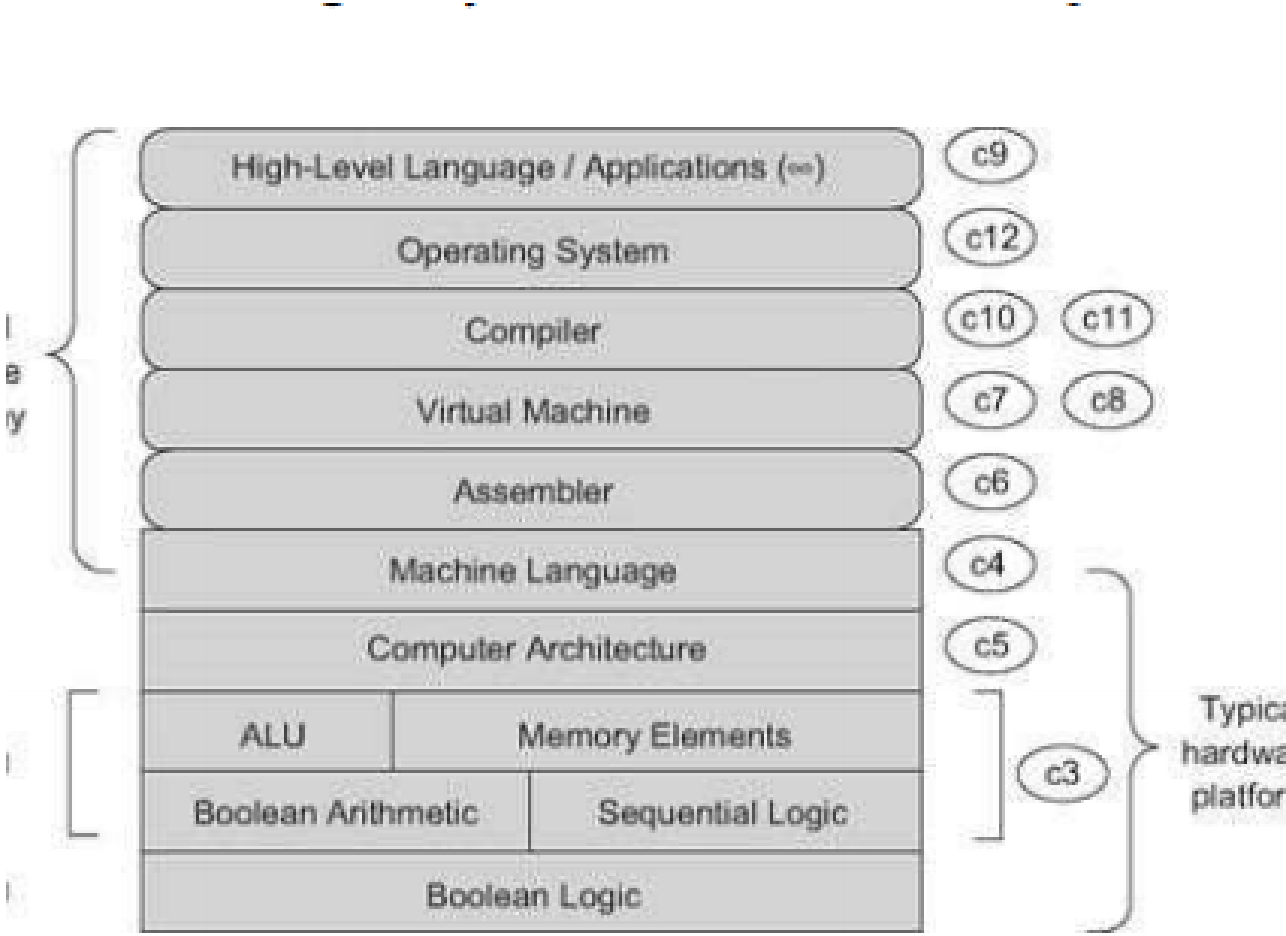
コンピュータシステムの理論と実装

もくじ

1. Intrduction
2. 使い方
3. ユースケース

1. Intrduction

- コース概要
  - https://www.nand2tetris.org
  - course https://www.coursera.org/learn/build-a-computer/lecture/tfRns/unit-0-0-introduction
  - ted talk
  - moden pcを自分で一から作ってみようといコンセプト



Intrduction

- 最近話題のライセンス
  - https://www.nand2tetris.org/licens

背景

- バソコンは全て情報の保存と処理を行うために設計された **論理ゲート(logic gate)** から構成されている。
- 本章では, **ブールゲート(boolean gate)** と呼ばれる最も単純なゲートに焦点をあてる。
- ブールゲートは **ブール関数(boolean function)** を物理的に実現したもんであるため、ブール関数についてみます。

ブール代数

- ブール代数はブール値を扱う(0, 1)のみ
- ブール関数はブール値を受け取り、ブール値を返す関数
- ブール関数を表現する方法
  - 真理値表(truth table)
  - ブール式

真理値表(truth table)

- 全ての入力に対数る関数の出力を全て列挙する
- 変数の組み合わせは 2^3 通りある
- 真理値表から **ブール式(boolean expression)** を導くことができる

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

Figure 1.1 Truth table representation of a Boolean function (example).

