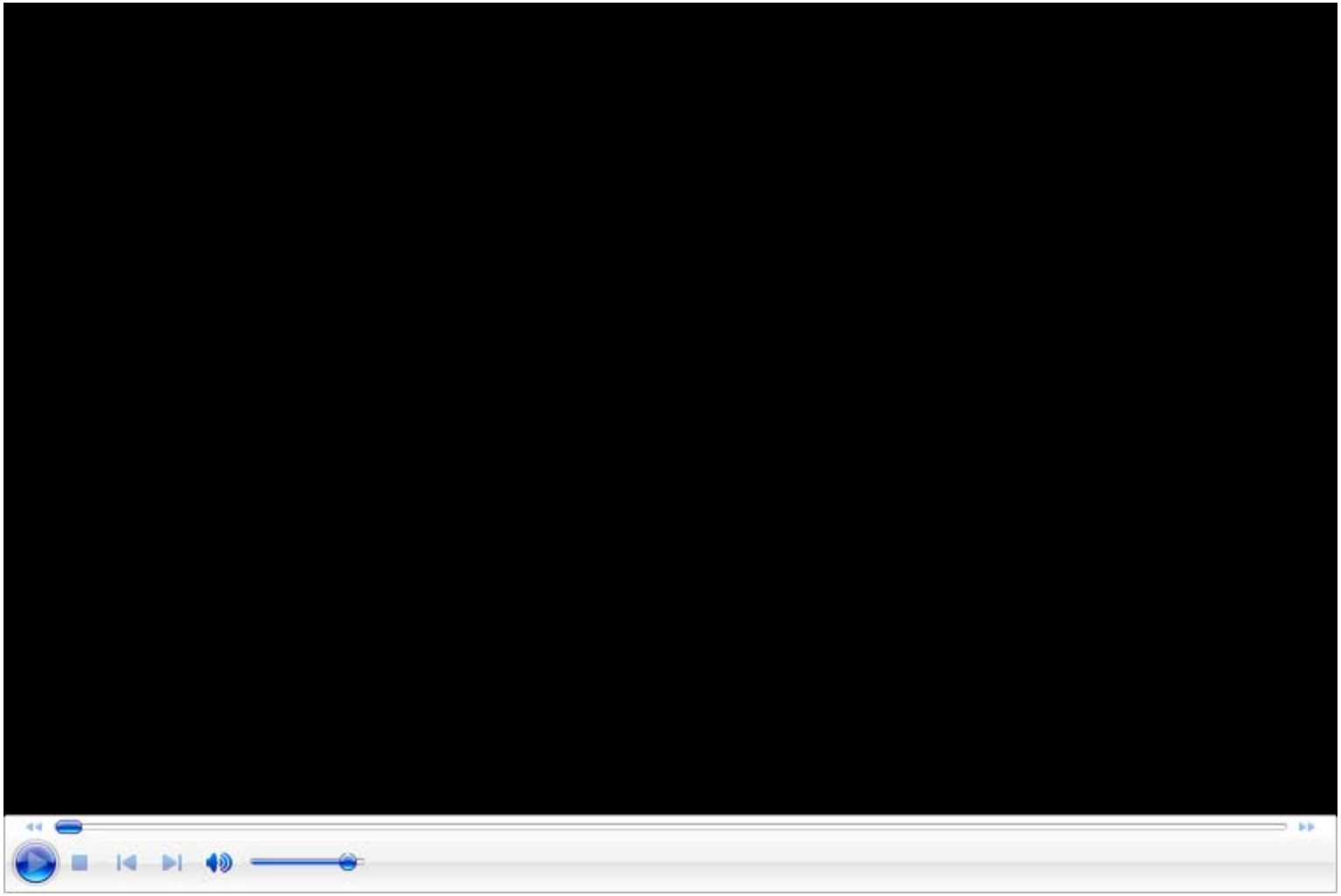
論理演算

内容

- 論理演算
- 論理値 0 と 1
- 論理演算 AND, OR, NOT
- よく使う他の論理演算 NAND, NOR, XOR
- 論理ゲート回路

- ビットごと論理演算
- 論理シフト
- 符号拡張・ゼロ拡張

「8逃げ」



http://www.youtube.com/watch?v=_Ao1ofpI0mE

例題

ファミリーコンピュータ用ゲーム「ドラゴンクエストIV 導かれし者たち」((株) エニックス, 1990年) では，敵との戦闘中に「にげる」操作を8 回行うと，それ以降，敵へのすべての攻撃が強力なものとなる(会心の一撃と呼ばれる) 現象が生じた．内部でどのような処理が行われていたか推測して述べよ．

