筑波大学数理物質科学研究科 物理学専攻

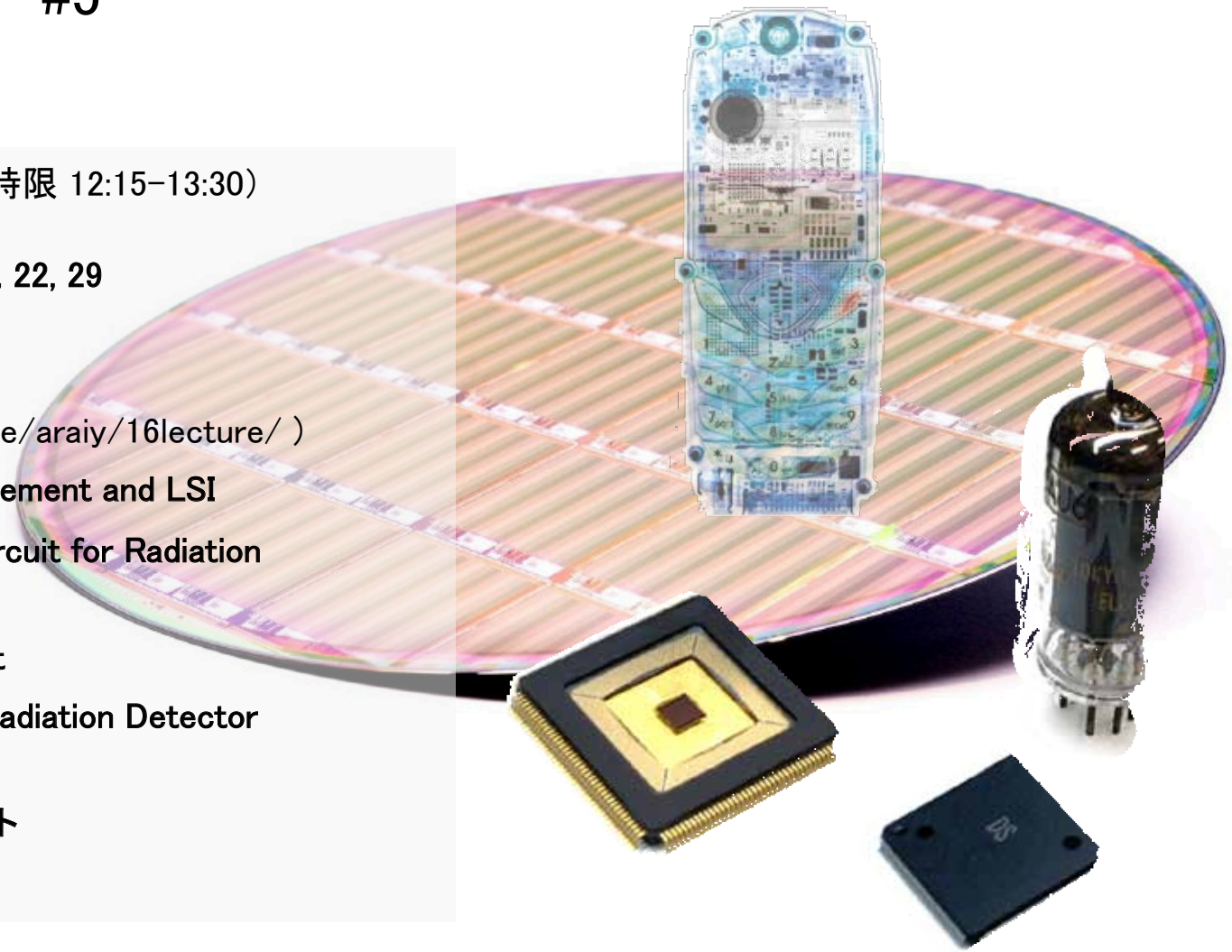
エレクトロニクス・データ処理

（放射線計測とLSI）

#5

2016年7月22日
高エネルギー加速器研究機構
素粒子原子核研究所
新井康夫　(yasuo.arai@kek.jp)

- Lecture Schedule（金曜第3時限 12:15-13:30）
  June　17, 24
  July　1, 8, *15(No lecture)*, 22, 29
  Total 6 times
- Contents
  (http://research.kek.jp/people/araiy/16lecture/)
  - ❑ Radiation Measurement and LSI
  - ❑ Analog CMOS Circuit for Radiation Measurement
  - ❑ Digital LSI Circuit
  - ❑ Semiconductor Radiation Detector
- 単位認定(Credit)
  講義出席及びレポート
  （田中氏の分と合算）



Lecture Plan

1. Radiation measurement and LSI

高エネルギー物理実験とLSI、
LSI技術の変遷
半導体放射線検出器

2. Basic low and tools

オームの法則、
信号伝送、
信号規格 ...

3. Semiconductor devices

半導体の基礎、
半導体プロセス、
MOSデバイス基礎、
...

4. Analog CMOS circuit

1段増幅回路、差動増幅回路、
カレントミラー、
アナログ・シミュレーション、
オペアンプ回路、
雑音, ...

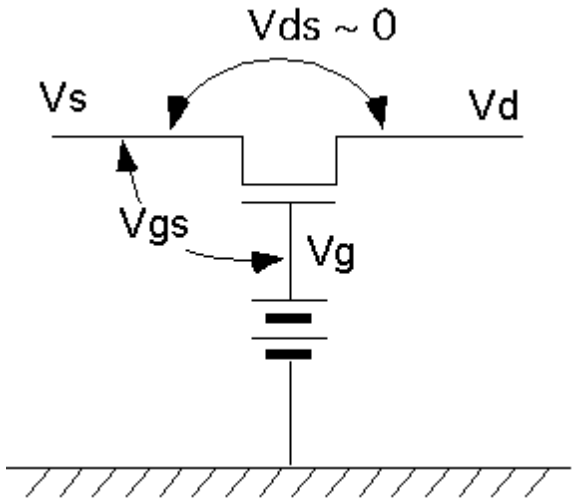
5. Digital LSI circuit

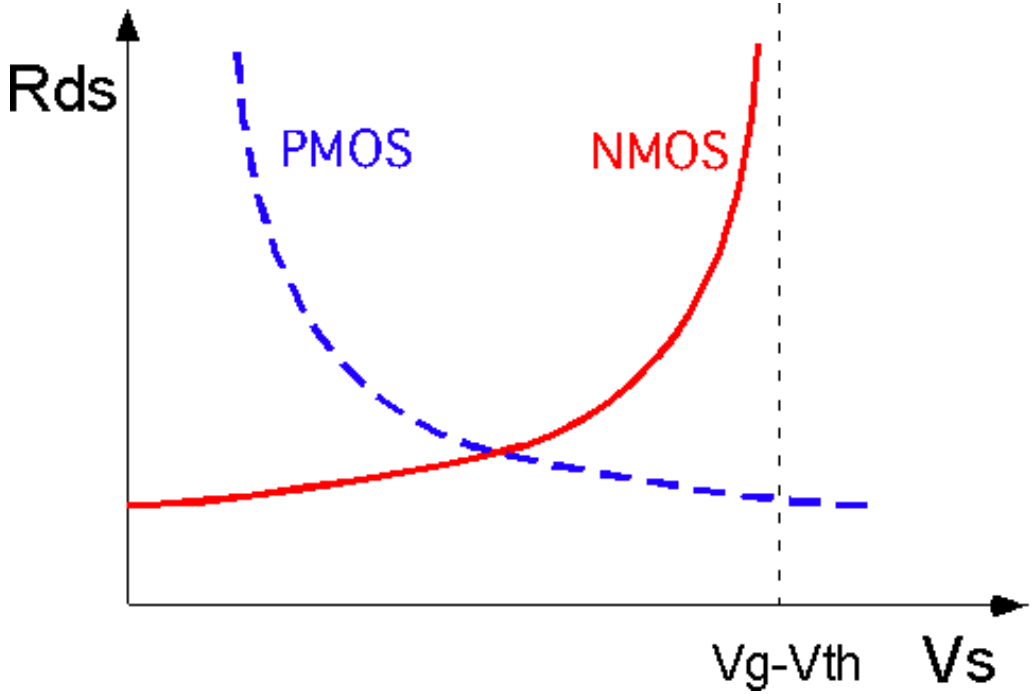
CMOS論理回路、メモリー、
演算器、カウンター、同期回
路, 順序回路、ADC, TDC、...
論理合成

6. Semiconductor radiation detector

放射線の相互作用
検出器の動作原理、
実例、...

MOS Transistor in Digital Circuit





Vd~Vsの時、線形領域なので

$$I_{ds} = \beta\left[\left(V_{gs} - V_{th}\right)V_{ds} - \frac{V_{ds}^2}{2}\right] \approx \beta\left(V_{gs} - V_{th}\right)V_{ds}$$

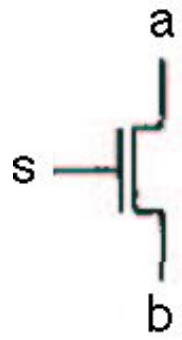
$$R_{ds} = \frac{V_{ds}}{I_{ds}} \approx \frac{1}{\beta\left(V_{gs} - V_{th}\right)} = \frac{1}{\beta\left((V_g - V_{th}) - V_s\right)}$$

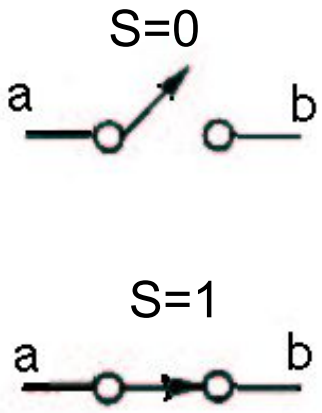
NMOS has low resistance when Source (Drain) voltage is low, but it become high when the voltage is high.

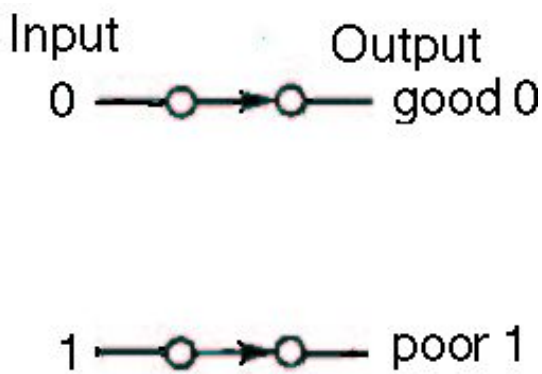
(PMOS has reverse characteristic)

