第2回 ハードウェア記述言語
～回路のRTL記述～

中野 秀洋

LSI 設計段階と表現

抽象度　記述量

高い　少ない

低い　多い

仕様設計

RTL設計
（機能設計）

ゲートレベル設計
（論理設計）

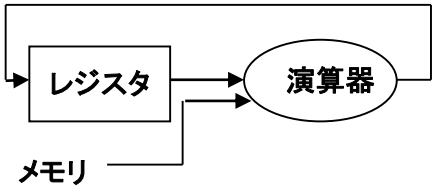
トランジスタレベル設計
（回路設計）

レイアウト設計

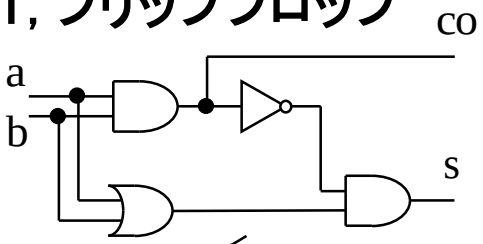
命令セット，レジスタセット
データ、アドレス、制御信号
パイプライン、割込み、キャッシュ

```
LD  GR1, 100, GR7
ADD GR1, 200, GR7
```

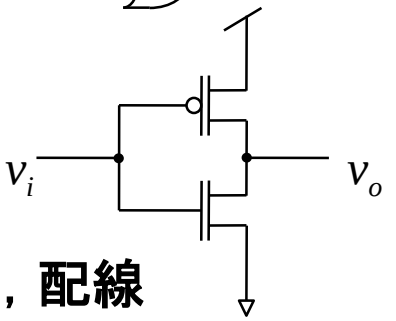
組合せ回路とレジスタ



AND, OR, NOT, フリップフロップ



トランジスタ



チップに配置，配線

RTL設計とゲートレベル設計

記述のレベル	RTL (Register Transfer Level)	ゲートレベル (Gate Level)
基本構成要素	組み合わせ回路 レジスタ	AND, OR, NOT フリップフロップ
抽象度	高い	低い
記述量	少ない	多い
論理回路合成	自動（論理合成ツール）	手動（カルノー図等）
簡略化・最適化	論理合成ツールの性能次第	設計者の腕次第
大規模回路設計	容易	困難

RTL設計の利点

- そのままシミュレーションが可能
- 論理回路はツールが自動的に合成
- 大規模回路設計が容易

動作記述と構造記述

- 動作記述
  - 回路の動作を式と代入文によって記述
    例：assign文、always文（後述）
  - 回路の動作をそのまま記述
  - RTL設計で多く使用
- 構造記述
  - 回路の構造をポートと信号の接続によって記述
    例：instance文
  - 回路の動作は下位moduleの内部で定義
  - ゲートレベル設計で多く使用

例(1)

- 動作記述のみ

```
/* 半加算器のmoduleの定義 */
module half_add(a, b, co, s);
  /* input port */
  input a, b;
  /* output port */
  output co, s;
  /* wire */
  wire w0, w1, w2;

  /* assign文 */
  assign w0 = a & b;
  assign w1 = ~w0;
  assign w2 = a | b;
  assign s = w1 & w2;
  assign co = w0;
endmodule
```

- 動作記述と構造記述のハイブリッド

```
/* 全加算器のmoduleの定義 */
module full_add(a, b, ci, co, s);
  /* input port */
  input a, b, ci;
  /* output port */
  output co, s;
  /* wire */
  wire w0, w1, w2;

  /* instance文 */
  half_add a0
   (.a(a), .b(b), .co(w1), .s(w0));
  half_add a1
   (.a(w0), .b(ci), .co(w2), .s(s));
  /* assign文 */
  assign co = w1 | w2;
endmodule
```

例(2)

```
/* 4ビット加算器のmoduleの定義 */
module add4(a, b, ci, co, s);
  /* input port */
  input  [3:0] a, b;
  input        ci;
  /* output port */
  output       co;
  output [3:0] s;
  /* wire */
  wire w0, w1, w2;
  /* instance文 */
  full_add a0
   (.a(a[0]), .b(b[0]), .ci(ci),
              .co(w0), .s(s[0]));
  full_add a1
   (.a(a[1]), .b(b[1]), .ci(w0),
              .co(w1), .s(s[1]));
  full_add a2
   (.a(a[2]), .b(b[2]), .ci(w1),
              .co(w2), .s(s[2]));
  full_add a3
   (.a(a[3]), .b(b[3]), .ci(w2),
              .co(co), .s(s[3]));
endmodule
```

- 構造記述のみ

前回の回路設計

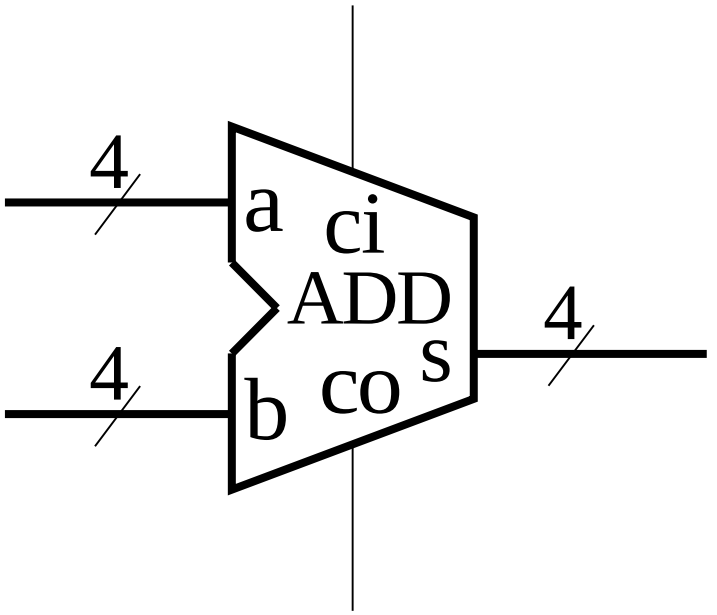
全てゲートレベル設計

ゲートレベル設計の半加算器
➔加算器の基本構成要素

マルチプレクサや
コンパレータも同様

4bit加算器のRTL設計

```
/* 4ビット加算器のmoduleの定義 */
module add4(a, b, ci, co, s);
  /* input port */
  input  [3:0] a, b;
  input        ci;
  /* output port */
  output       co;
  output [3:0] s;
  /* assign文 */
  assign { co, s } = a + b + ci;
endmodule
```