(2010年度 期末試験)

論理値と論理演算

論理値

- 「3 は 4 より小さい」は真 (true): 数値 1 で表す
- 「カエルは哺乳動物である」は偽 (false): 数値 0 で表す

論理演算:

- 論理積 (AND): $A \cdot B$ あるいは単に AB
  他の記法: $A \wedge B,\ A \,\&\, B$
- 論理和 (OR): $A + B$
  他の記法: $A \vee B,\ A \mid B$
- 論理否定 (NOT): $\overline{A}$
  他の記法: $\neg A,\ A',\ !A,\ \tilde{}A$

真理値表とゲート記号

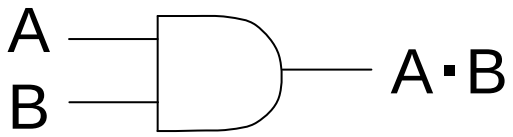
真理値表

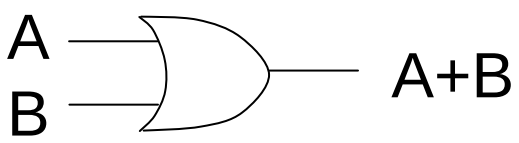
A	B	A･B
0	0	0
0	1	0
1	0	0
1	1	1

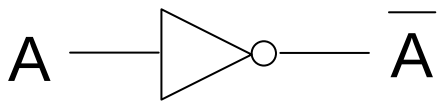
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

A	$\overline{A}$
0	1
1	0

ゲート記号



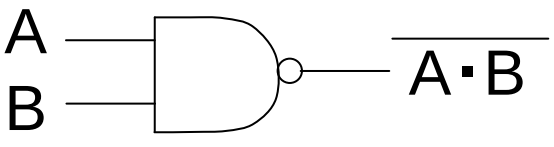




よく使う他の論理演算

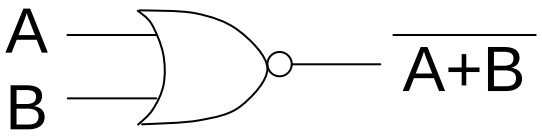
否定論理積 (NAND)

A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0



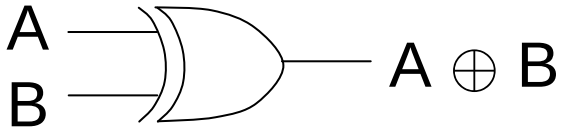
否定論理和 (NOR)

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0



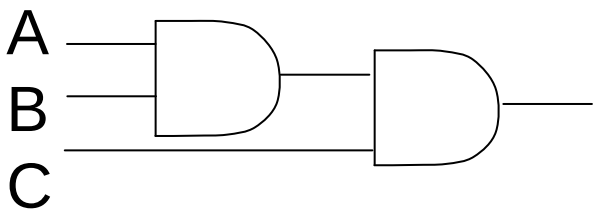
排他的論理和 (eXclusive OR) (XOR)

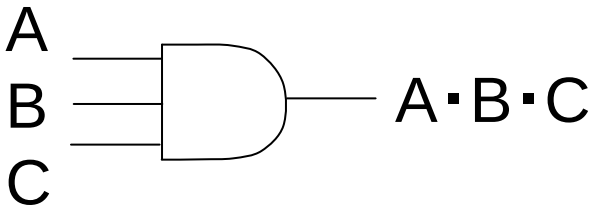
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