MOS Transistors in Digital Circuit

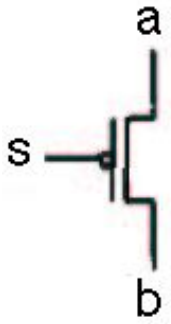
NMOS

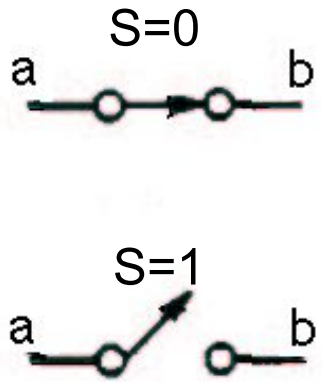


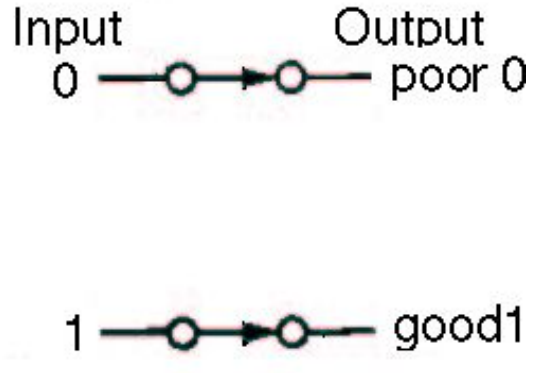




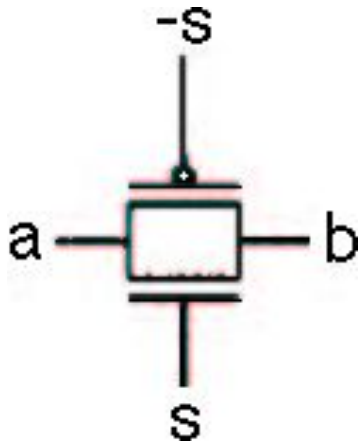
PMOS



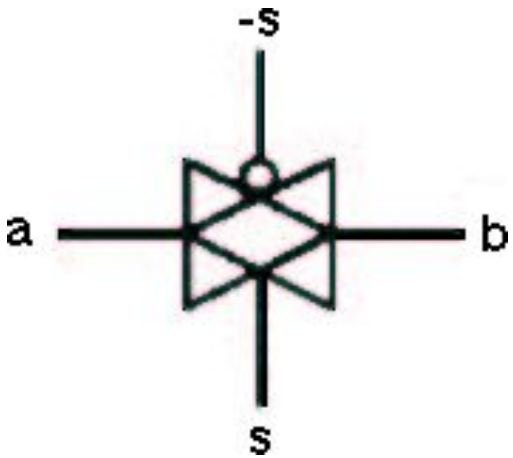


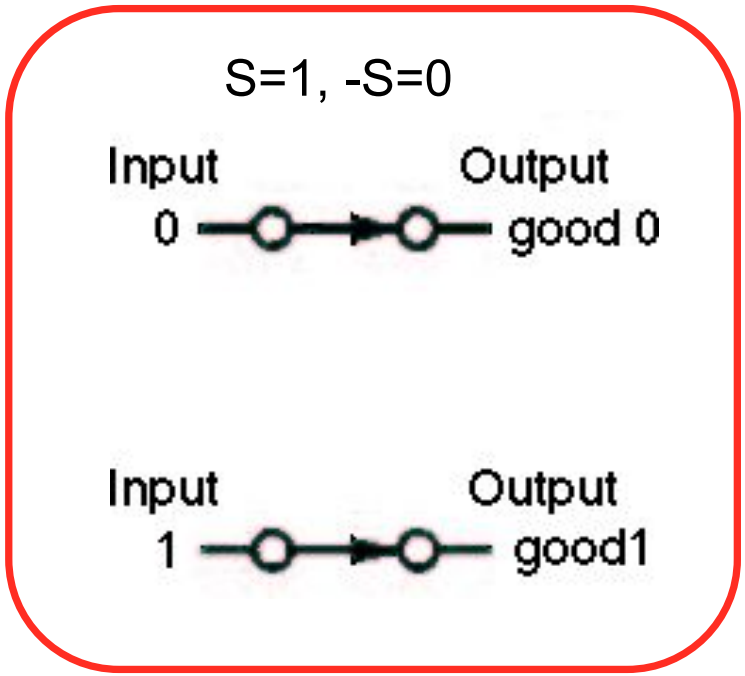


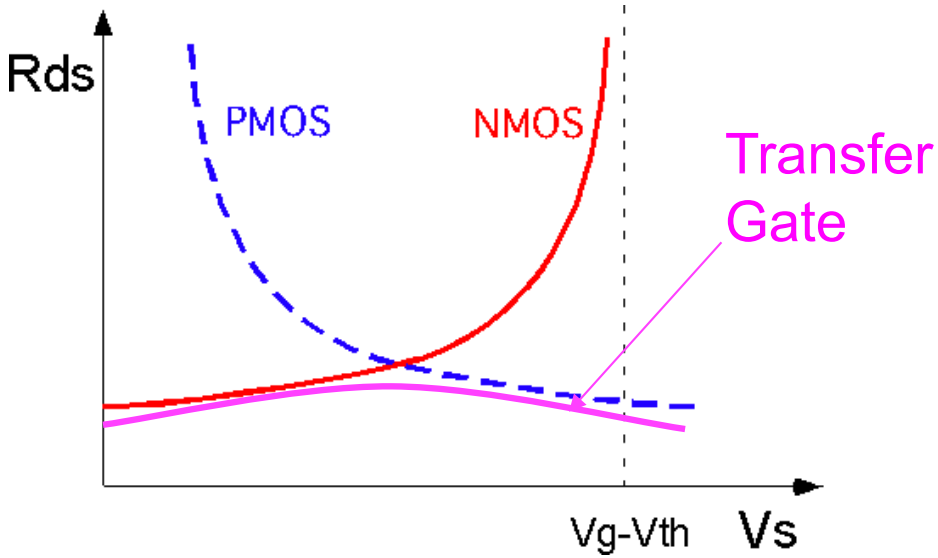
Transfer gate



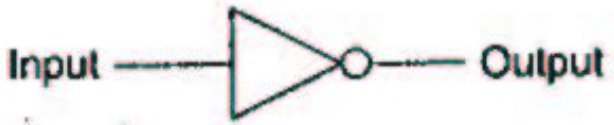
又は



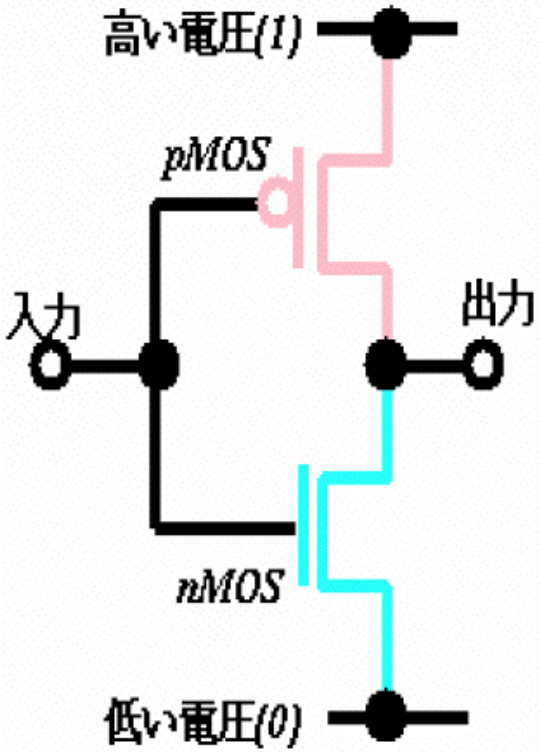


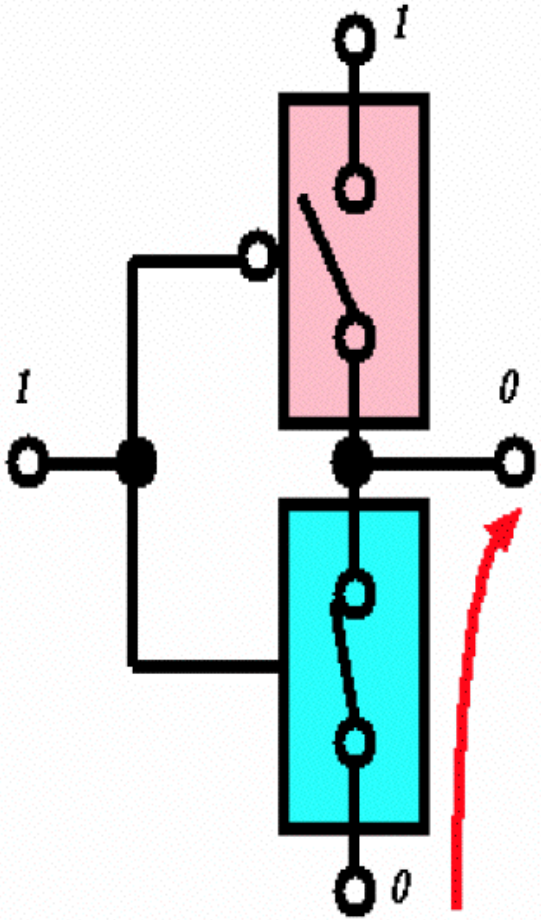


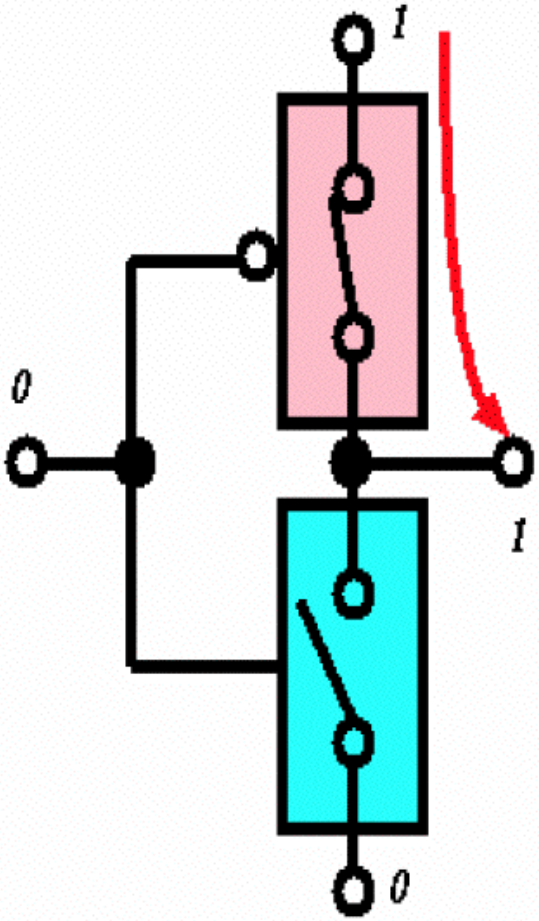
Inverter



Input	Output
0	1
1	0



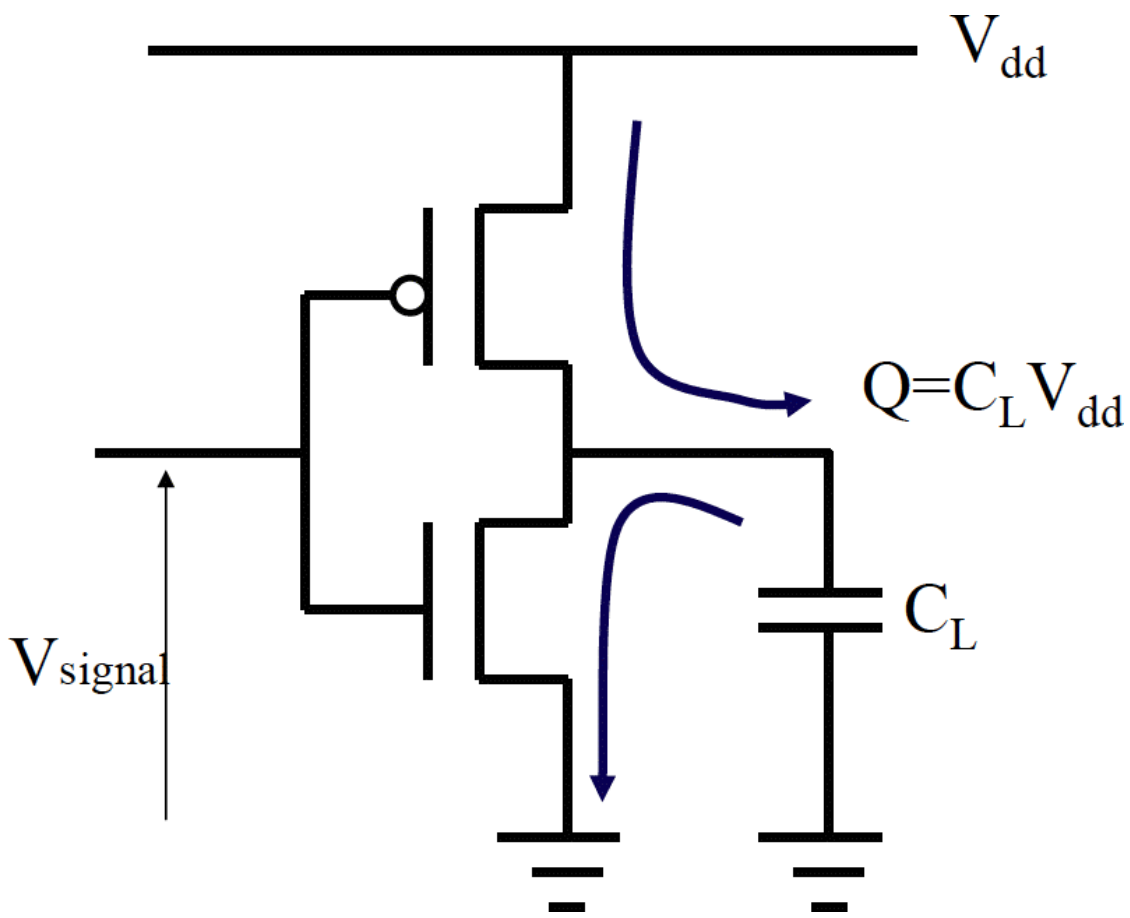




Power Consumption

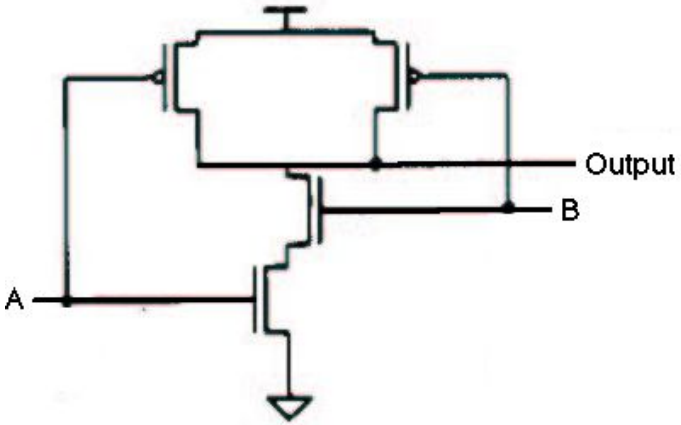
$$P = C \cdot V^2 \cdot F$$

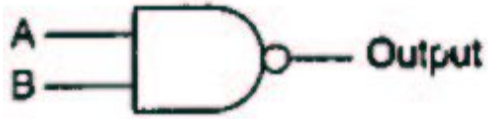
C: Load Capacitance
V: Power Supply Voltage
F : Switching Frequency



CMOSではスイッチングの際に電流が流れるのみで、定常状態では電流が流れない。

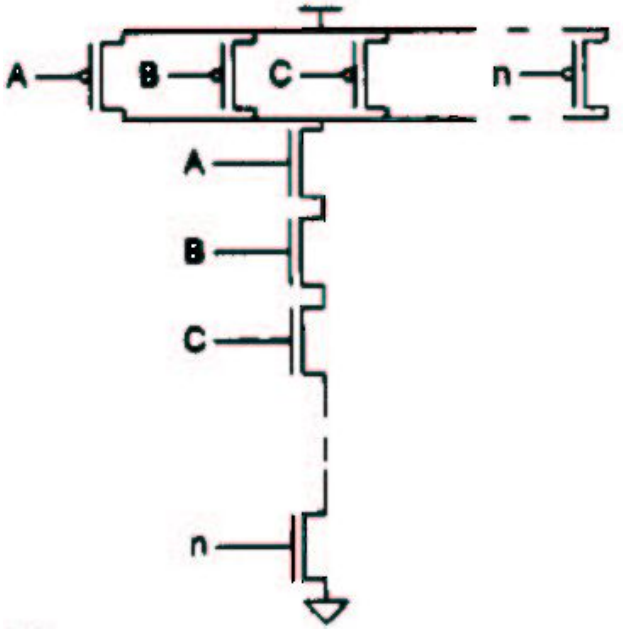
2 input NAND





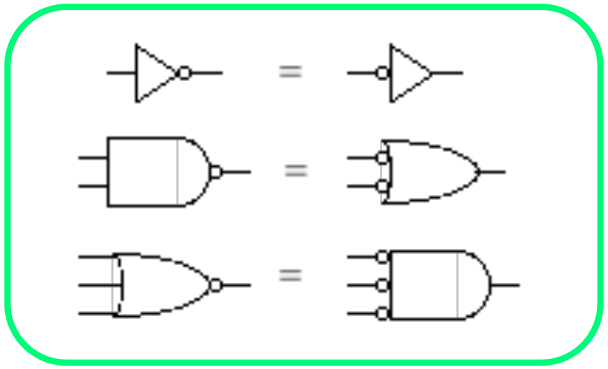
A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

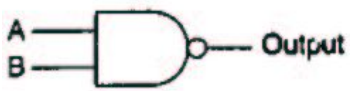
n input NAND



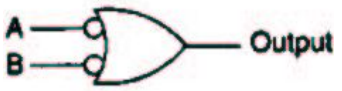
How to write digital logic ...

- 正論理：電圧レベルがHiの時を1(Assert)、Lowの時を0(Negate)。
- 負論理：電圧レベルがLowの時を1(Assert)、Hiの時を0(Negate)。
- 正論理入力のNAND(NOR)は負論理入力のOR(AND) と同等。
- 正（負）論理信号は、負（正）論理入力に繋がないように書く。
- ロジックを考える時はHi/Lowではなく Assert/Negate で考える。



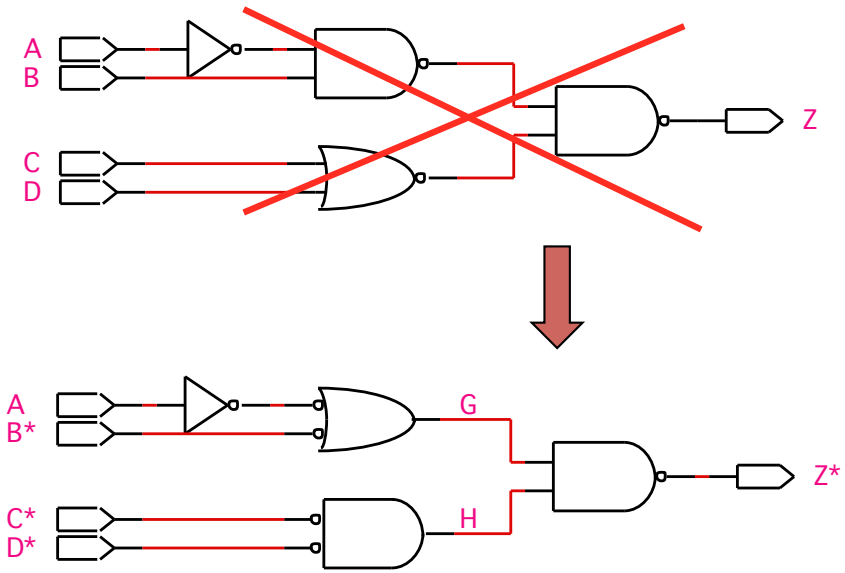


A	B	Output
0(N)	0(N)	1(N)
0(N)	1(A)	1(N)
1(A)	0(N)	1(N)
1(A)	1(A)	0(A)

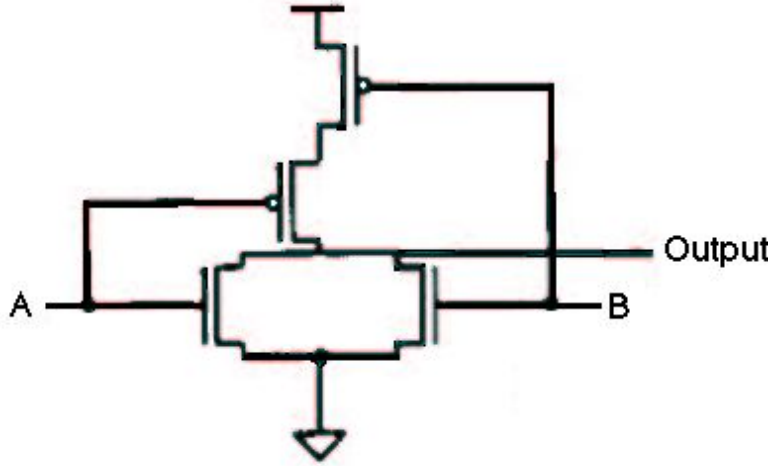


Equal

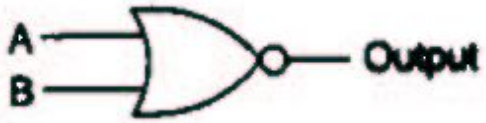
A	B	Output
0(A)	0(A)	1(A)
0(A)	1(N)	1(A)
1(N)	0(A)	1(A)
1(N)	1(N)	0(N)