ゲートレベル設計した回路と動作が同じ回路を合成可能
（論理回路の構成が同じになるとは限らない）

RTL設計の利点

- そのままシミュレーションが可能
- 論理回路はツールが自動的に合成
- 大規模回路設計が容易

always文

- always文・・・複雑な動作を記述する際に使用

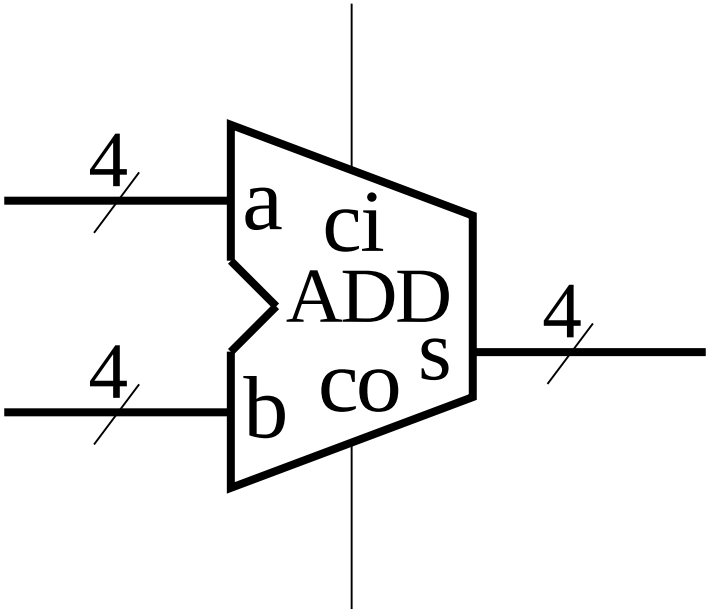
```
always@ ( センシティビティリスト ) begin
   文
end
```

センシティビティリストの
いずれかの信号が
変化したとき文を評価

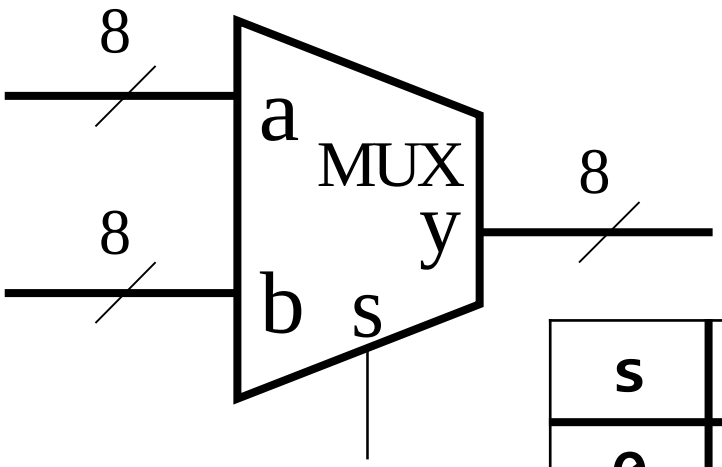
- 例）4bit加算器

```
module add4(a, b, ci, co, s);
  input  [3:0] a, b;
  input        ci;
  output       co;
  output [3:0] s;
  reg          co;
  reg    [3:0] s;


  always@(a or b or ci) begin
    {co, s} <= a + b + ci;
  end
endmodule
```

always文で
代入する信号は
regで宣言



8bitの2入力マルチプレクサ



s	y
0	a
1	b

```
module mux2(a, b, s, y);
  input  [7:0] a, b;
  input        s;
  output [7:0] y;
  reg    [7:0] y;

  always@(a or b or s) begin
    if(s == 1'b0) begin
      y <= a;
    end else begin
      y <= b;
    end
  end
endmodule
```

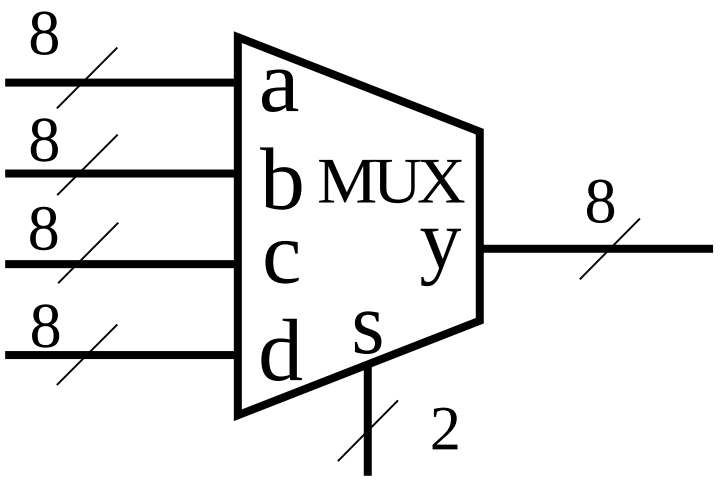
条件が真のとき文1

条件が偽のとき文2

```
if 条件式
  文1
else
  文2
```

2つの8bitの入力の中から1つを選択して出力

8bitの4入力マルチプレクサ




```
module mux4(a, b, c, d, s, y);
  input  [7:0] a, b, c, d;
  input  [1:0] s;
  output [7:0] y;
  reg    [7:0] y;

  always@(a or b or c or d or s) begin
    case(s)
      2'b00:   y <= a;
      2'b01:   y <= b;
      2'b10:   y <= c;
      2'b11:   y <= d;
      default: y <= 8'bxxxxxxxx;
   endcase
  end
endmodule
```