ブール式(boolean expression)

- ブール式の基本は And, Or, Notの3である

Boolean Operations

x And y

$x \wedge x$

x	y	And
0	0	0
0	1	0
1	0	0
1	1	1

x Or y

$x \vee y$

x	y	Or
0	0	0
0	1	1
1	0	1
1	1	1

Not(x)

$\neg x$

x	Not
0	1
1	0

ブール式(boolean expression)　例

Boolean Expressions

Not(0 Or (1 And 1)) =

Not(0 Or 1) =

Not(1) =

0

ブール式(boolean expression)

- $f(x, y, z) = (x + y) * \tilde{z}$

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

Figure 1.1 Truth table representation of a Boolean function (example).

ブール式(boolean expression)

Boolean Identities

- $(x \text{ And } y) = (y \text{ And } x)$
- $(x \text{ Or } y) = (y \text{ Or } x)$

commutative laws

- $(x \text{ And } (y \text{ And } z)) = ((x \text{ And } y) \text{ And } z)$
- $(x \text{ Or } (y \text{ Or } z)) = ((x \text{ Or } y) \text{ Or } z)$

associative laws

- $(x \text{ And } (y \text{ Or } z)) = (x \text{ And } y) \text{ Or } (x \text{ And } z)$
- $(x \text{ Or } (y \text{ And } z)) = (x \text{ Or } y) \text{ And } (x \text{ Or } z)$

distributive laws

- $\text{Not}(x \text{ And } y) = \text{Not}(x) \text{ Or } \text{Not}(y)$
- $\text{Not}(x \text{ Or } y) = \text{Not}(x) \text{ And } \text{Not}(y)$

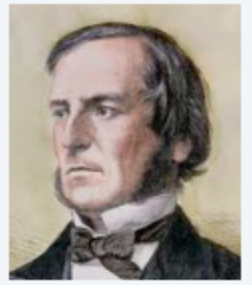
De Morgan laws

ブール式(boolean expression)

Boolean algebra

Developed by George Boole in 1840s to study logic problems
- Variables represent *true* or *false* (1 or 0 for short).
- Basic operations are AND, OR, and NOT (see table below).

Widely used in mathematics, logic and computer science.



George Boole
1815–1864

operation	*Java notation*	*logic notation*	*circuit design (this lecture)*
AND	x && y	$x \wedge y$	xy
OR	x || y	$x \vee y$	$x + y$
NOT	! x	$\neg x$	x'

← various notations in common use

Example: (stay tuned for proof)

DeMorgan's Laws

$$(xy)' = (x' + y')$$
$$(x + y)' = x'y'$$

Relevance to circuits. Basis for next level of abstraction.

Copyright 2004, Sidney Harris
http://www.sciencecartoonsplus.com

14

ブール式(boolean expression)

Truth table proofs

Truth tables are convenient for establishing identities in Boolean logic

- One row for each possibility.
- Identity established if columns match.

Proofs of DeMorgan's laws

x	y	xy	$(xy)'$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

x	y	x'	y'	$x' + y'$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

$(xy)' = (x' + y')$

x	y	$x + y$	NOR $(x + y)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

x	y	x'	y'	NOR $x'y'$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

$(x + y)' = x'y'$

16

ブール式(boolean expression)

Boolean Algebra

Not(Not(x) And Not(x Or y)) = — De Morgan law

Not(Not(x) And (Not(x) And Not(y))) = — associative law

Not((Not(x) And Not(x)) And Not(y)) = — idempotence

Not(Not(x) And Not(y)) = — De Morgan law

Not(Not(x)) Or Not(Not(y)) = — double negation

x Or y

雑談 shanoおじさん

A basis for digital devices

Claude Shannon connected *circuit design* with Boolean algebra in 1937.

> *" Possibly the most important, and also the most famous, master's thesis of the [20th]*
>
> *– Howard Gardner*

Key idea. Can use Boolean algebra to systematically analyze circuit behavior.