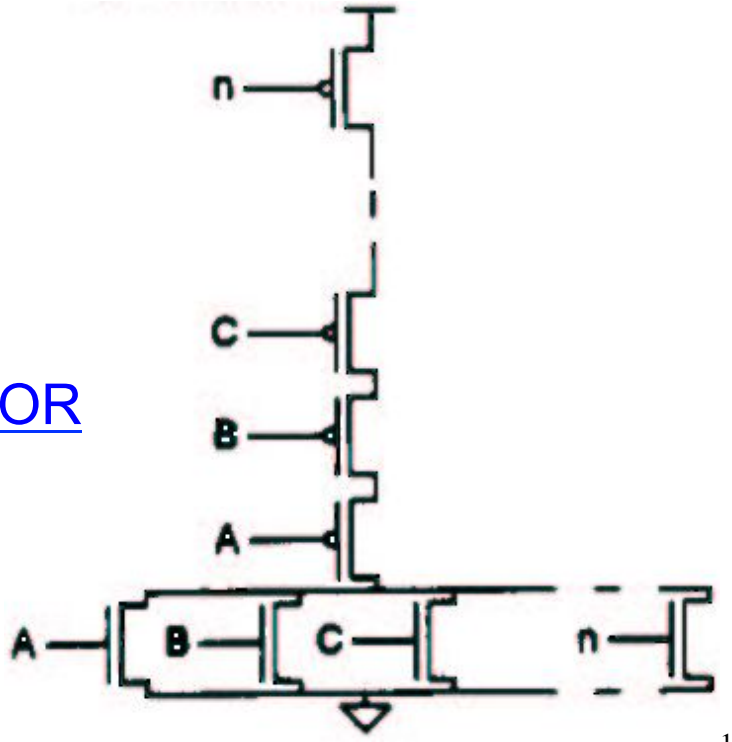
2 input NOR



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

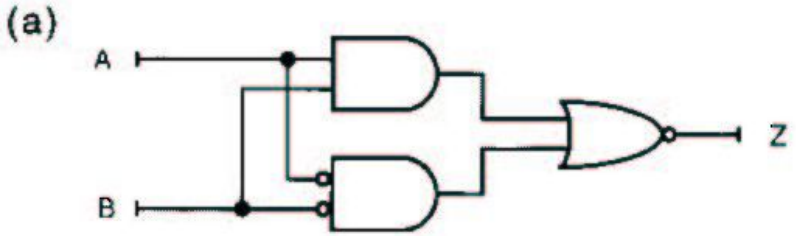


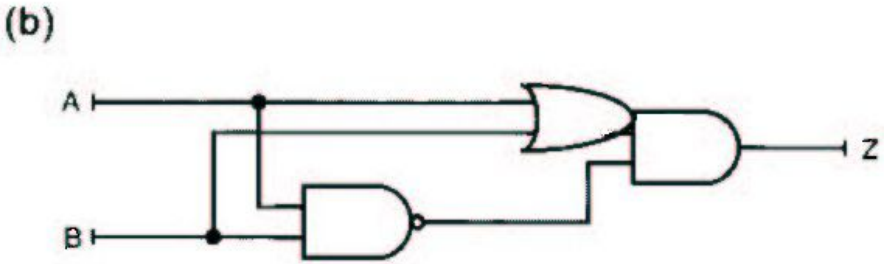
n input NOR

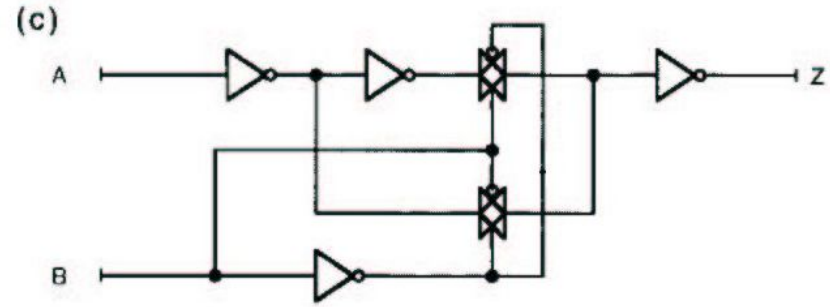


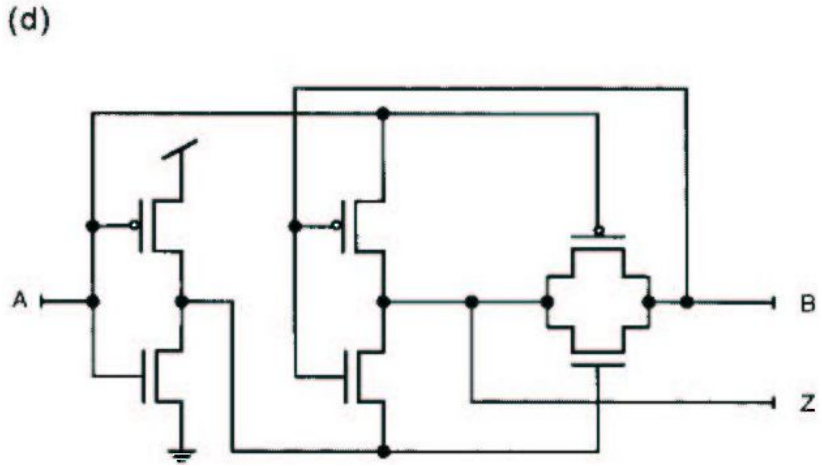
EXOR（Exclusive OR, 排他的論理和)