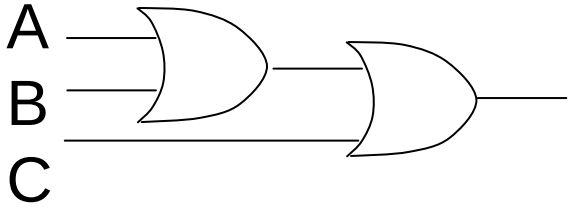


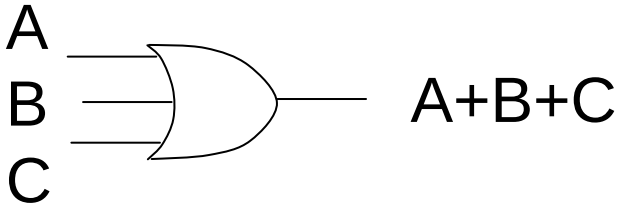
- 片方だけ 1 のときに限って1
- つまり入力が相異なるときに 1

多入力AND, 多入力OR



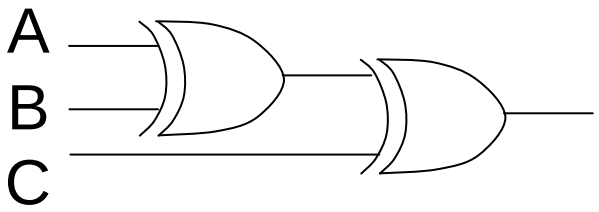


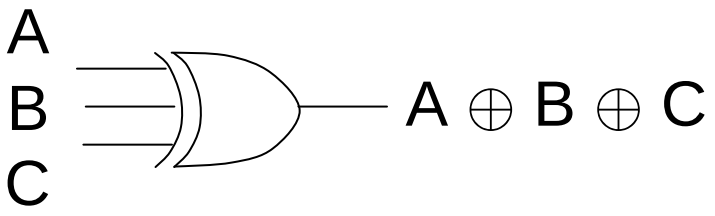




- 論理積も論理和も，交換則 (AB = BA, A+B = B+A) と結合則 ((AB)C = A(BC), (A+B)+C = A+(B+C)) が成り立つ.
- 入力の順序に関係なく,
  - 多入力ANDは入力が一つでも 0 なら 0
  - 多入力 OR は入力が一つでも 1 なら 1
- NAND，NOR も同様