A Symbolic Analysis of Relay and Switching Circuits

By CLAUDE E. SHANNON

I. Introduction

IN THE CONTROL and protective circuits of complex electrical systems it is frequently necessary to make intricate interconnections of relay contacts and switches. Examples of these circuits occur in automatic telephone exchanges, industrial motor-control equipment, and in almost any circuits designed to perform complex operations automatically. In this paper a mathematical analysis of certain of the properties of such networks will be made. Particular attention will be given to the problem of network synthesis. Given certain characteristics, it is required to find a circuit incorporating these characteristics. The solution of this type of problem is not unique and methods of finding those particular circuits requiring the least number of relay contacts and switch blades will be studied. Methods will also be described for finding any number of circuits equivalent to a given circuit in all operating characteristics. It will be shown that several of the well-known theorems on impedance networks have roughly analogous theorems in relay circuits. Notable among these are the delta-wye and star-mesh transformations, and the duality theorem.

The method of attack on these problems may be described briefly as follows: any circuit is represented by a set of equations, the terms of the equations corresponding to the various relays and switches in the circuit. A calculus is developed for manipulating these equations by simple mathematical processes, most of which are similar to ordinary algebraic algorisms. This calculus is shown to be exactly analogous to the calculus of propositions used in the symbolic study of logic. For the synthesis problem the desired characteristics are first written as a system of equations, and the equations are then manipulated into the form representing the simplest circuit. The circuit may then be immediately drawn from the equations. By this method it is always possible to find the simplest circuit containing only series and parallel connections, and in some cases the simplest circuit containing any type of connection.

Our notation is taken chiefly from symbolic logic. Of the many systems in common use we have chosen the one which seems simplest and most suggestive for our interpretation. Some of our phraseology, as node, mesh, delta, wye, etc., is borrowed from ordinary network theory for similar concepts in switching circuits.

II. Series-Parallel Two-Terminal Circuits

Fundamental Definitions and Postulates

We shall limit our treatment to circuits containing only relay contacts and switches, and therefore at any given time the circuit between any two terminals must be either open (infinite impedance) or closed (zero impedance). Let us associate a symbol X_{ab} or more simply X, with the terminals a and b. This variable, a function of time, will be called the *hindrance* of the two-terminal circuit a-b. The symbol 0 (zero) will be used to represent the hindrance of a closed circuit, and the symbol 1 (unity) to represent the hindrance of an open circuit. Thus when the circuit a-b is open $X_{ab} = 1$ and when closed $X_{ab} = 0$. Two hindrances X_{ab} and X_{ab} will be said to be equal if whenever the circuit a-b is open, the circuit c-d is open, and whenever a-b is closed, c-d is closed. Now let the symbol + (plus) be defined to mean the series connection of the two-terminal circuits whose hindrances are added together. Thus $X_{ab} + X_{cd}$ is the hindrance of the circuit a-d when b and c are connected together. Similarly the product of two hindrances $X_{ab} \cdot X_{cd}$ or more briefly $X_{ab}X_{cd}$ will be defined to mean the hindrance of the circuit formed by connecting the circuits a-b and c-d in parallel. A relay contact or switch will be represented in a circuit by the symbol in figure 1, the letter being the corresponding hindrance function. Figure 2 shows the interpretation of the plus sign and figure 3 the multiplication sign. This choice of symbols makes the manipulation of hindrances very similar to ordinary numerical algebra.

It is evident that with the above definitions the following postulates will hold:

Postulates

1. a. $0 \cdot 0 = 0$ — A closed circuit in parallel with a closed circuit is a closed circuit.
   b. $1 + 1 = 1$ — An open circuit in series with an open circuit is an open circuit.
2. a. $1 + 0 = 0 + 1 = 1$ — An open circuit in series with a closed circuit in either order (i.e., whether the open circuit is to the right or left of the closed circuit) is an open circuit.
   b. $0 \cdot 1 = 1 \cdot 0 = 0$ — A closed circuit in parallel with an open circuit in either order is a closed circuit.
3. a. $0 + 0 = 0$ — A closed circuit in series with a closed circuit is a closed circuit.
   b. $1 \cdot 1 = 1$ — An open circuit in parallel with an open circuit is an open circuit.
4. At any given time either $X = 0$ or $X = 1$.

These are sufficient to develop all the theorems which will be used in connection with circuits containing only series and parallel connections. The postulates are arranged in pairs to emphasize a duality relationship between the operations of addition and multiplication and the quantities zero and one. That, if in any of the a postulates the zero's are replaced by one's and the multiplications by additions and vice versa, the corresponding b postulate will result. This fact is of great importance. It gives each theorem a dual theorem, it being necessary to prove only one to establish both. The only one of these postulates which differs from ordinary algebra is 1b. However, this enables great simplifications in the manipulation of these symbols.