値が一致した文を実行

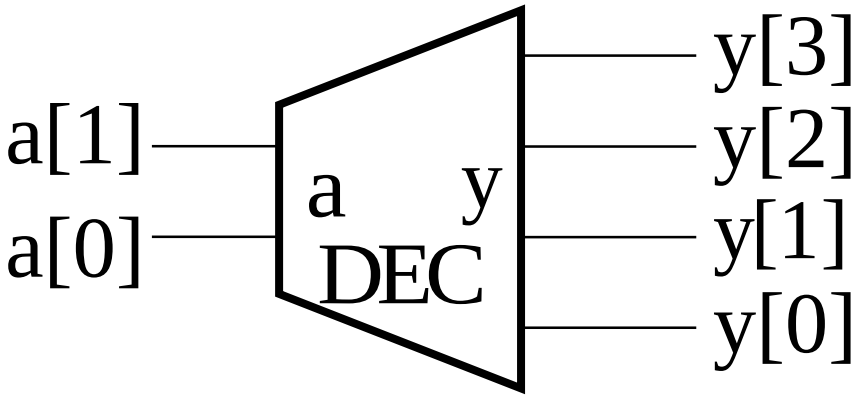
値が一致するものがないときこの文を実行

```
case(式)
  値1: 文;
  値2: 文;
  ...
  default: 文;
endcase
```

x は不定値を表す定数

4つの8bitの入力の中から1つを選択して出力

デコーダ




```
module dec(a, y);
  input  [1:0] a;
  output [3:0] y;
  reg    [3:0] y;

  always@(a) begin
    case(a)
      2'b00 :  y <= 4'b0001;
      2'b01 :  y <= 4'b0010;
      2'b10 :  y <= 4'b0100;
      2'b11 :  y <= 4'b1000;
      default: y <= 4'bxxxx;
    endcase
  end
endmodule
```

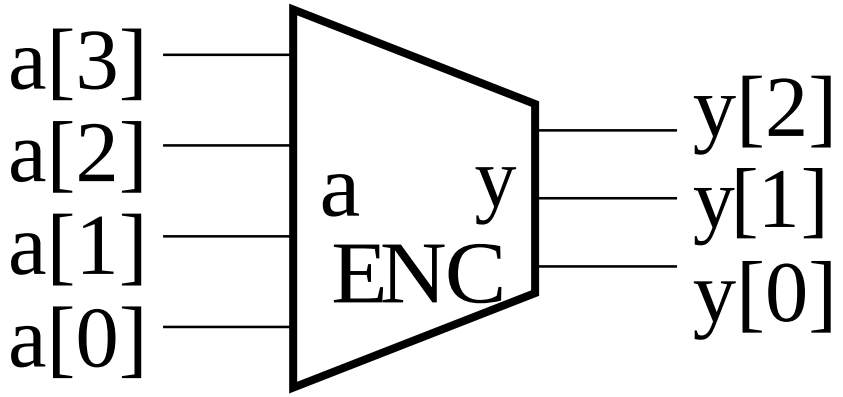
a1	a0	y3	y2	y1	y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

入力された2進数の値と

対応する番号のポートのみから1を出力

エンコーダ

a3	a2	a1	a0	y2	y1	y0
x	x	x	1	1	0	0
x	x	1	0	1	0	1
x	1	0	0	1	1	0
1	0	0	0	1	1	1
0	0	0	0	0	0	0

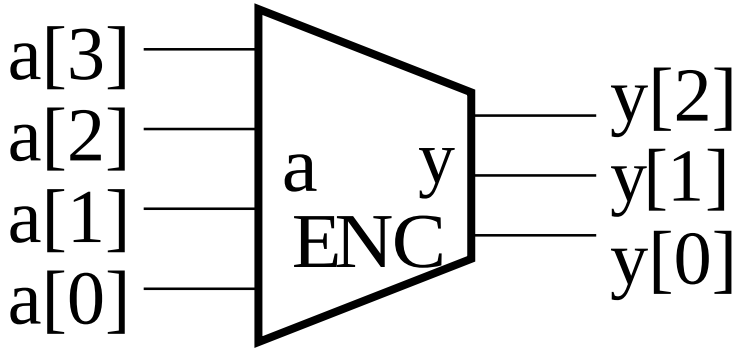


入力がある信号線の番号と対応する2進数を出力

複数の入力があるときは小さい番号の入力が優先

y[2]は入力が0でなければ1を出力

エンコーダ（記述例１）



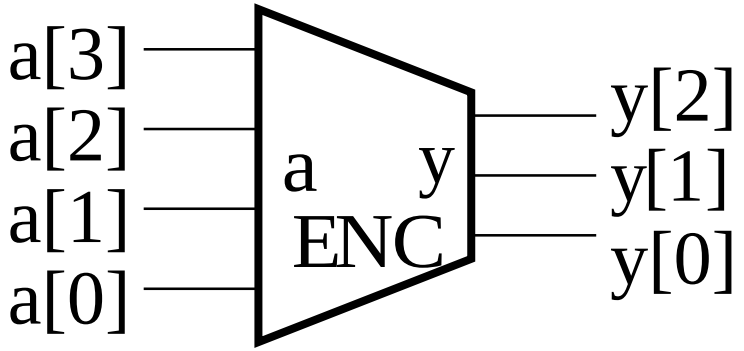
a3	a2	a1	a0	y2	y1	y0
x	x	x	1	1	0	0
x	x	1	0	1	0	1
x	1	0	0	1	1	0
1	0	0	0	1	1	1
0	0	0	0	0	0	0

```
module enc(a, y);
  input  [3:0] a;
  output [2:0] y;
  reg    [2:0] y;

  always@(a) begin
    case(a)
      4'b0001, 4’b0011, 4’b0101, 4’b0111,
      4’b1001, 4’b1011, 4’b1101, 4’b1111:
        y <= 3’b100;
      4’b0010, 4’b0110, 4’b1010, 4’b1110:
        y <= 3’b101;
      4’b0100, 4’b1100:
        y <= 3’b110;
      4’b1000:
        y <= 3’b111;
      default:
        y <= 3’b000;
    endcase
  end
endmodule
```

複数の条件を
まとめて書く場合

```
case(式)
  値1, 値2: 文1;
  値3, 値4: 文2;
  ...
  default: 文;
endcase
```

エンコーダ（記述例2）



a3	a2	a1	a0	y2	y1	y0
x	x	x	1	1	0	0
x	x	1	0	1	0	1
x	1	0	0	1	1	0
1	0	0	0	1	1	1
0	0	0	0	0	0	0