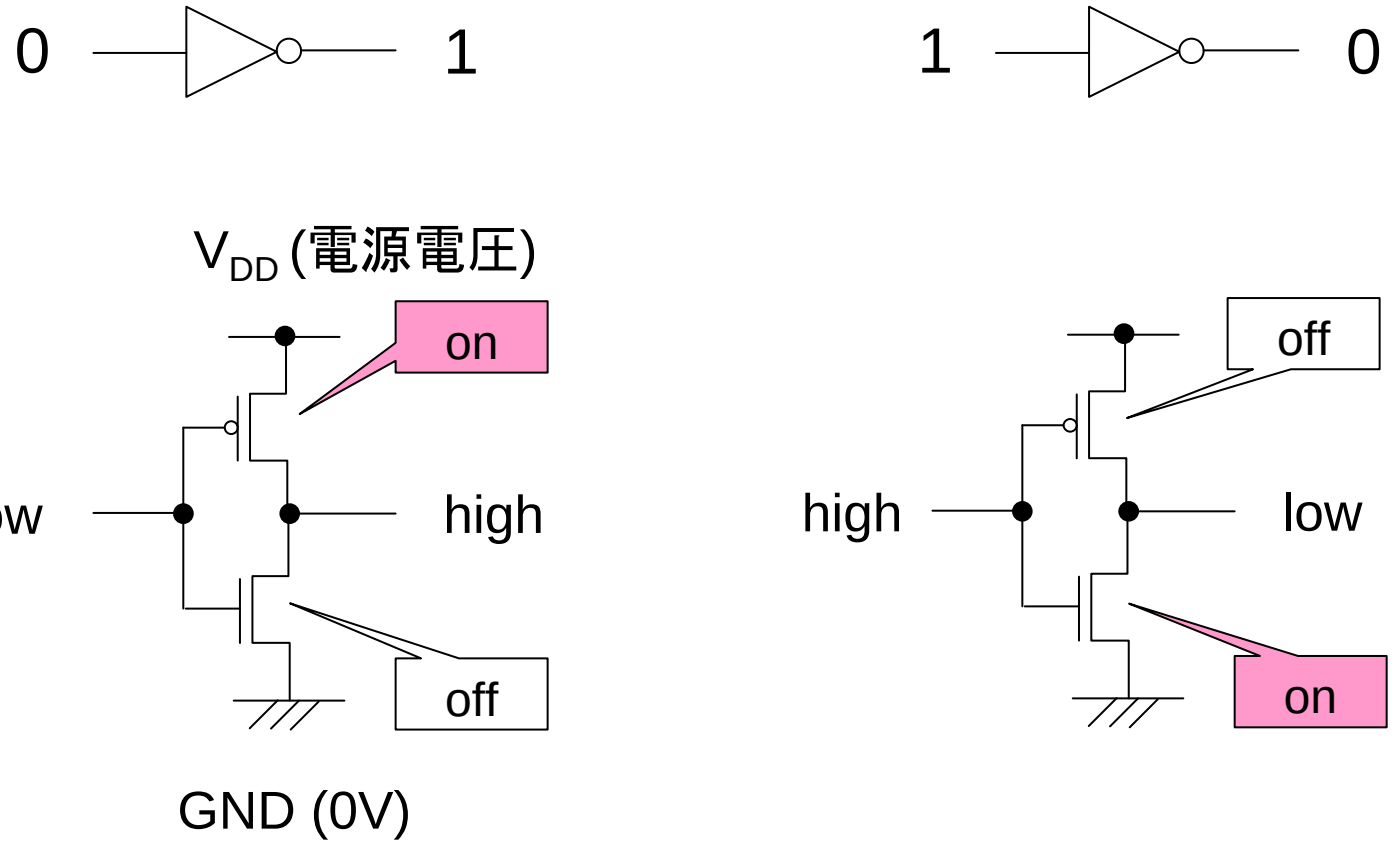
多入力XOR





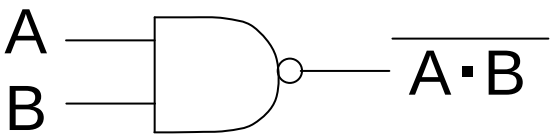
- $X \oplus 1 = \overline{X}$
- $X \oplus 0 = X$
- A, B, C, ... と順に見ていって，入力に 1 が現れる度に出力は反転する
- 結局，多入力 XOR は，入力のうち 1 の数が奇数なら 1に，偶数なら 0 になる

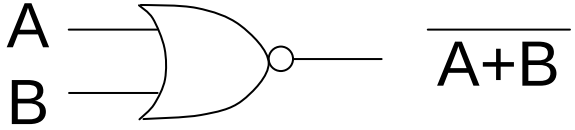
NOTゲート の回路 (インバータ)

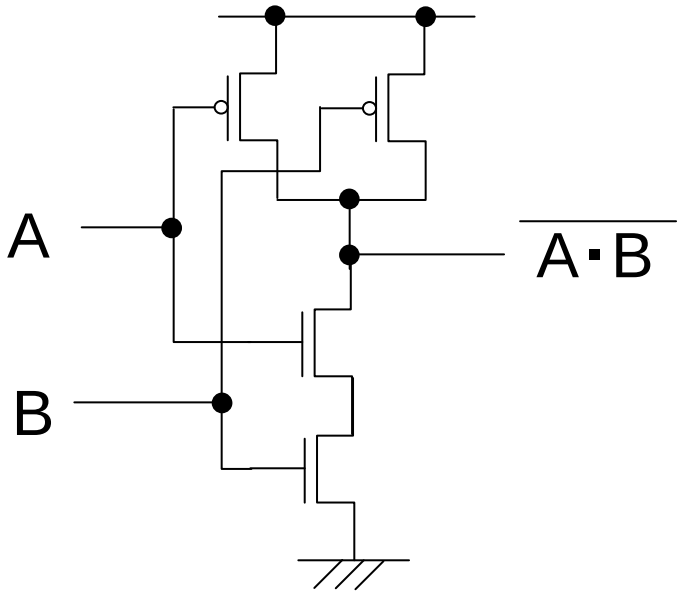


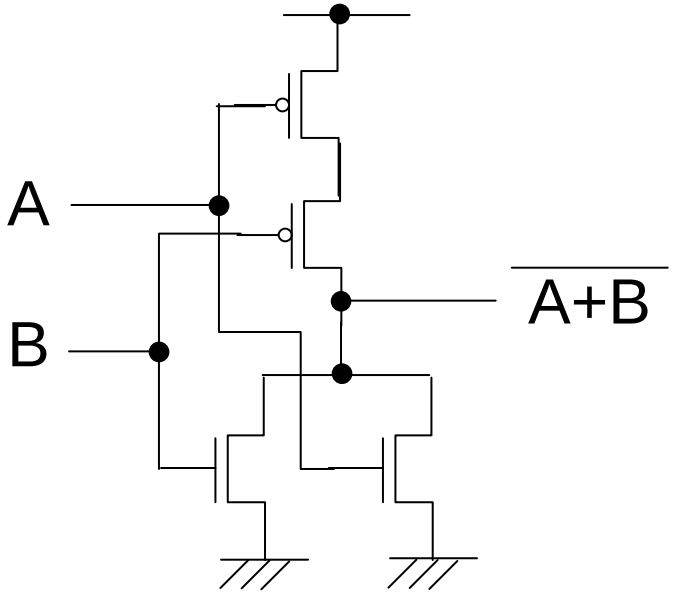
CMOS技術 (Complementary MOS):
NMOS と PMOS を常に対にして使う (低消費電力)