Paper number 38-80, recommended by the AIEE committees on communication and basic sciences and presented at the AIEE summer convention, Washington, D. C., June 20-24, 1938. Manuscript submitted March 1, 1938; made available for preprinting May 19, 1938.

CLAUDE E. SHANNON is a research assistant in the department of electrical engineering at Massachusetts Institute of Technology, Cambridge. This paper is an abstract of a thesis presented at MIT for the degree of master of science. The author is indebted to Doctor F. L. Hitchcock, Doctor Vannevar Bush, and Doctor S. H. Caldwell, all of MIT, for helpful encouragement and criticism.

22

Claude Shannon
1916–2001

ブール式(boolean expression)

Boolean expression ➔ truth table

f(*x*, *y*, *z*) = (*x* And *y*) Or (Not(*x*) And *z*)

x	*y*	*z*	*f*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

ブール式(boolean expression)

Boolean expression ← truth table

f(*x*, *y*, *z*) = (*x* And *y*) Or (Not(*x*) And *z*)

x	*y*	*z*	*f*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

正準表現

From truth table to a Boolean expression

x	*y*	*z*	*f*
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(Not(x) And Not(y) And Not(z)) Or

(Not(x) And y And Not(z)) Or

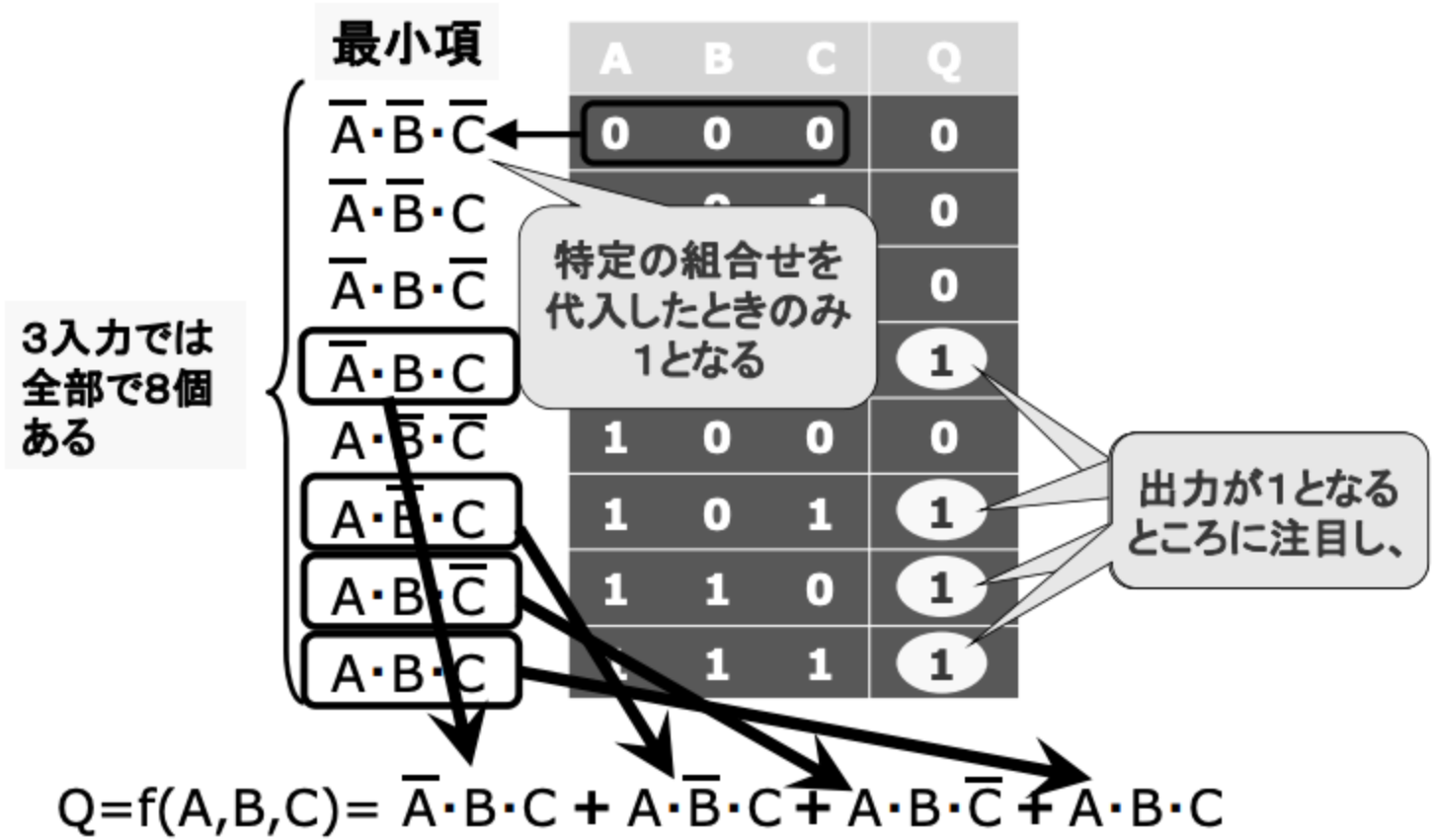
Or

(x And Not(y) And Not(z))

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

組合せ論理回路設計(3)

■ 加法標準形(最小項展開)



正準表現

Theorem

Lemma: Any Boolean function can be represented using an expression containing And, Or And Not operations.

Proof:

Use the truth table to Boolean expression method