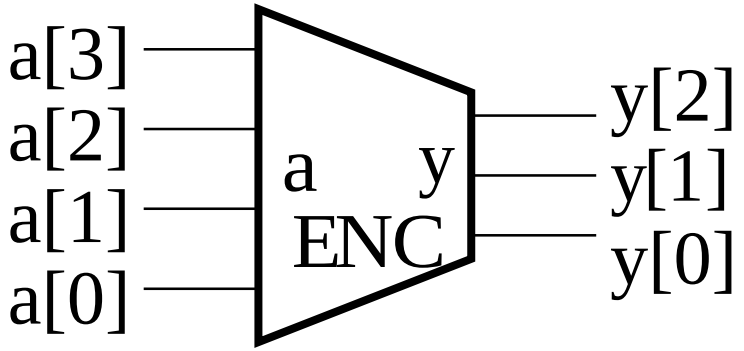
```
module enc(a, y);
  input  [3:0] a;
  output [2:0] y;
  reg    [2:0] y;

  always@(a) begin
    casex(a)
      4'b???1: y <= 3'b100;
      4'b??10: y <= 3'b101;
      4'b?100: y <= 3'b110;
      4'b1000: y <= 3'b111;
      default: y <= 3'b000;
    endcase
  end
endmodule
```

式に不定値のビットがある場合、そのビットの比較は行わない

不定値のビットは「？」で指定

```
casex(式)
  値1: 文;
  値2: 文;
  ...
  default: 文;
endcase
```

エンコーダ（記述例３）



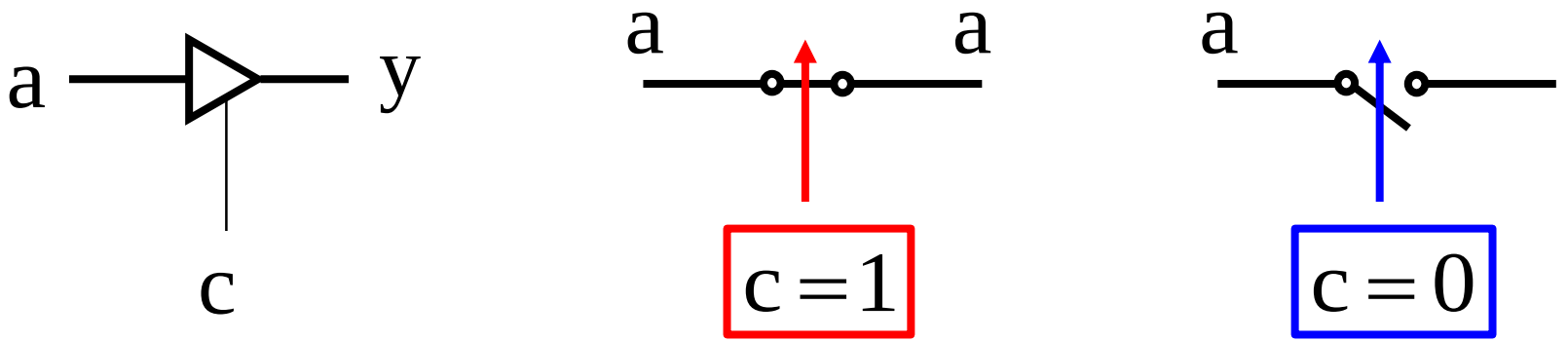
a3	a2	a1	a0	y2	y1	y0
x	x	x	1	1	0	0
x	x	1	0	1	0	1
x	1	0	0	1	1	0
1	0	0	0	1	1	1
0	0	0	0	0	0	0

```
module enc(a, y);
  input  [3:0] a;
  output [2:0] y;
  reg    [2:0] y;

  always@(a) begin
    if(a[0] == 1'b1) begin
      y <= 3'b100;
    end else if(a[1] == 1'b1) begin
      y <= 3'b101;
    end else if(a[2] == 1'b1) begin
      y <= 3'b110;
    end else if(a[3] == 1'b1) begin
      y <= 3'b111;
    end else begin
      y <= 3'b000;
    end
  end
endmodule
```

条件を上から順に判定
最初に満たした条件が優先

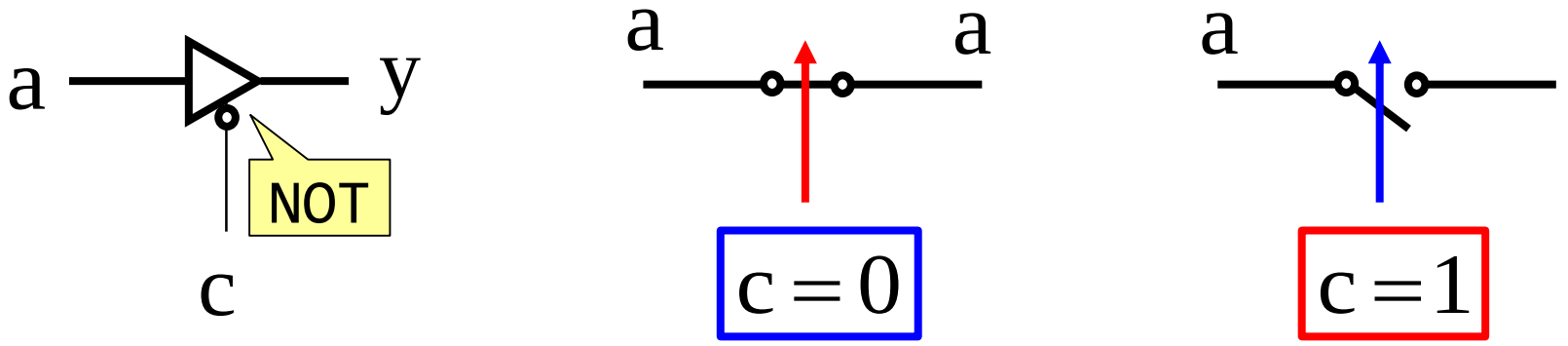
トライステートバッファ

正論理のトライステートバッファ



c が 1 のとき　a を y に接続
c が 0 のとき　y は開放

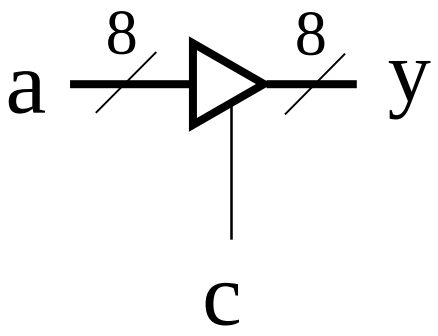
負論理のトライステートバッファ



c が 0 のとき　a を y に接続
c が 1 のとき　y は開放

8bitトライステートバッファ

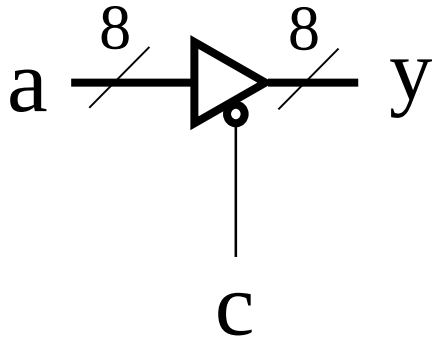
正論理のトライステートバッファ




```
module tribuf(a, c, y);
  input  [7:0] a;
  input        c;
  output [7:0] y;

  assign y = (c == 1'b1) ? a : 8'bzzzzzzzz;
endmodule
```