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0



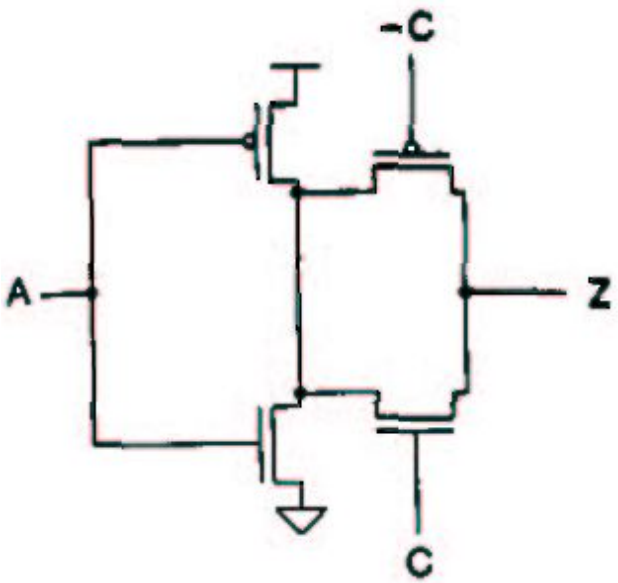


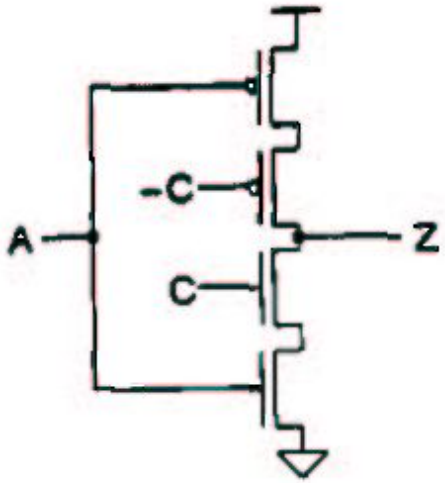


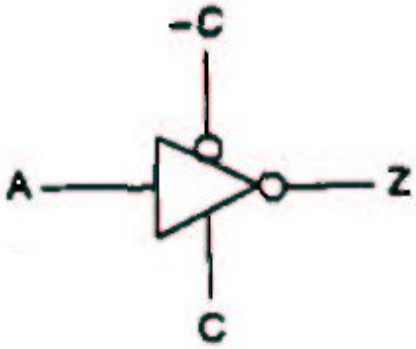


Tri-State Inverter

If C = H then Z = A*
else Z = Hi Impedance

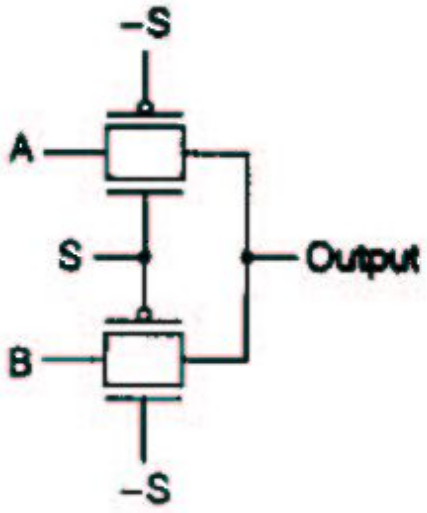


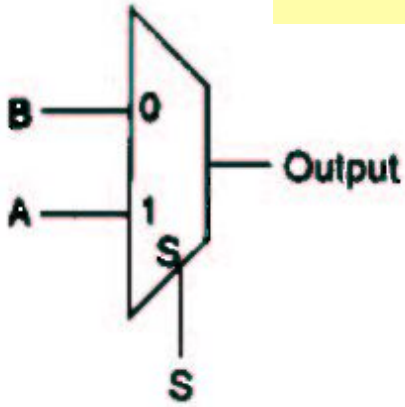




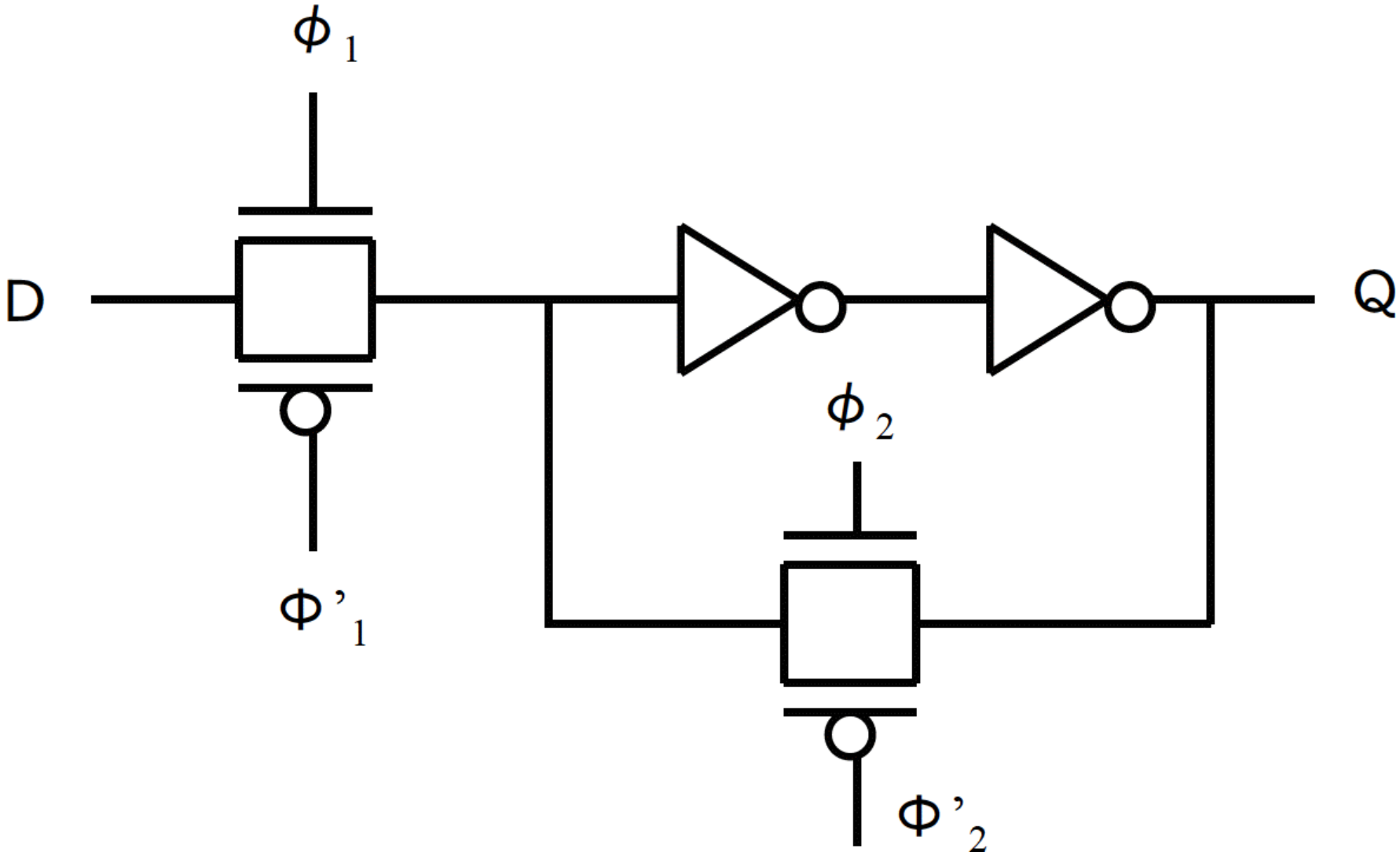
Multiplexor

If S = H then Output = A
else Output = B

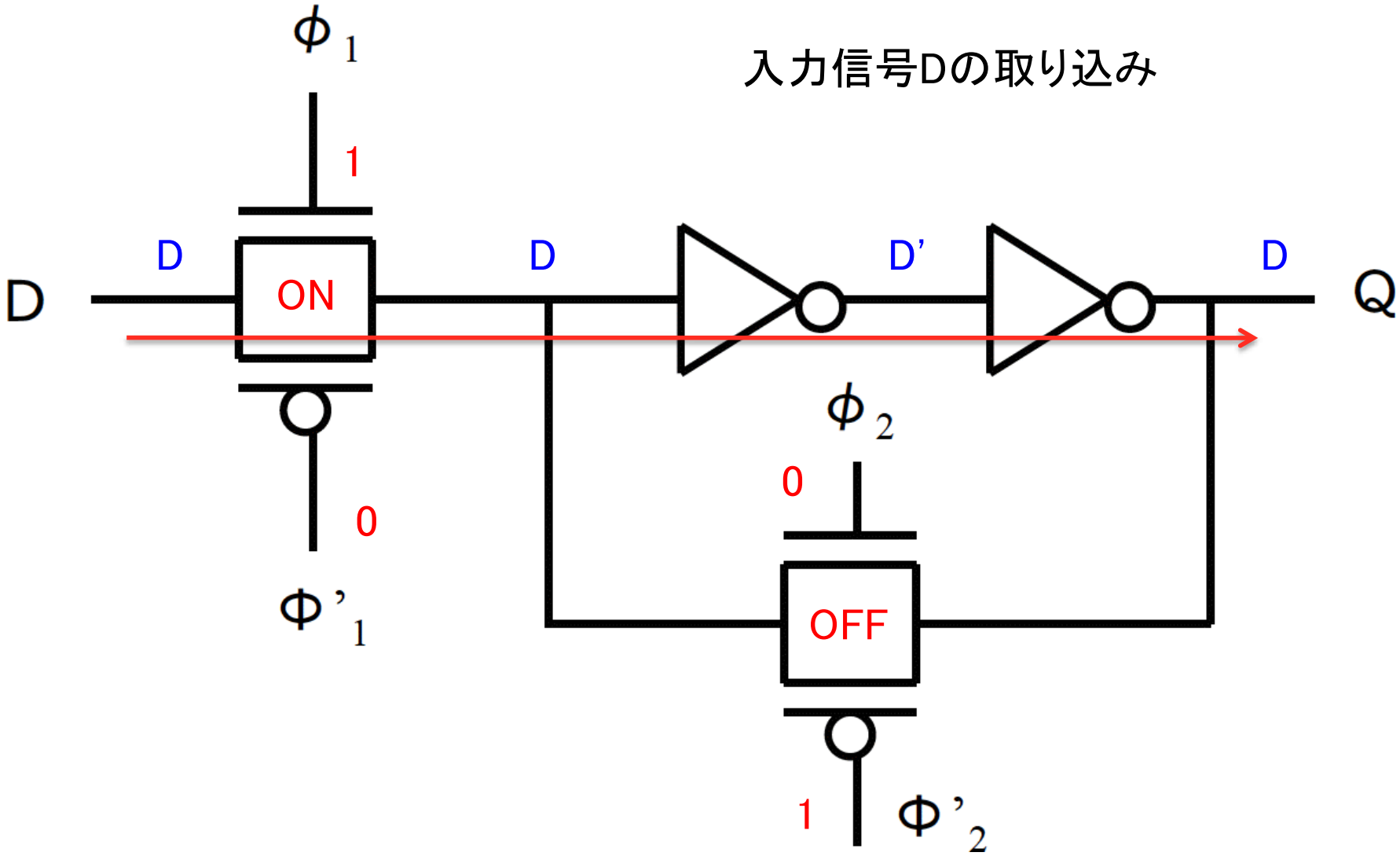




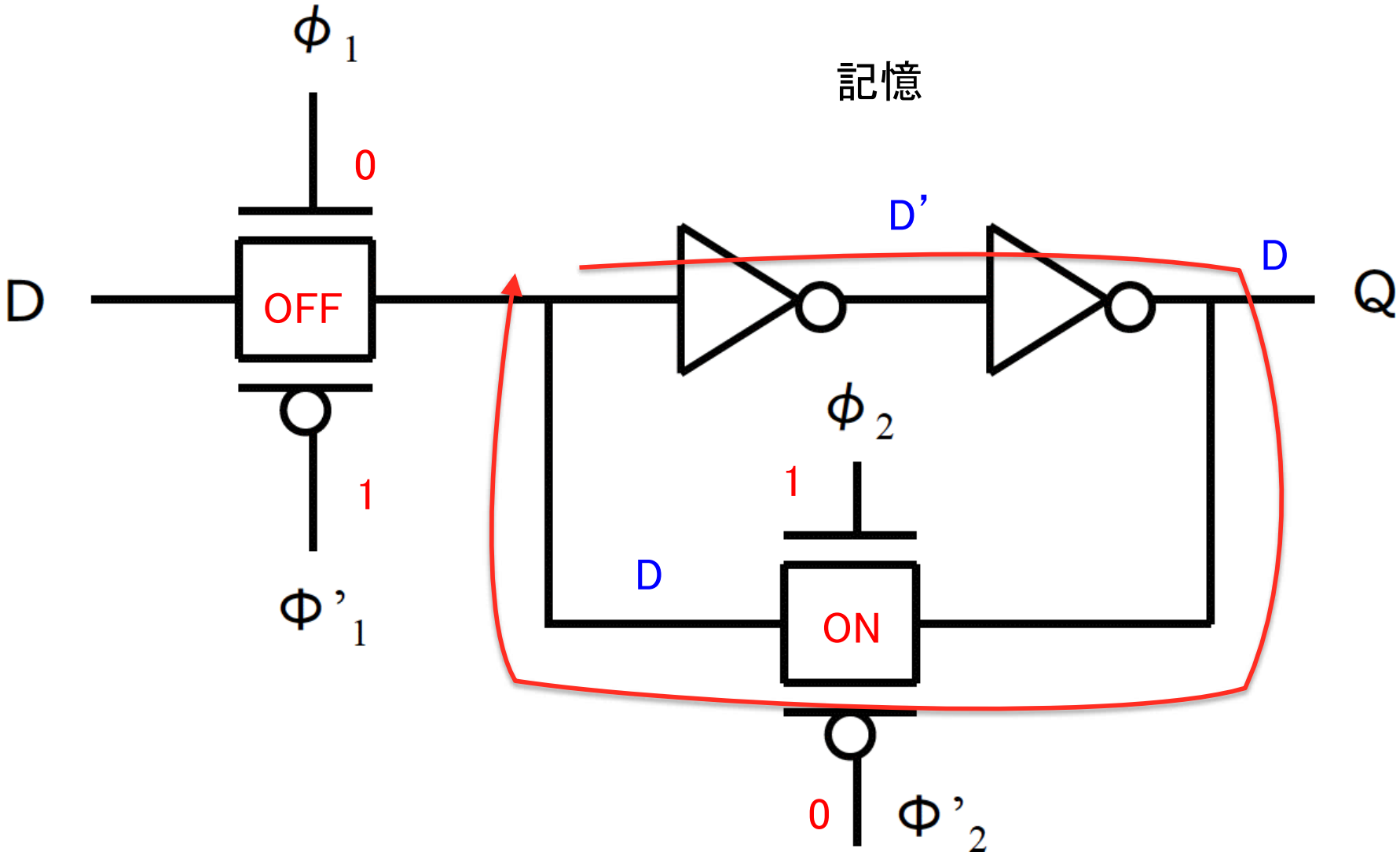
Latch（記憶素子）



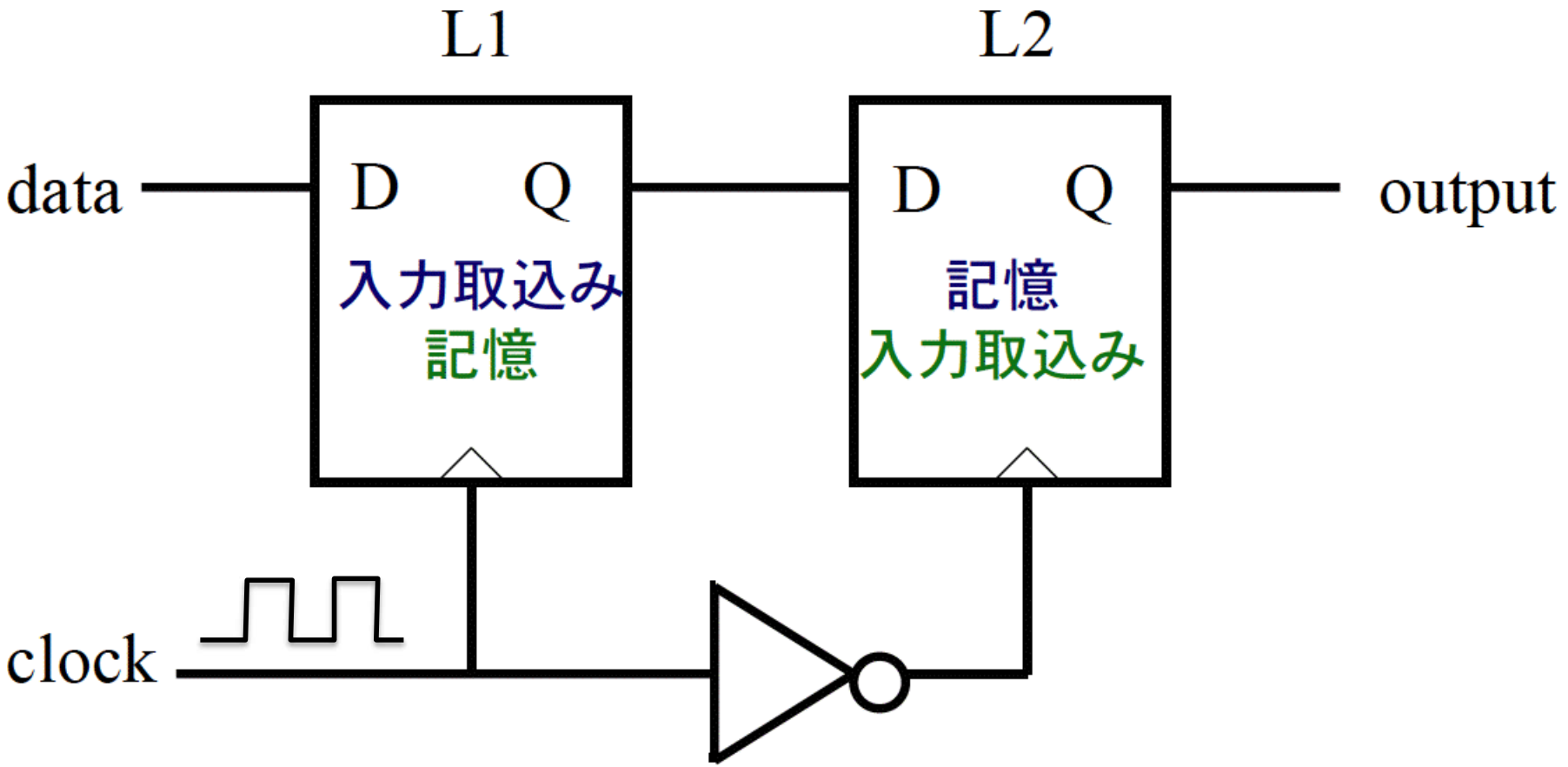
Latch（記憶素子）



Latch（記憶素子）



Master Slave (D-type) Flip Flop

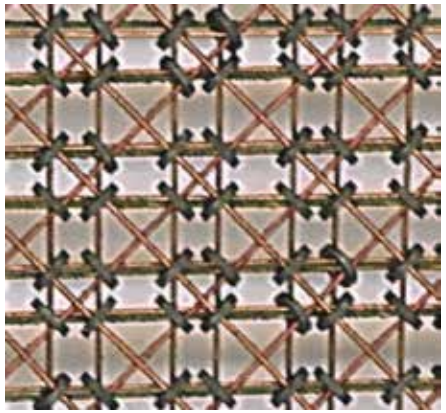


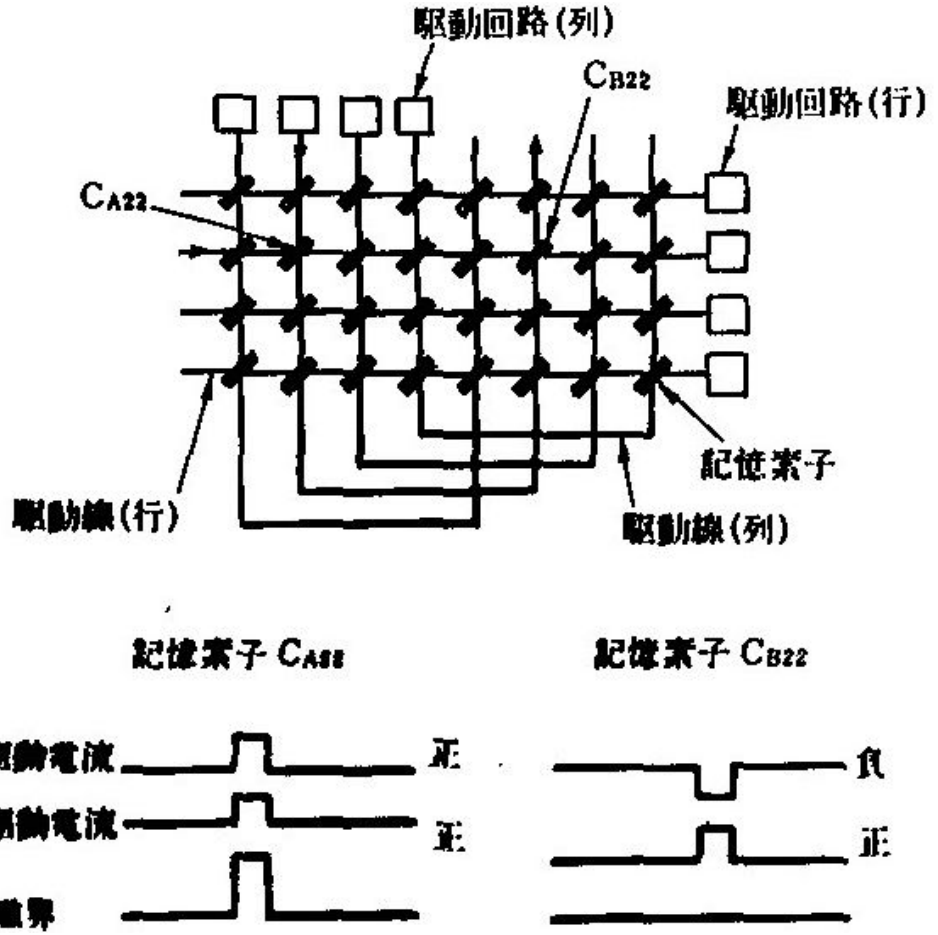
L1とL2が交互に記憶状態になり、data入力とoutputとは必ず切り離される。

Memory

コアメモリー（~1960年代）

1kビットコアメモリー:
50 x 50 x 20 cm, ~100W

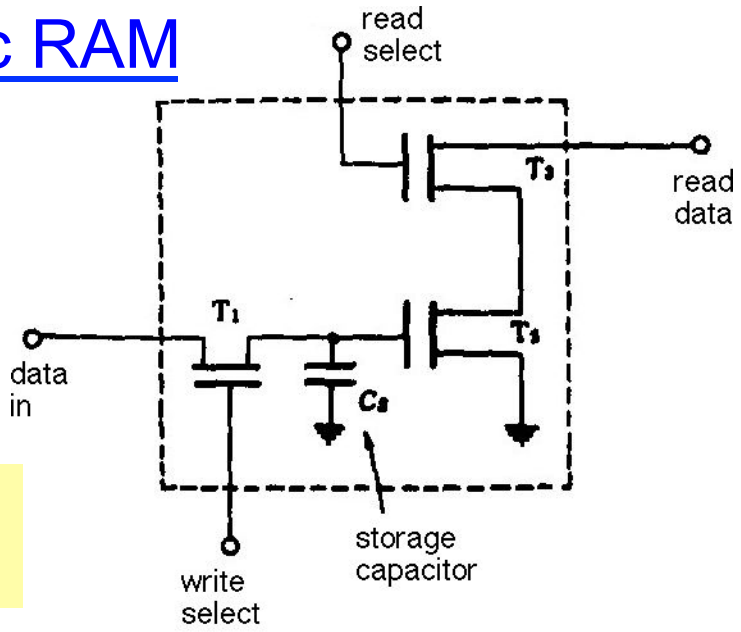




MOS Memory
- SRAM (Static Random Access Memory)
- DRAM (Dynamic Random Access Memory)
- Non-Volatile Memory (EPROM, Flash Memory . . .)
- ROM (Read Only Memory)

:

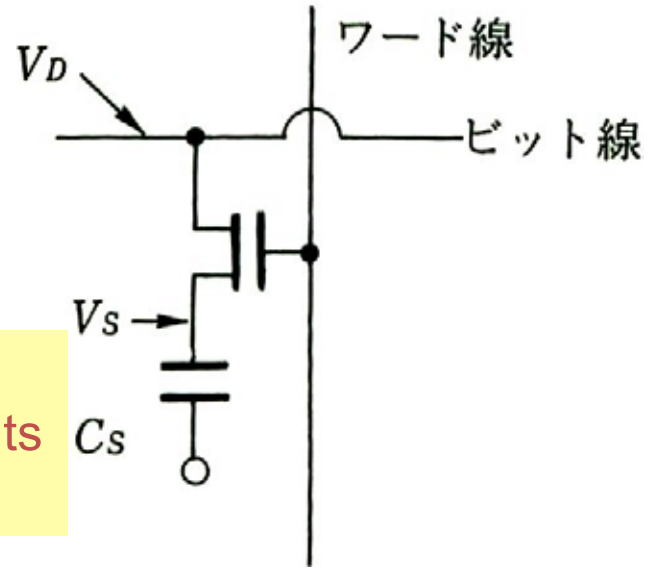
Dynamic RAM (DRAM)