Lemma: Any Boolean function can be represented using an expression containing And and Not operations.

Proof:

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

Can we do better than this?

Nand

- $And \circlearrowright$ Not

Nand

x	y	Nand
0	0	1
0	1	1
1	0	1
1	1	0

$(x \text{ Nand } y) = \text{Not}(x \text{ And } y)$

Nand

Theorem (revisited)

Lemma: Any Boolean function can be represented using an expression containing And, Or And Not operations.

Proof:

Use the truth table to Boolean expression method

Lemma: Any Boolean function can be represented using an expression containing And and Not operations.

Proof:

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

Theorem: Any Boolean function can be represented using an expression containing Nand operations only.

Proof:

- Not(x) = (x Nand x)
- (x And y) = Not(x Nand y)

入力のブール関数

- ◦ n個のバイナリに対するブール関数2^{2^n} このブール関数が定義される

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

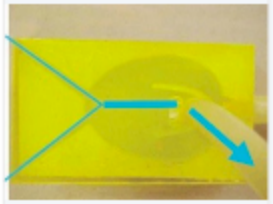
Figure 1.2 All the Boolean functions of two variables.

1.1.2 論理ゲート

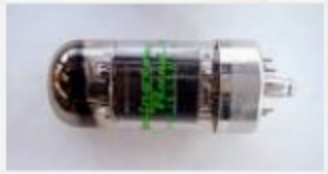
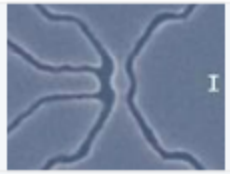
- gateはブール関数を実装するための物理ディバイスである。
  - n個の入力に対してmのバイナリを返すfを考えた場合、それを実装するゲートはn個の入力ピンにとm個の出力ピンを持つことになる。
  - このようなゲートも単純なゲートを組み合わせることで構成することができる。
  - ブール代数はいかなる技術を使ったとしても抽象化で切ることを表している
  - ブール代数と抽象化されたゲートについてのみ考えればよく、ハードウェアについては、プロに任せよう

基本ゲートと複合ゲート

Switches and wires: a first level of abstraction

technology	*"information"*	*switch*
pneumatic	air pressure	
fluid	water pressure	
relay (now)	electric potential	