(条件) ? 文1 : 文2 ;
条件が真のとき文1、偽のとき文2

負論理のトライステートバッファ

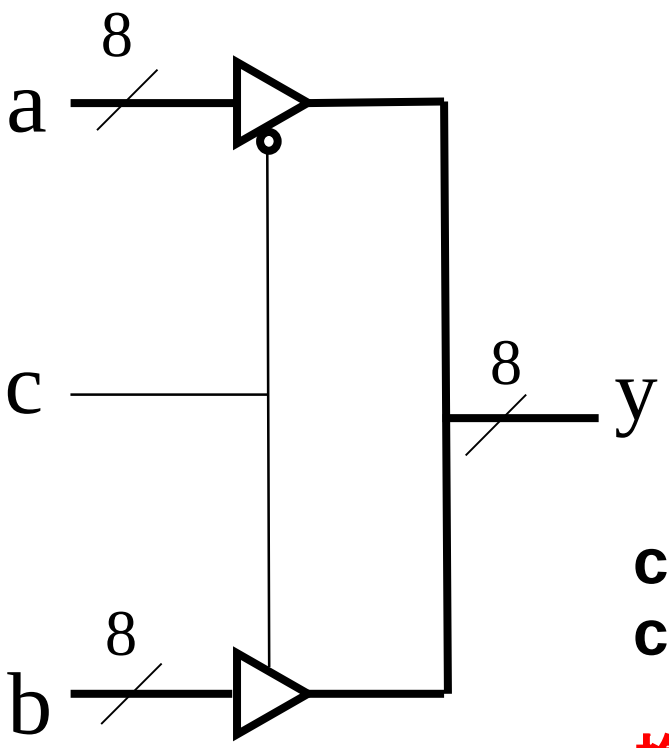



```
module tribuf(a, c, y);
  input  [7:0] a;
  input        c;
  output [7:0] y;

  assign y = (c == 1‘b0) ? a : 8'bzzzzzzzz;
endmodule
```

z はハイインピーダンス(開放)を表す定数

8bit信号選択回路

信号選択回路

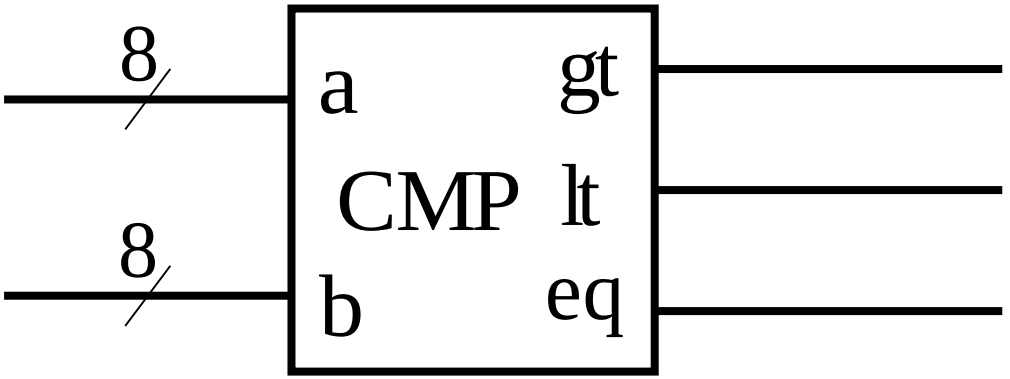



```
module tribuf(a, b, c, y);
  input  [7:0] a, b
  input        c;
  output [7:0] y;

  assign y = (c == 1'b0)
           ? a : 8'bzzzzzzzz;
  assign y = (c == 1'b1)
           ? b : 8'bzzzzzzzz;
endmodule
```

c が 1 のとき a を y に接続 (b は接続しない)
c が 0 のとき b を y に接続 (a は接続しない)

複数の信号から1つの信号を選択して出力
（マルチプレクサよりも回路が簡単）

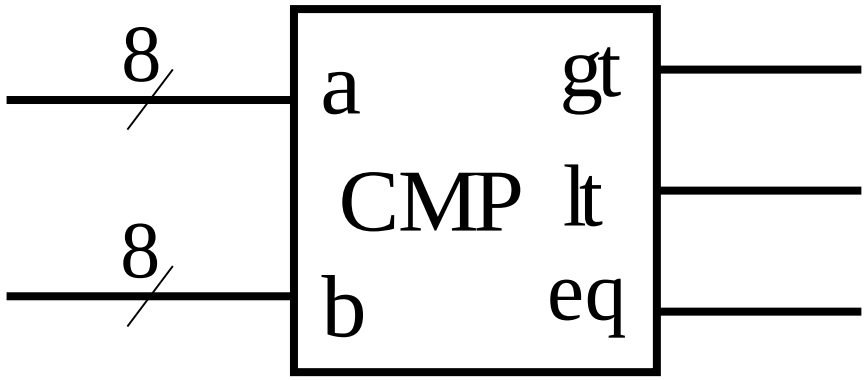
コンパレータ



	gt	lt	eq
1	a > b	a < b	a == b
0	a <= b	a >= b	a != b

入力された値の大小を比較して結果を出力

コンパレータ



if 文の条件を満たすとき
if 文の中に記述された値を出力

if 文の条件を満たさないとき
if 文の外に記述された値を出力
（デフォルト値を出力）

```
module cmp(a, b, gt, lt, eq);
  input  [7:0] a, b;
  output       gt, lt, eq;
  reg          gt, lt, eq;

  always@(a or b) begin
    gt <= 1'b0;
    lt <= 1'b0;
    eq <= 1'b0;
    if(a > b) begin
      gt <= 1'b1;
    end else if(a < b) begin
      lt <= 1'b1;
    end else begin
      eq <= 1'b1;
    end
  end
endmodule
```