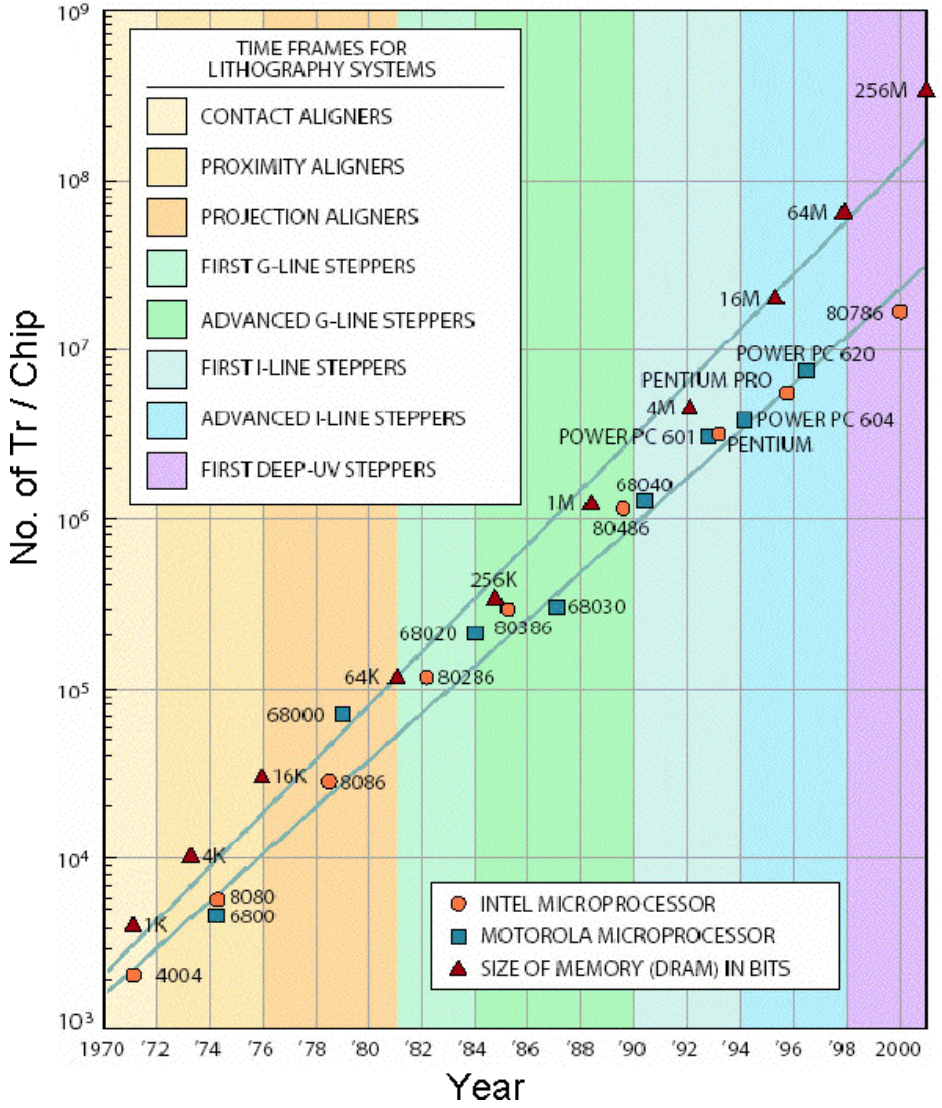
1971 Intel
3 Tr DRAM

インテル社の開発した1Kビット DRAM メモリセル

T_1：データ書き込み用選択トランジスタ
T_2：読み出し用トランジスタ
T_3：データ読み出し用選択トランジスタ
C_S：記憶用キャパシタンス



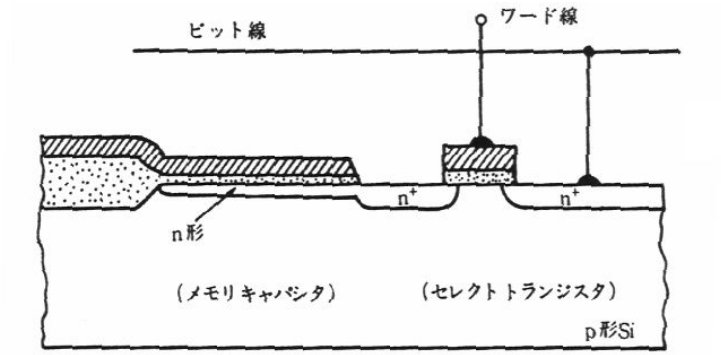
1972
Texas Instruments
1 Tr DRAM

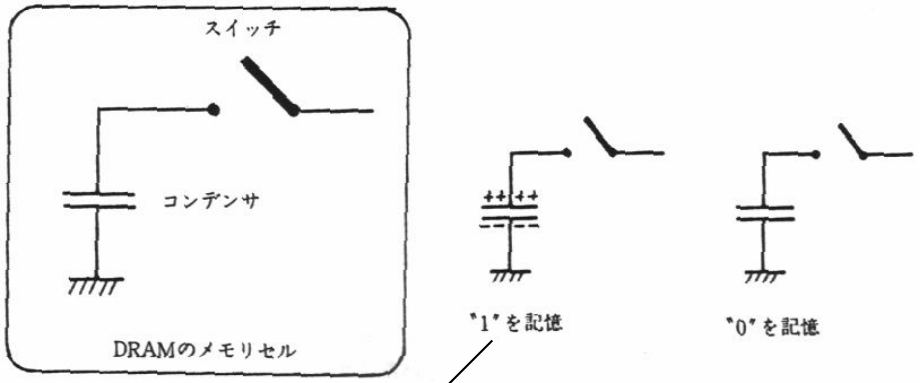


Memory : x1.6/year

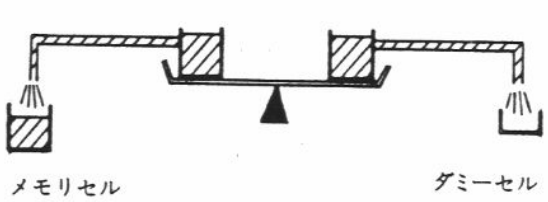
DRAM Refresh

Refreshが~msec毎に必要





少しずつ蓄積電荷が減少。



定期的にチェックして不足分を足す。

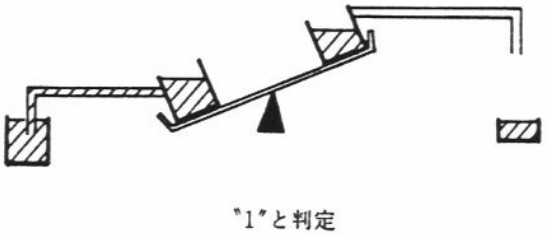


図 I-31 メモリセル用バケツの水位が半分以上か否かを判定する仕掛け

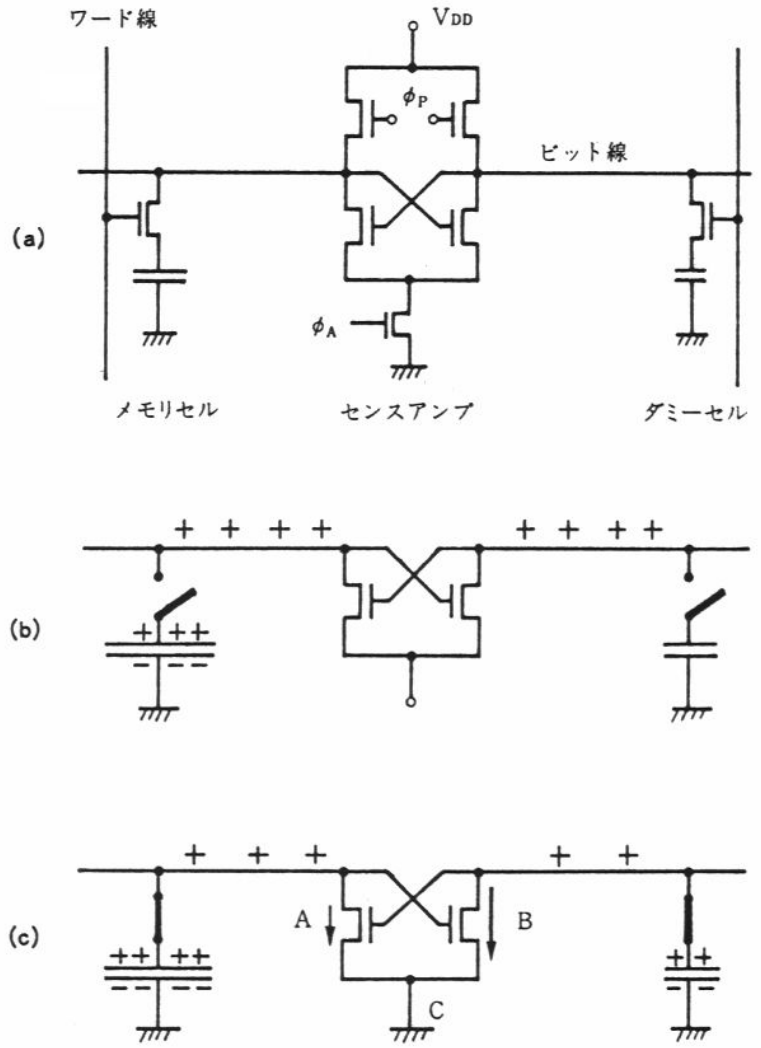
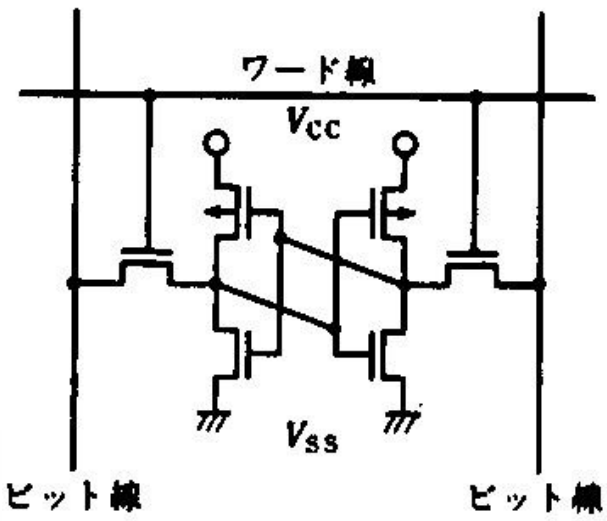
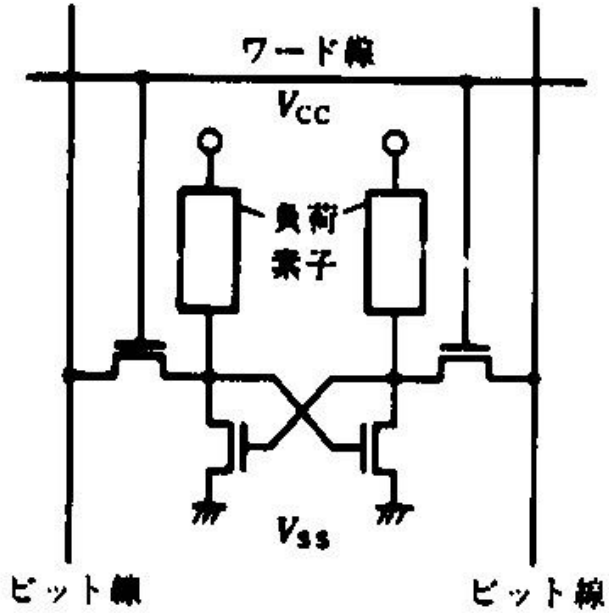


図 I-32 DRAMセンスアンプの動力原理。「1」の書き込まれたメモリセル(5Vに充電されたセル)のデータを読み出す

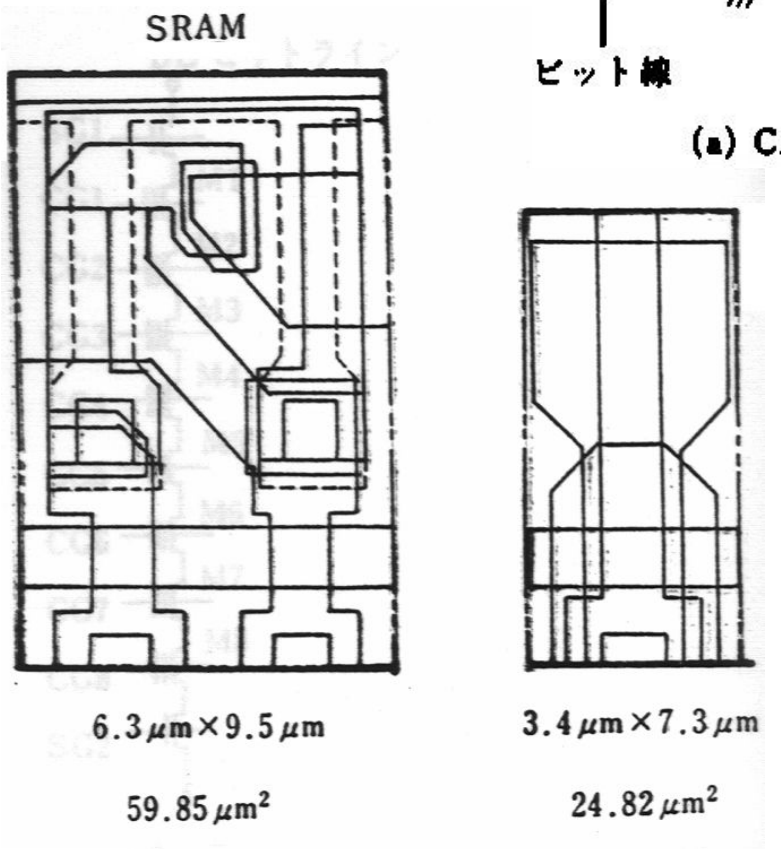
Static RAM (SRAM)



(a) CMOS 6Tr形



(b) 受動負荷素子形



DRAMとSRAMの1ビット当たりの面積比較（1μmルール）

高速アクセス

Refresh動作がいらない。

EEPROM
(Electrically Erasable Programmable ROM)

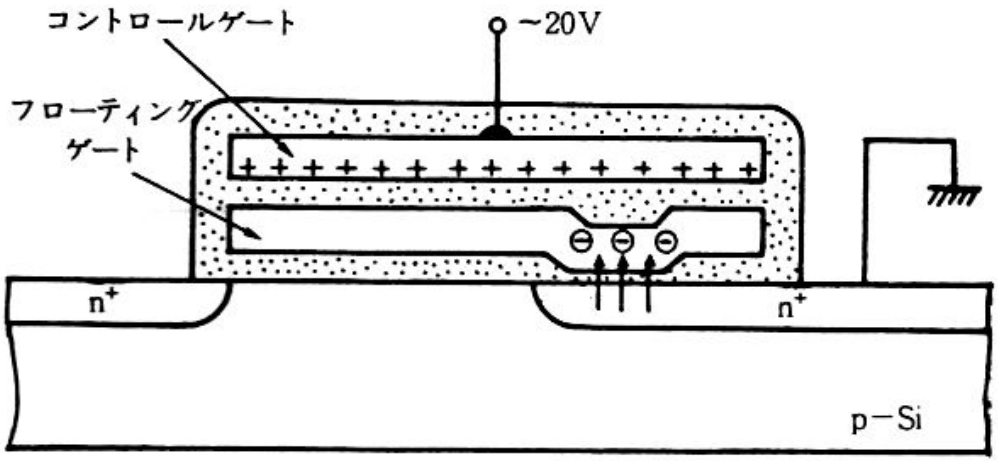


図 I-35 E²PROMセルの断面構造。薄い酸化膜（~100Å）を通して電子をトンネリングさせることにより、フローティングゲートに電子を出し入れし、データを記憶