NAND, NORゲートの回路









- AND, OR は NAND, NOR, NOT から作れる
- XOR は AND, OR, NOT から作れる

プログラミングで使う論理演算

```
if (x > 20 && x < 80) {
    ...
}

if (a < 0 || b < 0) {
    ...
}

if (!isalpha(c)) {
    ...
}
```

C言語では，比較演算子 (>, >=, <, <=, ==, !=) は真のとき整数 1 を，偽のとき整数 0 を返す．

演算子 &&, ||, ！ が，論理積，論理和，論理否定を行う．

if や while などの条件部は，0 を偽，0 以外を真とみなして判定する．

ビットごと論理演算

C言語の演算子 &, |, ~,^ などは，&&, ||, ! とは異なり<u>ビットごとの</u>論理演算を行う

```
a = 0x1234;    // 0001 0010 0011 0100
b = 0xcafe;    // 1100 1010 1111 1110

c = a & b;     // 0000 0010 0011 0100 (AND)
c = a | b;     // 1101 1010 1111 1110 (OR)
c = ~a;        // 1110 1101 1100 1011 (NOT)
c = a ^ b;     // 1101 1000 1100 1010 (XOR)
```

論理シフト演算

```
unsigned short a, b, x, y; // 16 bit

a = 0x1234; // 0001 0010 0011 0100
x = a >> 2; // 0000 0100 1000 1101

b = 0xcafe; // 1100 1010 1111 1110
y = b << 3; // 0101 0111 1111 0000
```

シフトによって空いたビットには 0 を詰める

例: フラグを立てる

ドラゴンクエストIV (エニックス) の戦闘状態を保持するメモリ領域の一部は，以下のような構成だったと推測されている:

霧フラグ	?	?	?	会心フラグ	「にげる」コマンドカウンタ		
7	6	5	4	3	2	1	0

```
critical_hit_bit = 1 << 3;      // 0000 1000
fog_bit = 1 << 7;               // 1000 0000

status = status | critical_hit_bit;  // 会心の一撃ON
status = status & ~critical_hit_bit; // 会心の一撃OFF
status = status ^ fog_bit;           // 霧状態を反転

if (status & fog_bit) {
    // 霧状態ならこの処理を実行
}
```

シフト演算と数の関係

ただのビット列ではなく「2進数」に対するシフト演算の効果は?

- 10進数: 左 k 桁シフトは 10^k 倍，右 k 桁シフトは 10^{-k} 倍
- 2進数: 左 k ビットシフトは 2^k 倍，右 k ビットシフトは 2^{-k} 倍

(ただしオーバフローが起きないことを前提とする; 端数は無視)

ただし，符号つき数の場合はこの限りではない．詳しく知りたい者は「算術シフト」について調査するとよい．

符号拡張とゼロ拡張

あるビット長の2進数を，より長いビット長の2進数に変換するとき，MSB 側を符号ビットで埋める方法を**符号拡張**と呼ぶ．2の補数表示の符号つき数の拡張の際に用いられる．

16ビットから32ビットへの符号拡張の例:

00000000 00001000 = 8 (10)

→ 00000000 00000000 00000000 00001000 = 8 (10)

11111111 11111000 = –8(10)

→ 11111111 11111111 11111111 11111000 = –8(10)

(–8 :「あと8 増やせば 2^n になる数」)

これに対して，常にゼロで埋める方法を**ゼロ拡張**と呼ぶ．符号なし数の拡張に用いられる．

練習問題

8ビットの符号つき数

(a) 00101100
(b) 11101101
(c) 10100101

をそれぞれ，

1) 10 進数，16進数で表せ
2) 符号を反転した数を，2の補数表示の符号つき数で表せ
3) 16ビットに符号拡張せよ
4) (a) を 3ビット右シフトし，元の数の8分の1になっているかどうか調べよ.

練習問題 解答例

1) 10進数: (a) 44 (b) – 19 (c) – 91
   16進数: (a) 2c (b) ed (c) a5
2) (a) 11010100 (b) 00010011 (c) 01011011
3) (a) 00000000 00101100 (b) 11111111 11101101
   (c) 11111111 10100101
4) 000000101 (44/8 の小数点以下切捨てになっている)

1) 正の数は普通に変換する. 負の数は, 10進にしてから 2^8 から引くか, 符号反転してから10進にするか. どうせ(2)で符号反転するんだから, 後者の方が楽かも.
2) ビット反転して 1 を足す.
3) MSBで拡張する.
4) 定義どおり計算.