デフォルト値を用いない記述

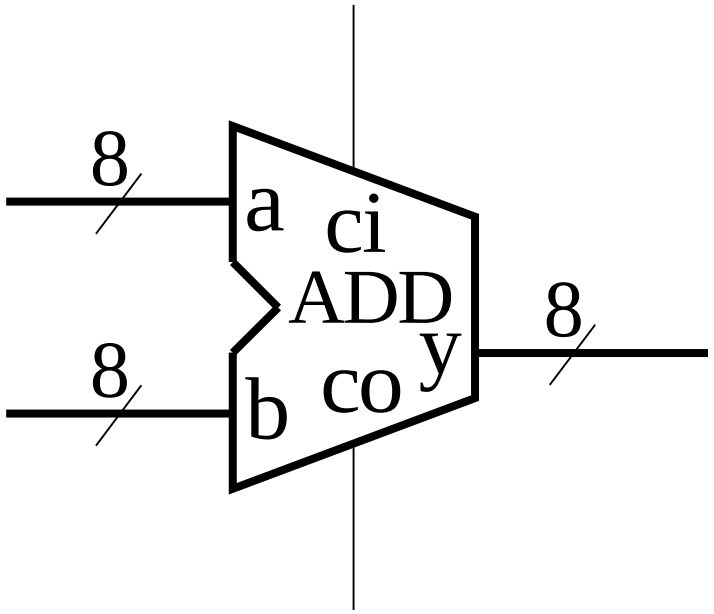
```
module cmp(a, b, gt, lt, eq);
  input  [7:0] a, b;
  output       gt, lt, eq;
  reg          gt, lt, eq;

  always@(a or b) begin
    if(a > b) begin
      gt <= 1'b1;
      lt <= 1’b0;
      eq <= 1’b0;
    end else if(a < b) begin
      gt <= 1’b0;
      lt <= 1'b1;
      eq <= 1’b0;
    end else begin
      gt <= 1’b0;
      lt <= 1’b0;
      eq <= 1'b1;
    end
  end
endmodule
```

デフォルト値を用いる記述

```
module cmp(a, b, gt, lt, eq);
  input  [7:0] a, b;
  output       gt, lt, eq;
  reg          gt, lt, eq;

  always@(a or b) begin
    gt <= 1’b0;
    lt <= 1’b0;
    et <= 1’b0;
    if(a > b) begin
      gt <= 1'b1;
    end else if(a < b) begin
      lt <= 1'b1;
    end else begin
      eq <= 1'b1;
    end
  end
endmodule
```

回路の動作は同じ
（記述量のみが異なる）

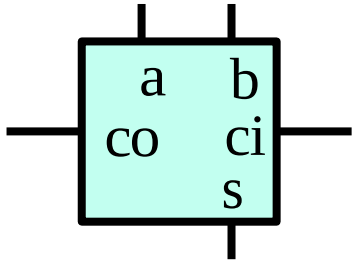
加算器

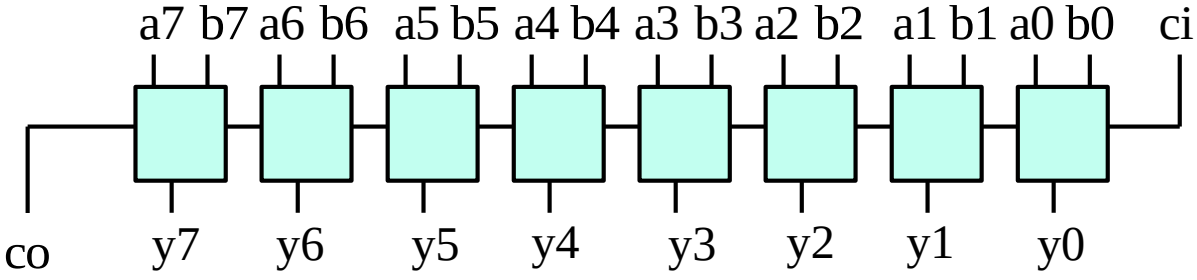



```
module add(a, b, ci, co, y);
  input  [7:0] a, b;
  input        ci;
  output       co;
  output [7:0] y;
  reg    [7:0] y;
  reg          co;

  always@(a or b or ci) begin
    {co, y} <= a + b + ci;
  end
endmodule
```

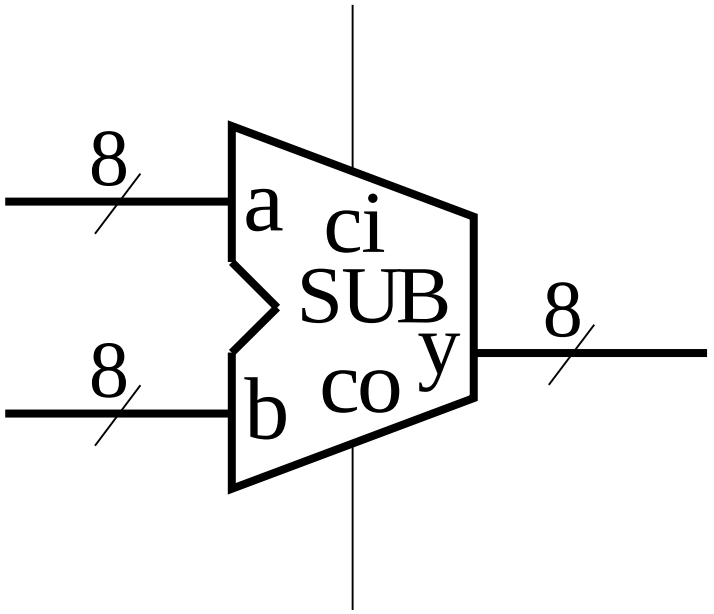
```
1  11  1
   0110  1010
+  1010  1100
-------------
   0001  0110
```