トンネリング注入

電源を切っても内容が失われない。

書き換え可能。

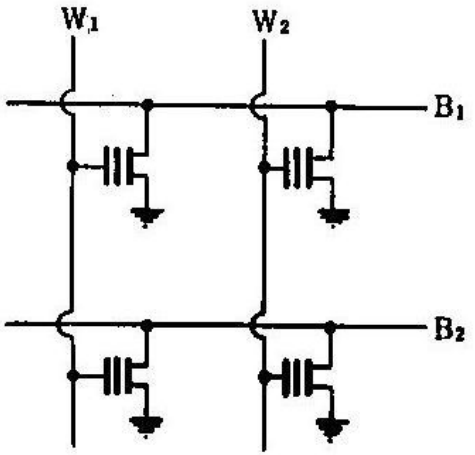
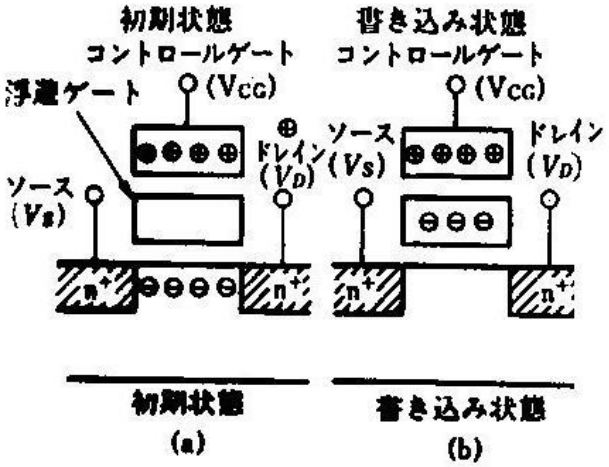
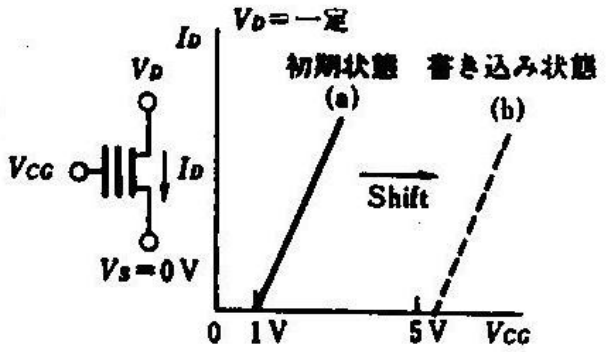


図 1.34 UVEPROMの等価回路



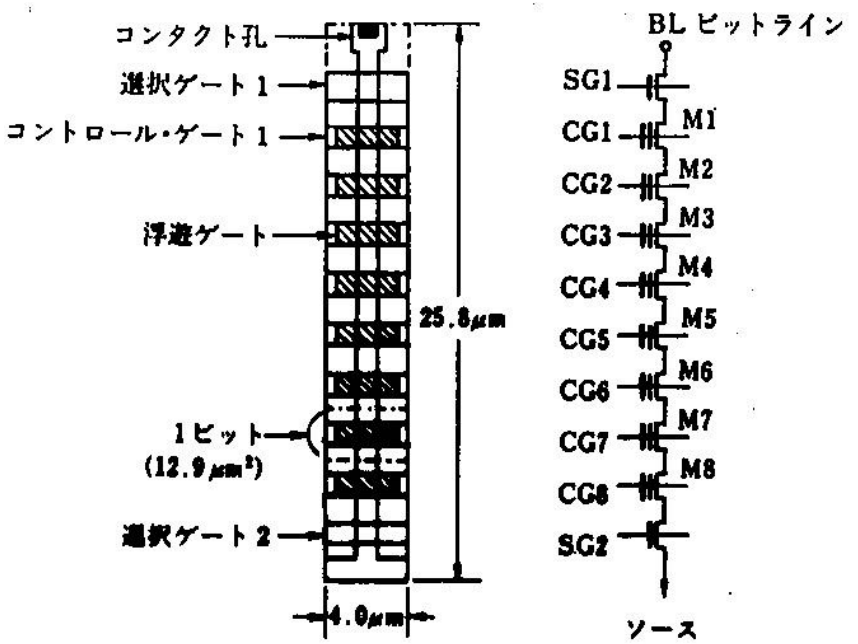


Flash Memory

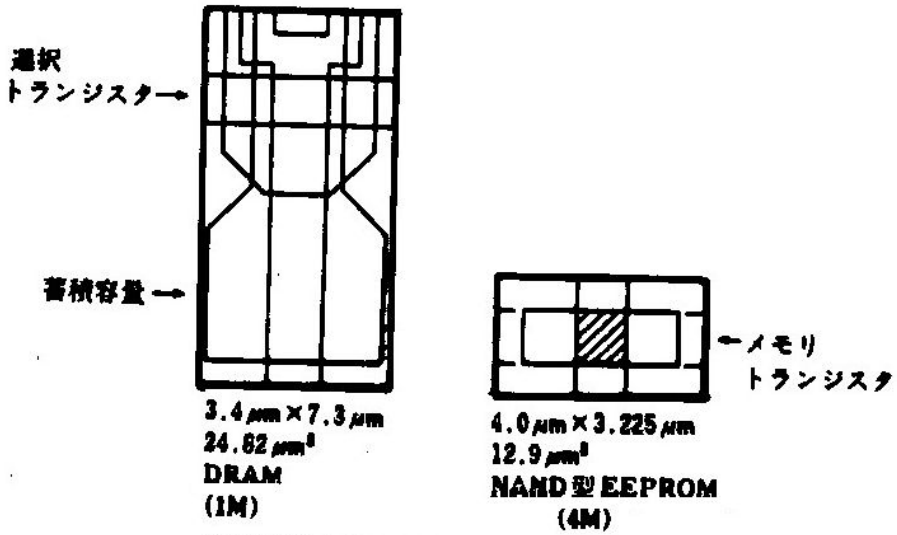
1987 東芝
NAND型EEPROM

一括消去
高密度

ホットエレクトロン注入



NAND型EEPROMの平面図と等価回路



DRAMとNAND型EEPROMセルの1ビット当りの面積比較（1μmルール）

Binary Number

コンピュータの内部では0と1の2つの状態を用いて数を表す。

数の表し方

2 進数	16 進数	10 進数
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15
10000	10	16
10001	11	17

How to express negative number

最も最も単純な方式は数値部と別に、最上位に符号を表すビットを付ける。
（初期のIBMコンピュータ）

5 = '0 0101' --> −5 = '1 0101'

但し、この場合0に2つの表現が出来る。

+0 = '0 0000', −0 = '1 0000'

また、単純に正の数と負の数を足しても正しい結果が出ない。

```
   0 0101  ( 5)
+) 1 0011  (-3)
   1 1000  (-8 ?)
```

How to express negative number (cont.)

1の補数(Ones' Complement)

負の数を表すのに、正の数の表現にNOTを施す(11...1から引く)。

5 = '0 0101' --> -5 = '1 1010'

但し、この場合も0に2つの表現が出来る。

+0 = '0 0000', -0 = '1 1111'

正と負の足し算はキャリー(桁あふれ)を戻して足す事で正しくなる。

```
    0 0101   ( 5)
+)  1 1100   (-3)
(1)0 0001
       1
-----------
    0 0010   (2)
```

[補数(Complement)]
足して桁上がりする数の最小値。
10進数の場合(10の補数):
3の補数は7、25の補数は75。

[減基数の補数(Complement)]
足して桁上がりしない数の最大値。
10進数の場合(9の補数):

How to express negative number (cont.)

2の補数(Two's Complement)

負の数を表すのに、1の補数に1を足す (2^Nから引く)。

+5 = 0 0101 −(1の補数)−> 1 1010

+) 1

(2の補数) 1 1011 = −5

この場合は0は1種類(0 0000)のみ。

足し算は正負に関係なく単純に行なえる(キャリーは無視)。

```
   0 0101  ( 5)
+) 1 1101  (-3)
(1)0 0010  ( 2)
```

ビット数を増やすときは符号を拡張

−5 : 1 011 (符号＋3ビット) −> 1 1011 (符号＋4ビット)

足し算

```
   0 0101   (+5)
+) 0 0001   (+1)
   0 0110   (+6)
```

```
   0 0101   ( +5)
+) 0 1100   (+12)
   1 0001   (-15+32)
        overflow
```

引き算

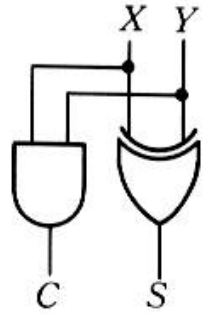
```
   0 0101   (+5)
+) 1 1011   (-5)
(1) 0 0000  ( 0)
```

```
   0 0101   (+5)
+) 1 1000   (-8)
   1 1101   (-3)
```

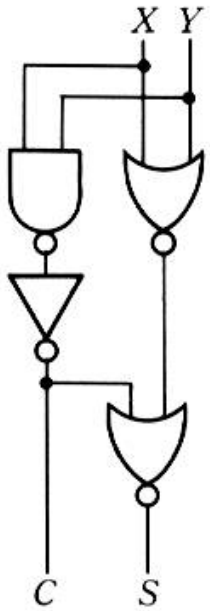
```
   1 1011   ( -5)
+) 1 0100   (-12)
(1) 0 1111  (+15-32)
       underflow
```

足し算器

Half Adder



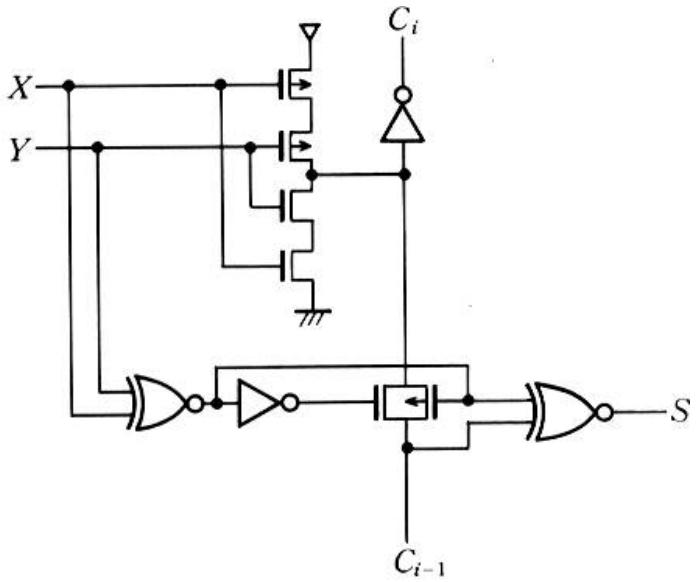
(a) 論理式どおりの論理回路図



(b) CMOS インプリメントを考慮した論理回路図

X Y	S C
0 0	0 0
0 1	1 0
1 0	1 0
1 1	0 1

Full Adder



全加算器の回路例

全加算器の真理値表と論理式

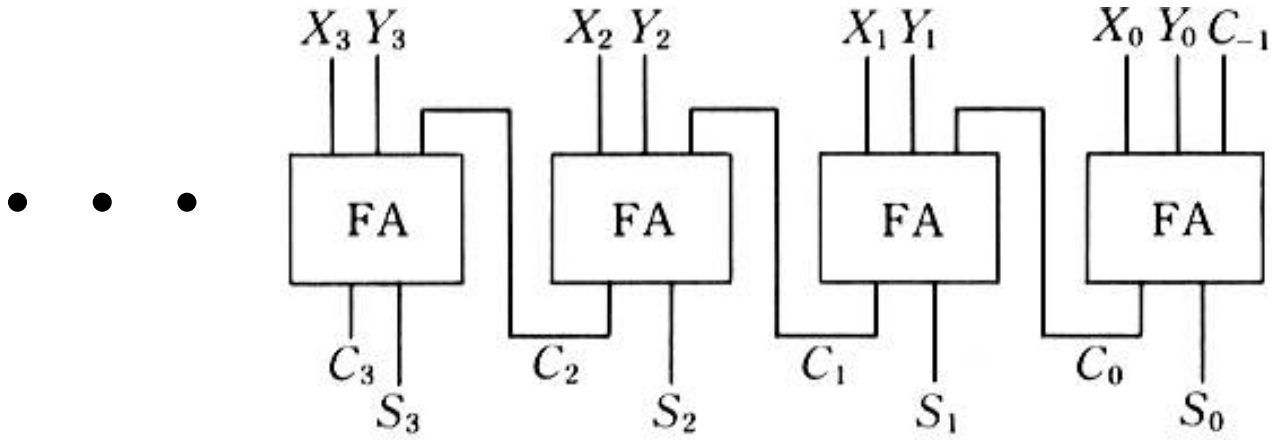
入力			出力	
X	Y	C_{i-1}	S	C_i
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

$S = C_{i-1} \oplus (X \oplus Y)$

$C_i = X * Y + Y * C_{i-1} + X * C_{i-1}$

ケタが増えると

ケタ上がりが増えてどんどん遅くなる！



（FA：Full Adder）

4 ビットリップルキャリーアダー

Carry Look Ahead (CLA)
（桁上げ先見方式）

全加算器の真理値表と論理式

入力			出力	
X	Y	C_{i-1}	S	C_i
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

$S = C_{i-1} \oplus (X \oplus Y)$

$C_i = X * Y + Y * C_{i-1} + X * C_{i-1}$

X, Y がともに“1”のときは下の桁からの桁上げ信号 C_{i-1} の値にかかわらず桁上げ信号 C_i が発生し，X, Y のどちらかが“1”のときは C_{i-1} の値で C_i が決定されているのがわかる。この様子を式で表すと，次のようになる。

$$C_i = G_i + P_i * C_{i-1}$$

ただし，$G_i = X_i * Y_i$

$$P_i = X_i \oplus Y_i$$

ここで G_i は桁上げ生成信号(generator)とよび，P_i は桁上げ伝搬信号(propagator)とよぶが，機能的には半加算器の C および S と全く等しい。

