function:  if (sel 000, 001, 010, 011, 100, 101, 110, 111)
           then out = a, b, c, d, e, f, g, h
```

Multi-Way/Multi-bit Demux

Sel[1]	Sel[0]	a	b	c	d
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	In

```
Chip name: Dmux4Way
Input:     in, sel[2]
Output:    a, b, c, d
Function:  if (sel==00)      then {a, b, c, d} = {1, 0, 0, 0},
           else if (sel==01) then {a, b, c, d} = {0, 1, 0 ,0},
           else if (sel==10) then {a, b, c, d} = {0, 0, 1, 0},
           else if (sel==11) then {a, b, c, d} = {0, 0, 0, 1}
```

```
Chip name: Dmux8Way
Input:     in, sel[3]
Output:    a, b, c, d, e, f, g, h
Function:  if (sel==000)      then {a, b, c, …, h} = {1, 0, 0, …, 0},
           else if (sel==001) then {a, b, c, …, h} = {0, 1, 0, …, 0},
           else if (sel==010) then {a, b, c, …, h} = {0, 0, 1, …, 0},
           …
           else if (sel==111) then {a, b, c, …, h} = {0, 0, 0, …, 1}
```