1ビット加算器





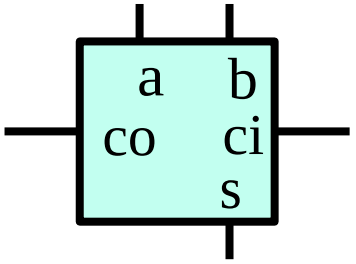
減算器

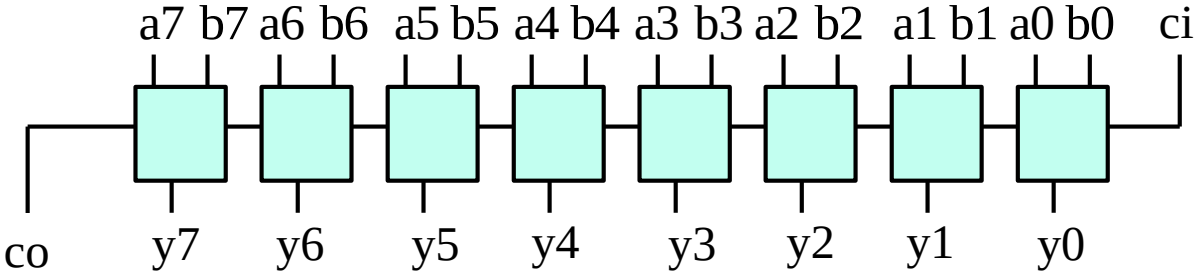



```
module sub(a, b, ci, co, y);
  input  [7:0] a, b;
  input        ci;
  output       co;
  output [7:0] y;
  reg    [7:0] y;
  reg          co;

  always@(a or b or ci) begin
    {co, y} <= a - b - ci;
  end
endmodule
```

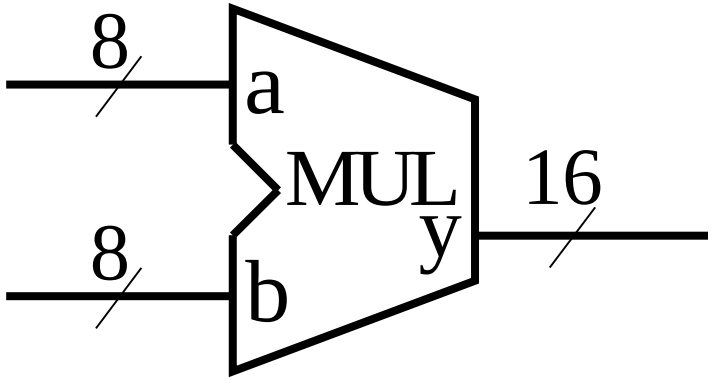
```
1  11  1
   1001  0110
-  1010  1100
-------------
   1110  1010
```

1ビット減算器





乗算器




```
module mul(a, b, y);
  input  [ 7:0] a, b;
  output [15:0] y;
  reg    [15:0] y;

  always@(a or b) begin
    y <= a * b;
  end
endmodule
```

```
   11000100
 x 11000001
 ----------
   11000100
  00000000
 00000000
00000000
00000000
00000000
11000100
11000100
----------------
1001001111000100
```

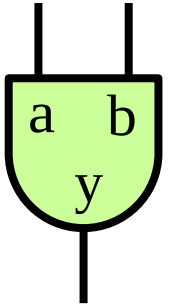
乗算器のHDL記述は数行のみ

実際の回路は（ビット数）^{2} 個の加算器が合成

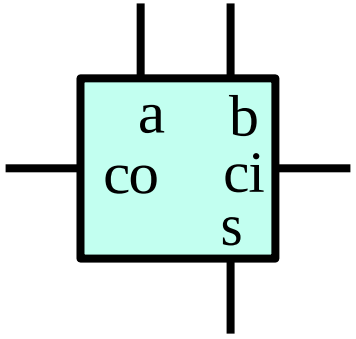
一般に、HDLソースファイルの記述量と
合成される回路量が比例するとは限らない

乗算器の構造

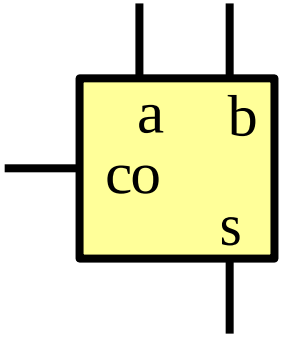
```
  11000100
x 11000001
----------
  11000100
 00000000 0
00000000 00
00000000 000
00000000 0000
00000000 00000
11000100 000000
11000100 0000000
----------------
1001001111000100
```



AND回路

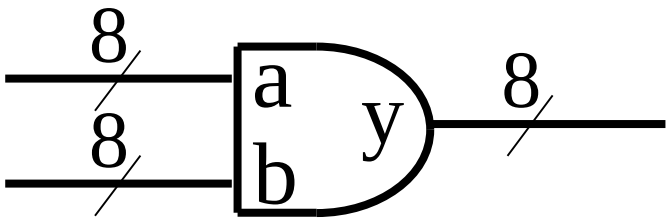


全加算器



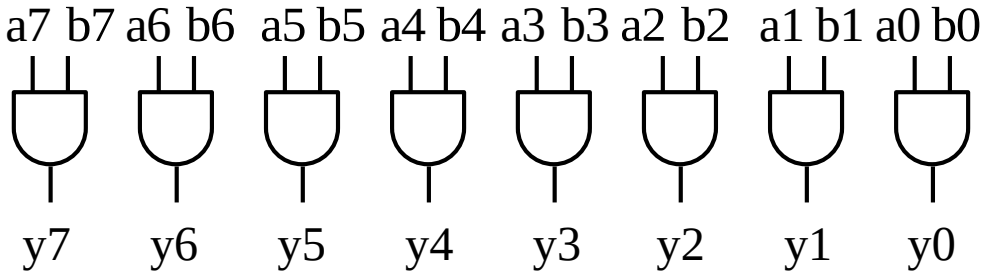
半加算器

論理演算回路～AND回路




```
module andl(a, b, y);
  input  [7:0] a, b;
  output [7:0] y;
  reg    [7:0] y;

  always@(a or b) begin
    y <= a & b;
  end
endmodule
```

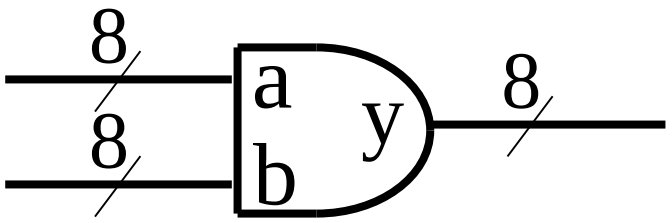
```
  1001 0110
& 1010 1100
-----------
  1000 0100
```