Amusing attempts that do not scale but prove the point

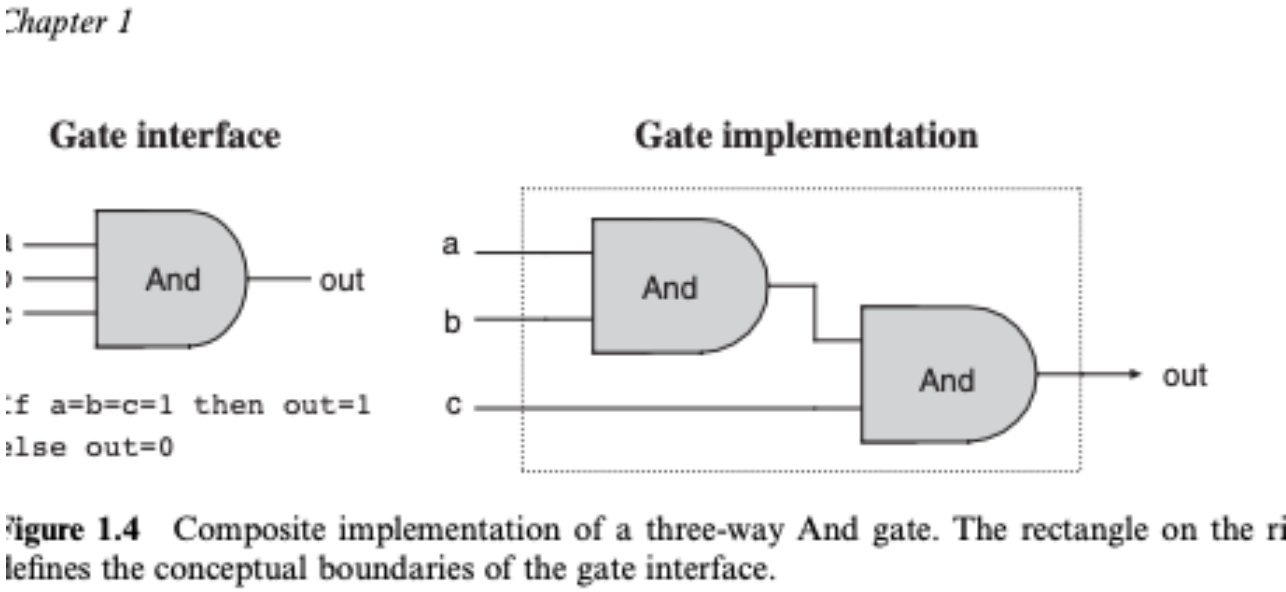
technology	*switch*
relay (1940s)	
vacuum tube	
transistor	
"pass transistor" in integrated circuit	
atom-thick transistor	

Real-world examples that prove the point

9

基本ゲートと複合ゲート

- $And(a, b, c)$ の実装を考える
  - $a * b * c$ とブール代数を使うとかける



Xor のれい

- $Or(And(a, Not(b)), And(Not(a),$
  とかける

つまり

- インターフェースは一つしか存在しないが、実装方法はたくさんある
- なるべくゲートが少なくなるように実装しましょう

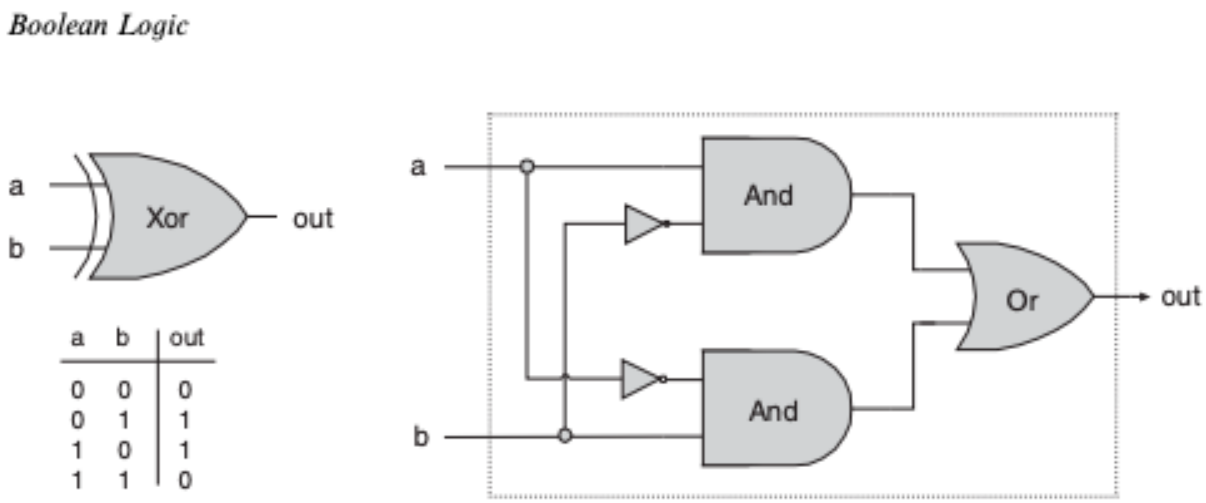


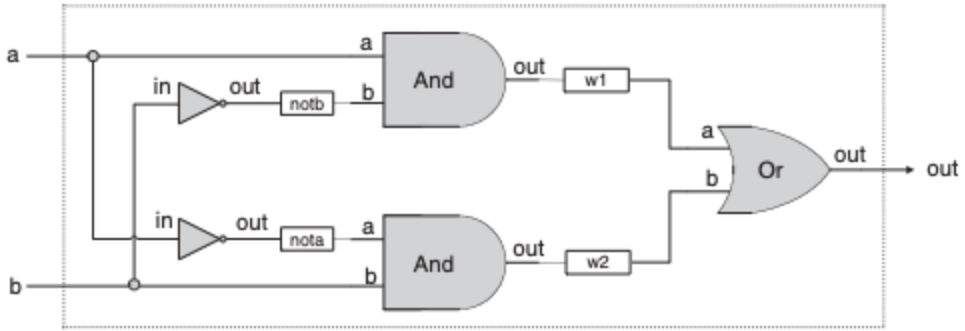
Figure 1.5 Xor gate, along with a possible implementation.