これを用いて各桁の桁上げ信号 C_i を表すと，

$$C_0 = G_0 + P_0 * C_{-1}$$

$$C_1 = G_1 + P_1 * C_0 = G_1 + G_0 * P_1 + P_0 * P_1 * C_{-1}$$

$$C_2 = G_2 + G_1 * P_2 + G_0 * P_1 * P_2 + P_0 * P_1 * P_2 * C_{-1}$$

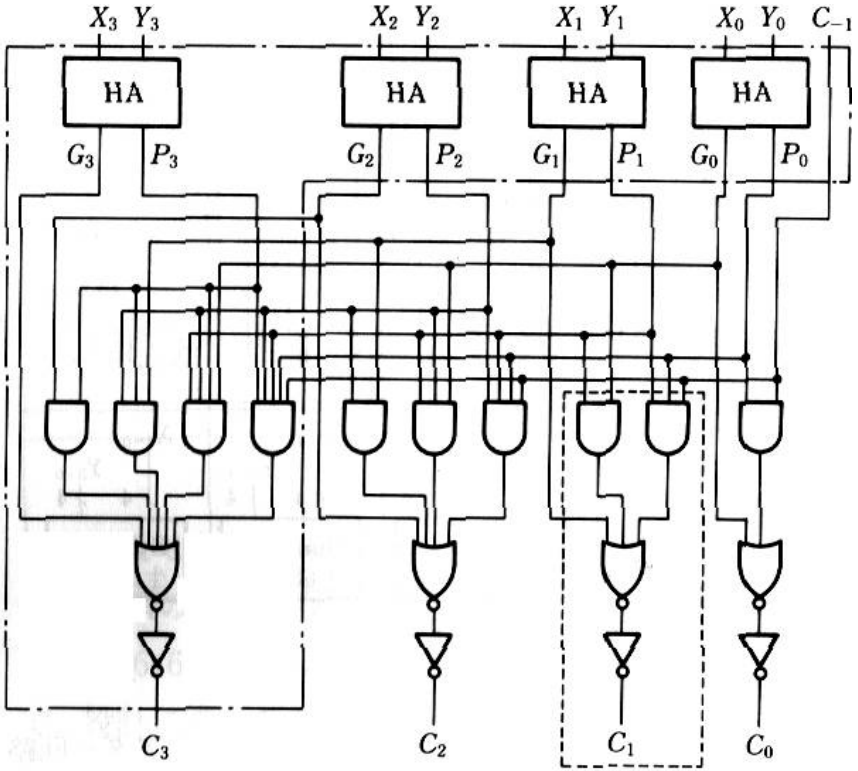
$$C_3 = G_3 + G_2 * P_3 + G_1 * P_2 * P_3 + G_0 * P_1 * P_2 * P_3 + P_0 * P_1 * P_2 * P_3 * C_{-1}$$

$$\cdots$$

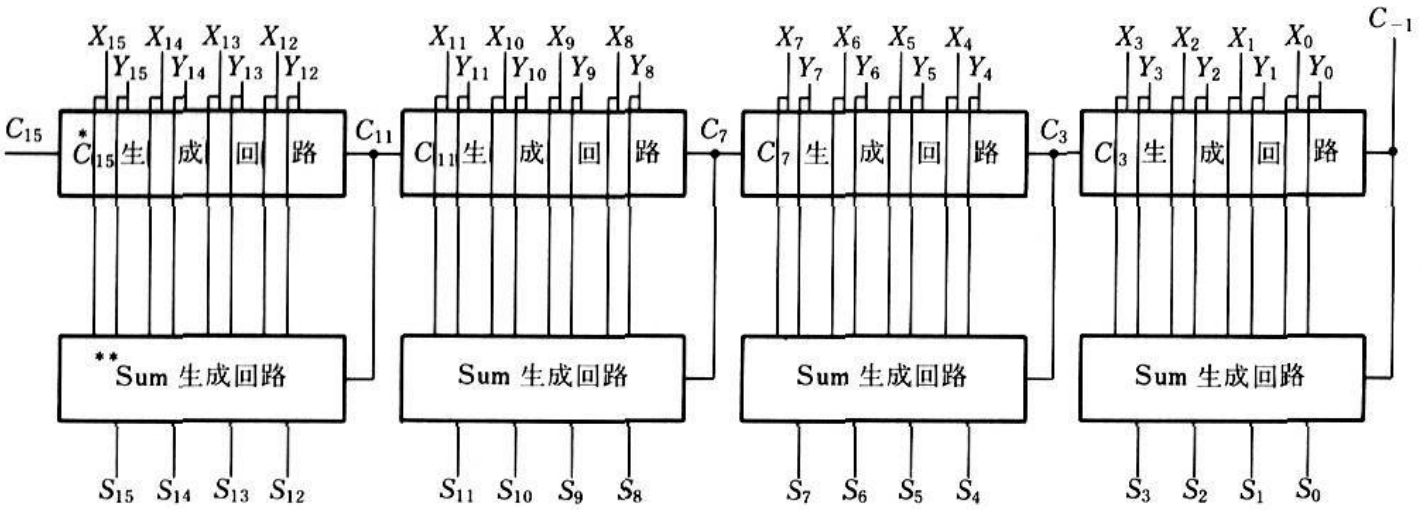
$$C_n = G_n + G_{n-1} * P_n + G_{n-2} * P_{n-1} * P_n + \cdots + G_0 * P_1 * P_2 * \cdots * P_n + P_0 * P_1 * P_2 * \cdots * C_{-1}$$

となり，すべての桁の桁上げ信号は自桁の入力値と最下位の桁上げ信号だけで生成できることがわかる。これを桁上げ先見方式(Carry Look Ahead)略して CLA とよぶ。

CLA



(a) 4 ビット分の回路例

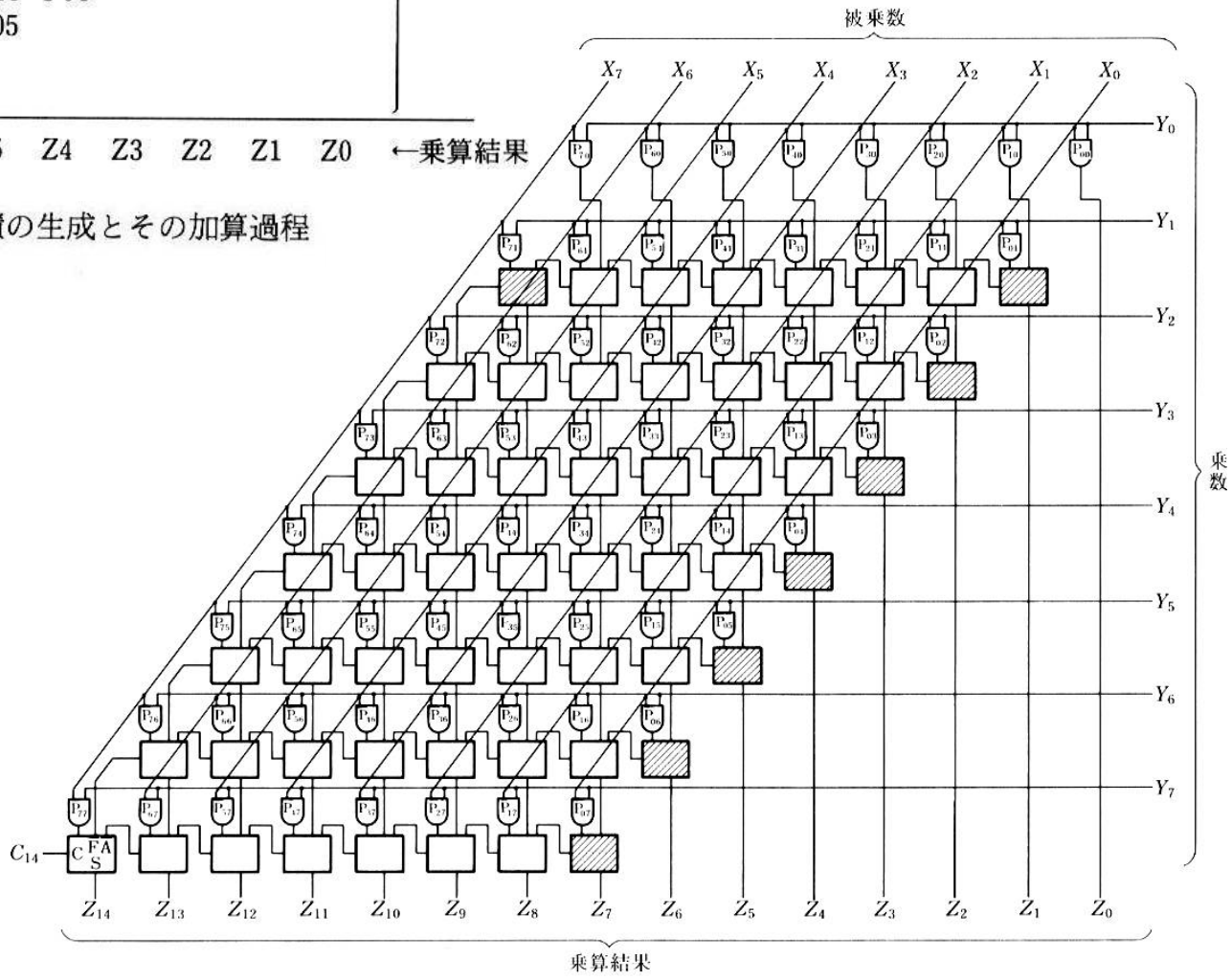


ブロック CLA を用いた 16 ビットアダー回路

乗算器

桁の重み					2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
					↓	↓	↓	↓	↓	↓	↓	↓	
					X7	X6	X5	X4	X3	X2	X1	X0	←被乗数
				×)	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	←乗数
					P70	P60	P50	P40	P30	P20	P10	P00	
				P71	P61	P51	P41	P31	P21	P11	P01		
			P72	P62	P52	P42	P32	P22	P12	P02			
		P73	P63	P53	P43	P33	P23	P13	P03				
	P74	P64	P54	P44	P34	P24	P14	P04					←部分積
P75	P65	P55	P45	P35	P25	P15	P05						
P76	P66	P56	P46	P36	P26	P16	P06						
+)P77	P67	P57	P47	P37	P27	P17	P07						
Z14	Z13	Z12	Z11	Z10	Z9	Z8	Z7	Z6	Z5	Z4	Z3	Z2	Z1	Z0	←乗算結果

8ビット×8ビットの乗算を例にとった部分積の生成とその加算過程
ここで $P_{ij}=X_i * Y_j$(論理積)

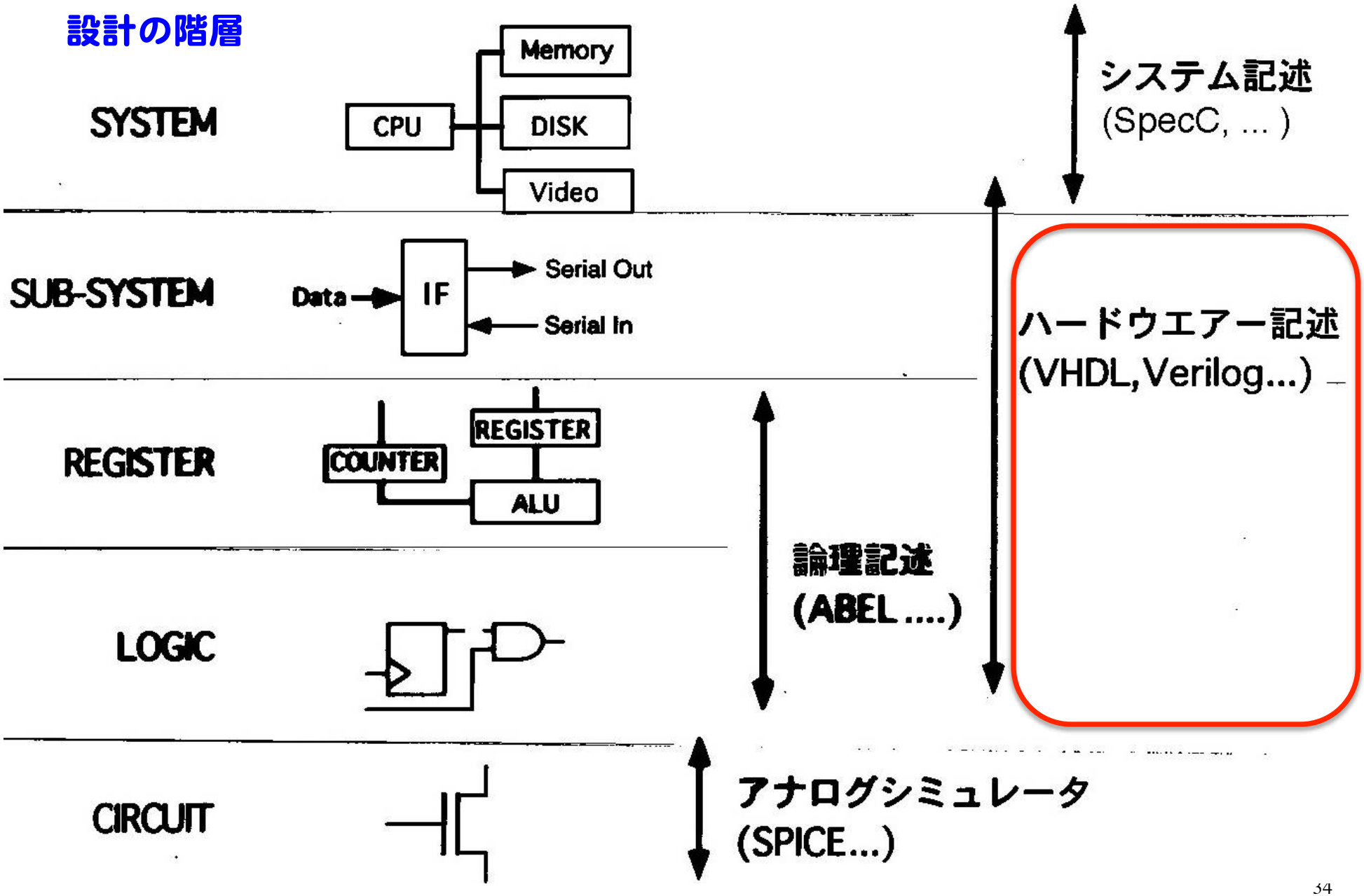


最も基本的な構成の8ビット×8ビット並列乗算器
8×8=64個のANDゲートをおくことにより，すべての部分積が同時に得られる。
ハッチした部分の部分積加算回路は2入力なので半加算器でよい。それ以外は全加算器を用いる。

Divide

- 1サイクルで割り算の出来る簡単な回路はない。
- add/subtract -> shift_left -> . . .
- Initial seed (table) -> Iteration

回路Simulation



Design with Hardware Description Language

利点

- 複雑な設計が可能(> 百万ゲート)
- LSIプロセスから独立した設計 --- 設計資産
- テストパターンの自動化 --- アルゴリズムによる検証
- 開発期間の短縮

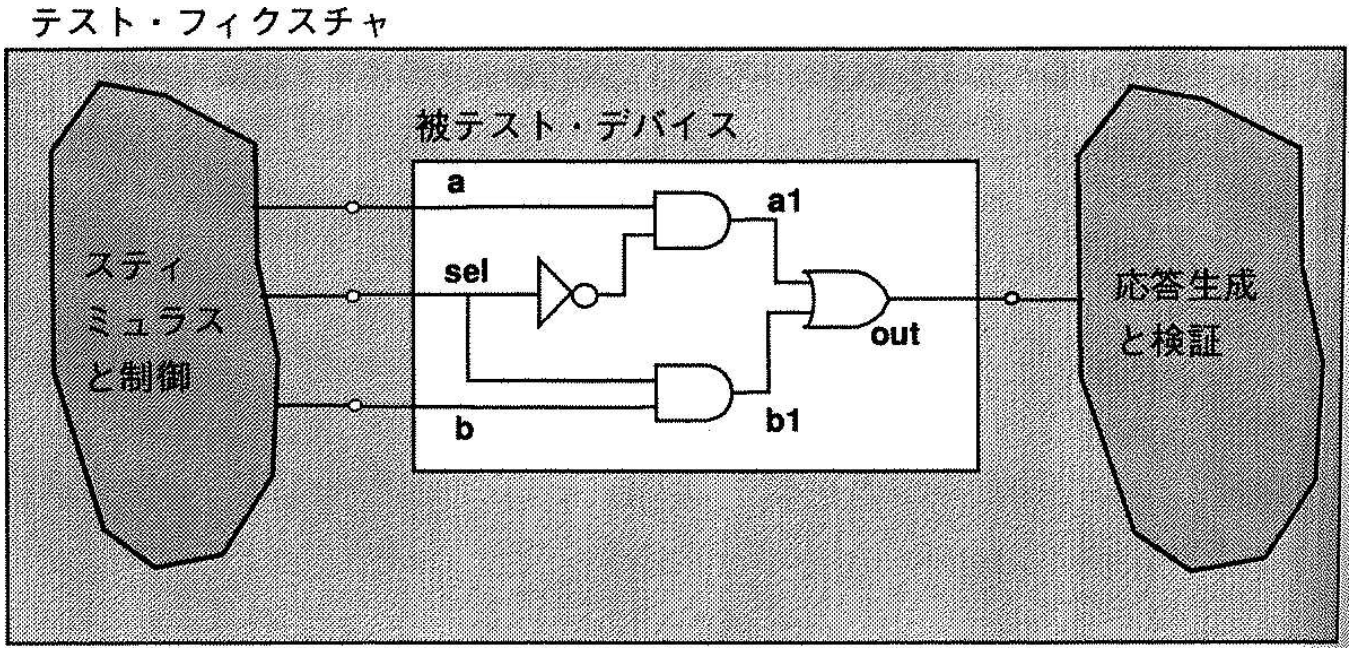
欠点

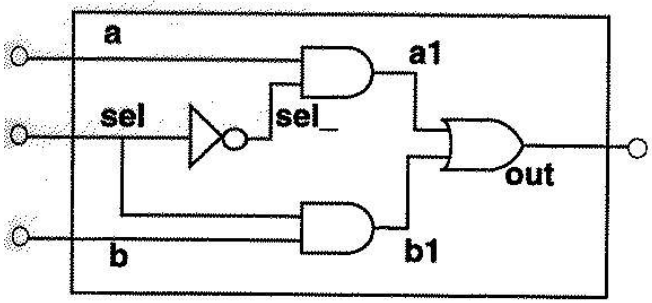
- 技術者が回路の動作を理解できなくなる。
- 性能がコンパイラーに依存(米国製)