1.1.4　ハードウェア記述言語(HDL)

- これらのゲートを実際に作りたい
  - 大変そう
- HDLを使って回路を設計しよう
  - テストが簡単
  - hardware simulatorを使ってHDLで書かれたプログラムを入力として読み込み、メモリ上にそのプログラムで指定された回路を表現する。
  - その回路に対してテストを走らせて動いてるかどうか確認する

1.1.4　ハードウェア記述言語(HDL)

16　　*Chapter 1*



HDL program (Xor.hdl)	***Test script*** (Xor.tst)	***Output file*** (Xor.out)

/* 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);
}
	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;
		a | b | out
		0 | 0 | 0
		0 | 1 | 1
		1 | 0 | 1
		1 | 1 | 0

ハードウェアシュミレータをダウンロードしよう

TODO

課題

Project 1

Given: Nand

Goal: Build the following gates:

Elementary logic gates	16-bit variants	Multi-way variants
Not	Not16	Or8Way
And	And16 ←	Mux4Way16 ←
Or	Or16	Mux8Way16
Xor	Mux16	DMux4Way
Mux ←		DMux8Way
DMux ←		

Why these 15 particular gates?

- Commonly used gates
- Comprise all the elementary logic gates needed to build our computer.

Nand

1.2.1 The Nand Gate

The starting point of our computer architecture is the Nand gate, from which all other gates and chips are built. The Nand gate is designed to compute the following Boolean function:

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

Throughout the book, we use "chip API boxes" to specify chips. For each chip, the API specifies the chip name, the names of its input and output pins, the function or operation that the chip effects, and an optional comment.

```
Chip name: Nand
Inputs:    a, b
Outputs:   out
Function:  If a=b=1 then out=0 else out=1
Comment:   This gate is considered primitive and thus there is
           no need to implement it.
```

Not

1.2.2 Basic Logic Gates

Some of the logic gates presented here are typically referred to as "elementary" or "basic." At the same time, every one of them can be composed from Nand gates alone. Therefore, they need not be viewed as primitive.

Not The single-input Not gate, also known as "converter," converts its input from 0 to 1 and vice versa. The gate API is as follows:

```
Chip name: Not
Inputs:    in
Outputs:   out
Function:  If in=0 then out=1 else out=0.
```

And The And function returns 1 when both its inputs are 1, and 0 otherwise.

```
Chip name: And
Inputs:    a, b
Outputs:   out
Function:  If a=b=1 then out=1 else out=0.
```

Or The Or function returns 1 when at least one of its inputs is 1, and 0 otherwise.

```
Chip name: Or
Inputs:    a, b
Outputs:   out
Function:  If a=b=0 then out=0 else out=1.
```

Xor The Xor function, also known as "exclusive or," returns 1 when its two inputs have opposing values, and 0 otherwise.

```
Chip name: Xor
Inputs:    a, b
Outputs:   out
Function:  If a≠b then out=1 else out=0.
```

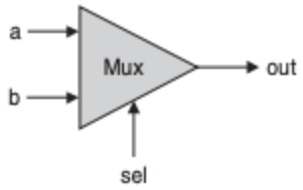
MUX

Chip name: Mux
Inputs: a, b, sel
Outputs: out
Function: If sel=0 then out=a else out=b.

Boolean Logic

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

sel	out
0	a
1	b



DMUX

sel	a	b
0	in	0
1	0	in

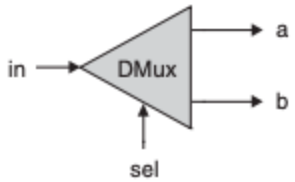


Figure 1.9 Demultiplexor.

Demultiplexor A demultiplexor (figure 1.9) performs the opposite function of a multiplexor: It takes a single input and channels it to one of two possible outputs according to a selector bit that specifies which output to chose.

```
Chip name: DMux
Inputs:    in, sel
Outputs:   a, b
Function:  If sel=0 then {a=in, b=0} else {a=0, b=in}.
```

多ビットの基本ゲート

- コンピュータのハードウェアはバスと呼ばれるタビットの配列を操作するように設計されているのが一般的である
- 16ビットのコンピュータを作るので16ビットの入力

多ビットの基本ゲート

Multi-Bit Not An *n*-bit Not gate applies the Boolean operation Not to every one of the bits in its *n*-bit input bus:

```
Chip name: Not16
Inputs:    in[16] // a 16-bit pin
Outputs:   out[16]
Function:  For i=0..15 out[i]=Not(in[i]).
```

Multi-Bit And An *n*-bit And gate applies the Boolean operation And to every one of the *n* bit-pairs arrayed in its two *n*-bit input buses:

```
Chip name: And16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  For i=0..15 out[i]=And(a[i],b[i]).
```

Multi-Bit Or An *n*-bit Or gate applies the Boolean operation Or to every one of the *n* bit-pairs arrayed in its two *n*-bit input buses:

```
Chip name: Or16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  For i=0..15 out[i]=Or(a[i],b[i]).
```

多ビットの基本ゲート

Multi-Bit Multiplexor An *n*-bit multiplexor is exactly the same as the binary multiplexor described in figure 1.8, except that the two inputs are each *n*-bit wide; the selector is a single bit.

```
Chip name: Mux16
Inputs:    a[16], b[16], sel
Outputs:   out[16]
Function:  If sel=0 then for i=0..15 out[i]=a[i]
           else for i=0..15 out[i]=b[i].
```

1.2.4 Multi-Way Versions of Basic Gates

Many 2-way logic gates that accept two inputs have natural generalization to multi-way variants that accept an arbitrary number of inputs. This section describes a set of multi-way gates that will be used subsequently in various chips in our computer architecture. Similar generalizations can be developed for other architectures, as needed.

Multi-Way Or An *n*-way Or gate outputs 1 when at least one of its *n* bit inputs is 1, and 0 otherwise. Here is the 8-way variant of this gate:

```
Chip name: Or8Way
Inputs:    in[8]
Outputs:   out
Function:  out=Or(in[0],in[1],...,in[7]).
```

Multi-Way/Multi-Bit Multiplexor An *m*-way *n*-bit multiplexor selects one of *m* *n*-bit input buses and outputs it to a single *n*-bit output bus. The selection is speci-

そのた

sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d

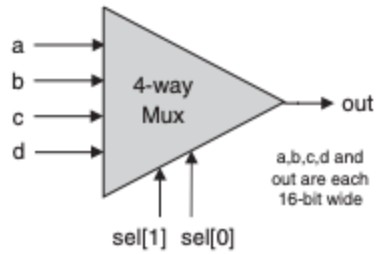


Figure 1.10 4-way multiplexor. The width of the input and output buses may vary.

```
Chip name: Mux4Way16
Inputs:    a[16], b[16], c[16], d[16], sel[2]
Outputs:   out[16]
Function:  If sel=00 then out=a else if sel=01 then out=b
           else if sel=10 then out=c else if sel=11 then out=d
Comment:   The assignment operations mentioned above are all
           16-bit. For example, "out=a" means "for i=0..15
           out[i]=a[i]".
```

```
Chip name: Mux8Way16
Inputs:    a[16],b[16],c[16],d[16],e[16],f[16],g[16],h[16],
           sel[3]
Outputs:   out[16]
Function:  If sel=000 then out=a else if sel=001 then out=b
           else if sel=010 out=c ... else if sel=111 then out=h
Comment:   The assignment operations mentioned above are all
           16-bit. For example, "out=a" means "for i=0..15
           out[i]=a[i]".
```

そのた

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

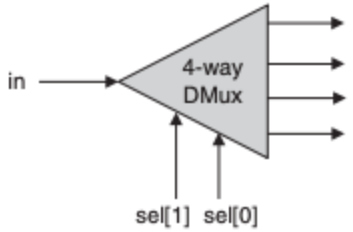


Figure 1.11 4-way demultiplexor.

```
Chip name: DMux4Way
Inputs:    in, sel[2]
Outputs:   a, b, c, d
Function:  If sel=00 then      {a=in, b=c=d=0}
           else if sel=01 then {b=in, a=c=d=0}
           else if sel=10 then {c=in, a=b=d=0}
           else if sel=11 then {d=in, a=b=c=0}.
```

```
Chip name: DMux8Way
Inputs:    in, sel[3]
Outputs:   a, b, c, d, e, f, g, h
Function:  If sel=000 then      {a=in, b=c=d=e=f=g=h=0}
           else if sel=001 then {b=in, a=c=d=e=f=g=h=0}
           else if sel=010 ...
           ...
           else if sel=111 then {h=in, a=b=c=d=e=f=g=0}.
```

そのた

Good luck