Verilog Hardware Description Language

- 1984 Gateway Design Automation社がVerilog を開発。
- やがて'90年頃米国国防相で規格制定されたVHDL(VHSIC Hardware Description Language)と共に業界標準となる。
- 一見C言語に似ているが、通常のプログラム言語との最大の違いは --- 並列記述言語で有るという点。
- プログラム中のどこに書こうが、すべての回路は同時に動く。

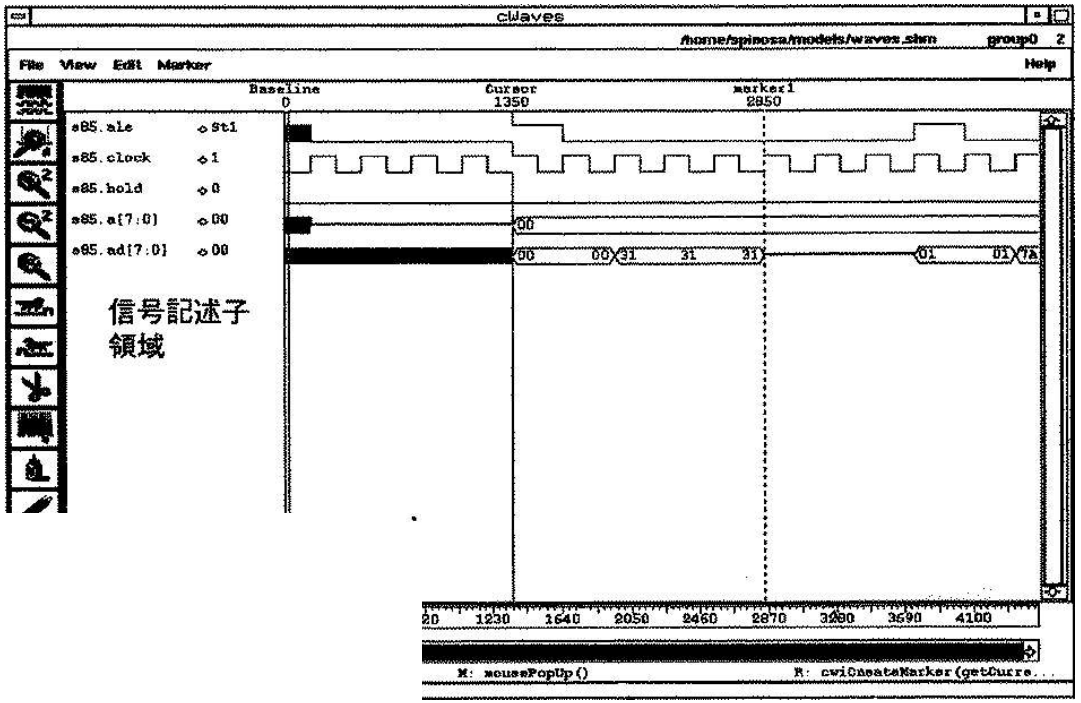
VerilogによるSimulation例





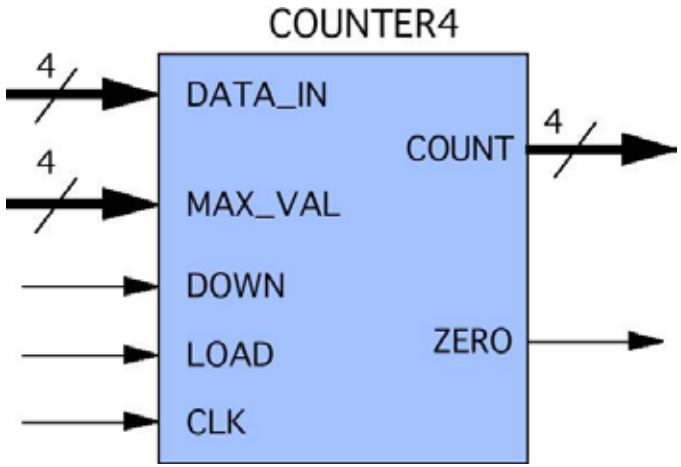

```
module MUX2_1 (out,a,b,sel);

  // ポートの宣言
  output out;
  input a,b,sel;

  // ネットリスト
  not (sel_, sel);
  and (a1, a, sel_);
  and (b1, b, sel);
  or (out, a1, b1);

endmodule
```

Simulation of Circuit Operation



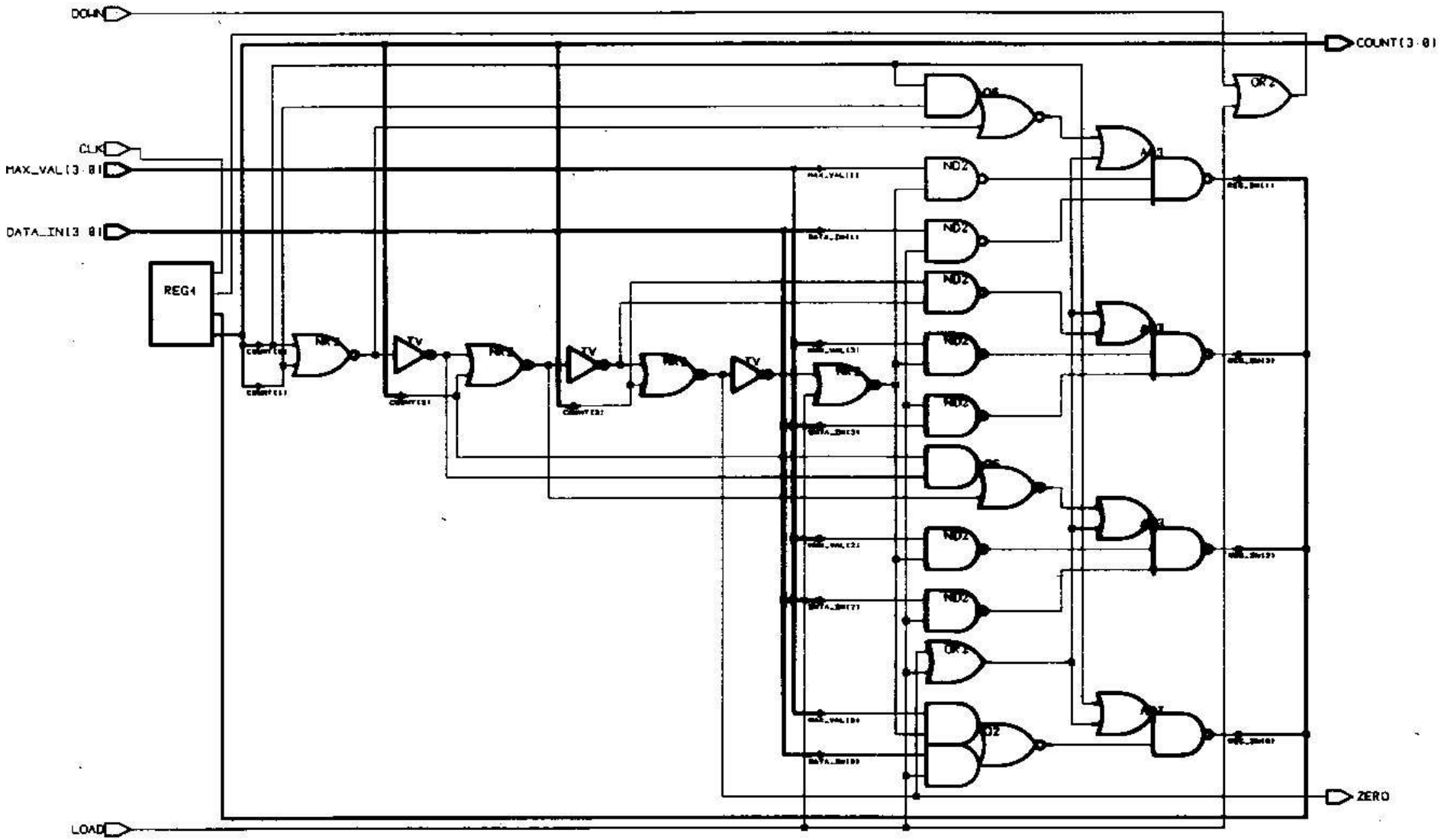

```
module testfixture;
reg a, b, sel;
//インスタンス MUX
  MUX2_1 mux (out, a, b, sel);
//スティミュラスを加える
initial
begin
  a = 0; b = 1; sel = 0;
  #5 b = 0;
  #5 b = 1; sel = 1;
  #5 a = 1;
  #5 $finish;
end
//結果を表示する
initial
$monitor($time,"out=%b a=%b b=%b sel=%b",out,a,b,sel);
endmodule
```

Logic Synthesis with Verulog

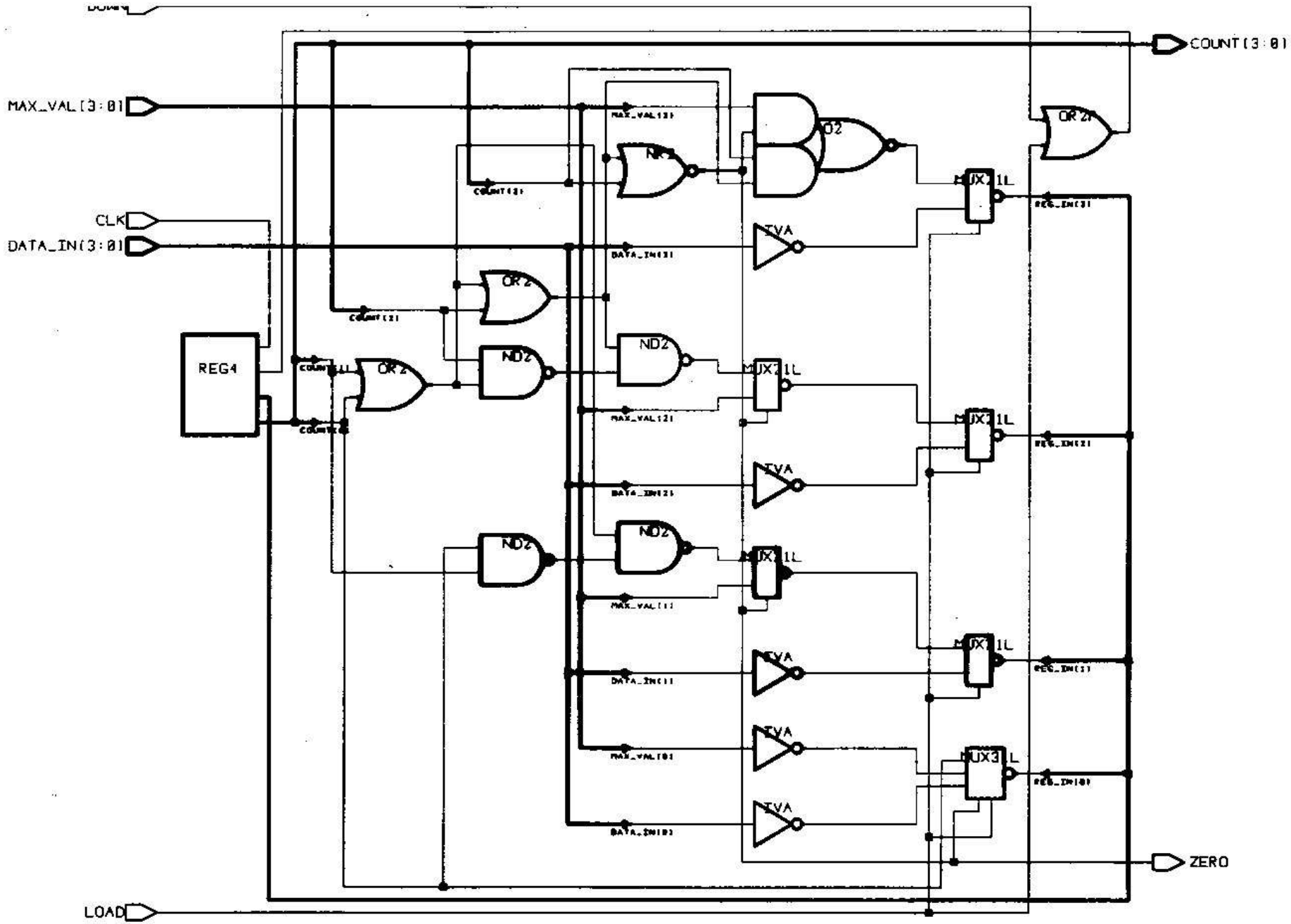



```
module COUNTER4 (DATA_IN,CLK,LOAD,DOWN,MAX_VAL,COUNT,ZERO );
input [3:0] DATA_IN;
input CLK, LOAD, DOWN;
input [3:0] MAX_VAL;
output [3:0] COUNT;
output ZERO;

wire [3:0] REG_IN, NEW_COUNT;
wire REG_LOAD;

assign    ZERO = ( COUNT == 4'b0 );
                          //COUNTゼロでZERO=1

assign    NEW_COUNT = ( COUNT == 4'b0 ) ? MAX_VAL : (COUNT - 1'b1);
                          //COUNTが0ならMAX_VALそれ以外は-1

assign  REG_IN = ( LOAD ) ? DATA_IN : NEW_COUNT;
                          //LOADでDATA_INをそれ以外はNEW_COUNT

assign  REG_LOAD = ( LOAD | DOWN );
                          //LOADまたはDOWNでREG_LOAD=1

REG4   REG_BLK1 ( REG_IN, CLK, REG_LOAD, COUNT );
                          //4ビットRegisterの参照

endmodule
```

ゲート数最小で最適化した場合



遅延時間最小で最適化した場合



Verilogの例

```
always @(posedge clk)   --- clkが変化したら以下の動作を行う
        begin
        if (reset) waiting_for_data <= #1 1; --- resetが出ていたらwaiting_for_data
        else if ( ~wait_cycle)                  を#1後に'1'にする
                begin
                if ( bit_count == 33 )   --- bit_countが33になったら実行
                        waiting_for_data <= #1 1;
                else if (serial_out == 1)
                        waiting_for_data <= #1 0;
                end
        end
//bit count                                  同時に実行される
always @(posedge clk)
        begin
        if (reset | waiting_for_data )
                bit_count <= #1 0;
        else if ( ~wait_cycle )
                bit_count <= #1 bit_count +1; --- clkが変化する毎にbit_countを
        end                                      '1ずつ増やす
```