ビットごとに論理演算

論理演算ではキャリーは無し

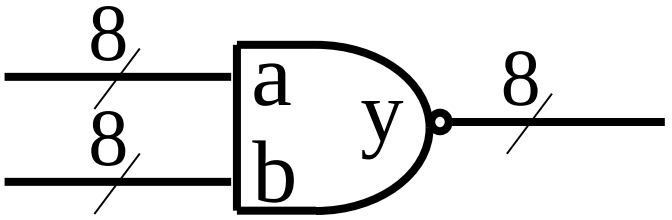
AND回路




```
  1001  0110
& 1010  1100
-------------
  1000  0100
```

```
module andl(a, b, y);
  input  [7:0] a, b;
  output [7:0] y;
  reg    [7:0] y;

  always@(a or b) begin
    y <= a & b;
  end
endmodule
```

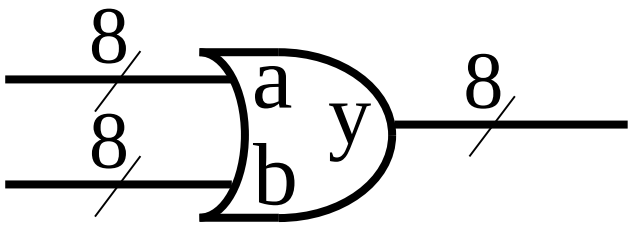
NAND回路




```
  1001  0110
~& 1010  1100
-------------
  0111  1011
```

```
module nandl(a, b, y);
  input  [7:0] a, b;
  output [7:0] y;
  reg    [7:0] y;

  always@(a or b) begin
    y <= ~(a & b);
  end
endmodule
```

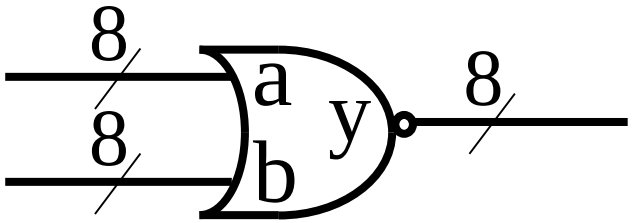
OR回路




```
  1001  0110
| 1010  1100
-------------
  1011  1110
```

```
module or1(a, b, y);
  input  [7:0] a, b;
  output [7:0] y;
  reg    [7:0] y;

  always@(a or b) begin
    y <= a | b;
  end
endmodule
```

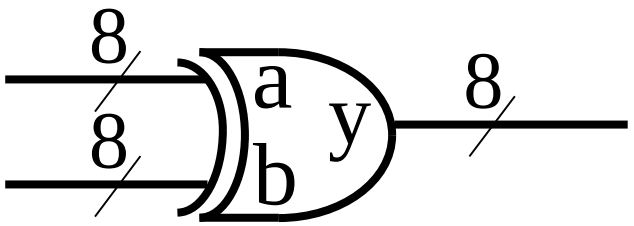
NOR回路




```
  1001  0110
~| 1010  1100
-------------
  0100  0001
```

```
module nor1(a, b, y);
  input  [7:0] a, b;
  output [7:0] y;
  reg    [7:0] y;

  always@(a or b) begin
    y <= ~(a | b);
  end
endmodule
```

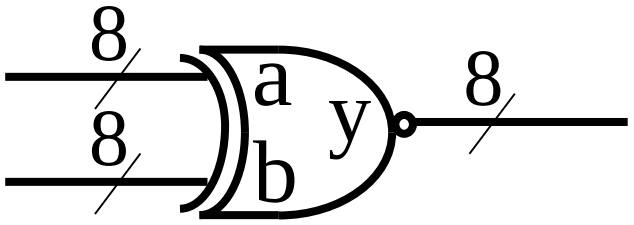
XOR回路




```
  1001  0110
^ 1010  1100
-------------
  0011  1010
```

```
module xorl(a, b, y);
  input  [7:0] a, b;
  output [7:0] y;
  reg    [7:0] y;

  always@(a or b) begin
    y <= a ^ b;
  end
endmodule
```

XNOR回路

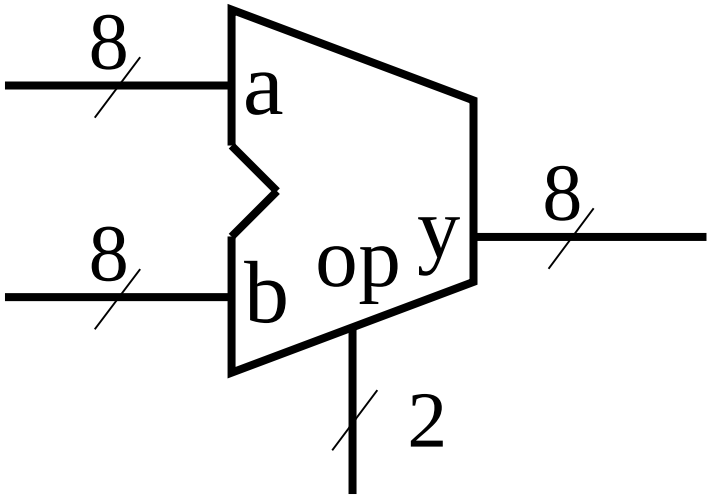



```
  1001  0110
~^ 1010  1100
-------------
  1100  0101
```

```
module xnorl(a, b, y);
  input  [7:0] a, b;
  output [7:0] y;
  reg    [7:0] y;

  always@(a or b) begin
    y <= ~(a ^ b);
  end
endmodule
```

課題

以下の論理演算ユニットを3通りの方法で設計し
それぞれ同様に動作することを確認せよ



op	y	演算
00	a & b	AND
01	a | b	OR
10	a ^ b	XOR
11	~(a | b)	NOR

4つの演算結果のうち1つを選択して出力
2ビットの制御信号で選択

設計方法(1)

動作記述で設計(テストベンチは各自作成せよ)

```
`define AND 2'b00
`define OR  2'b01
`define XOR 2'b10
`define NOR 2'b11

module lu(y, a, b, op);
  input  [7:0] a, b;
  input  [1:0] op;
  output [7:0] y;
  reg    [7:0] y;
```

```
  always@(a or b or op) begin
    case(op)
      `AND     : y <= a & b;
      `OR      : y <= a | b;
      `XOR     : y <= a ^ b;
      `NOR     : y <= ~(a | b);
      default : y <= 8'bxxxxxxxx;
    endcase
  end
endmodule
```

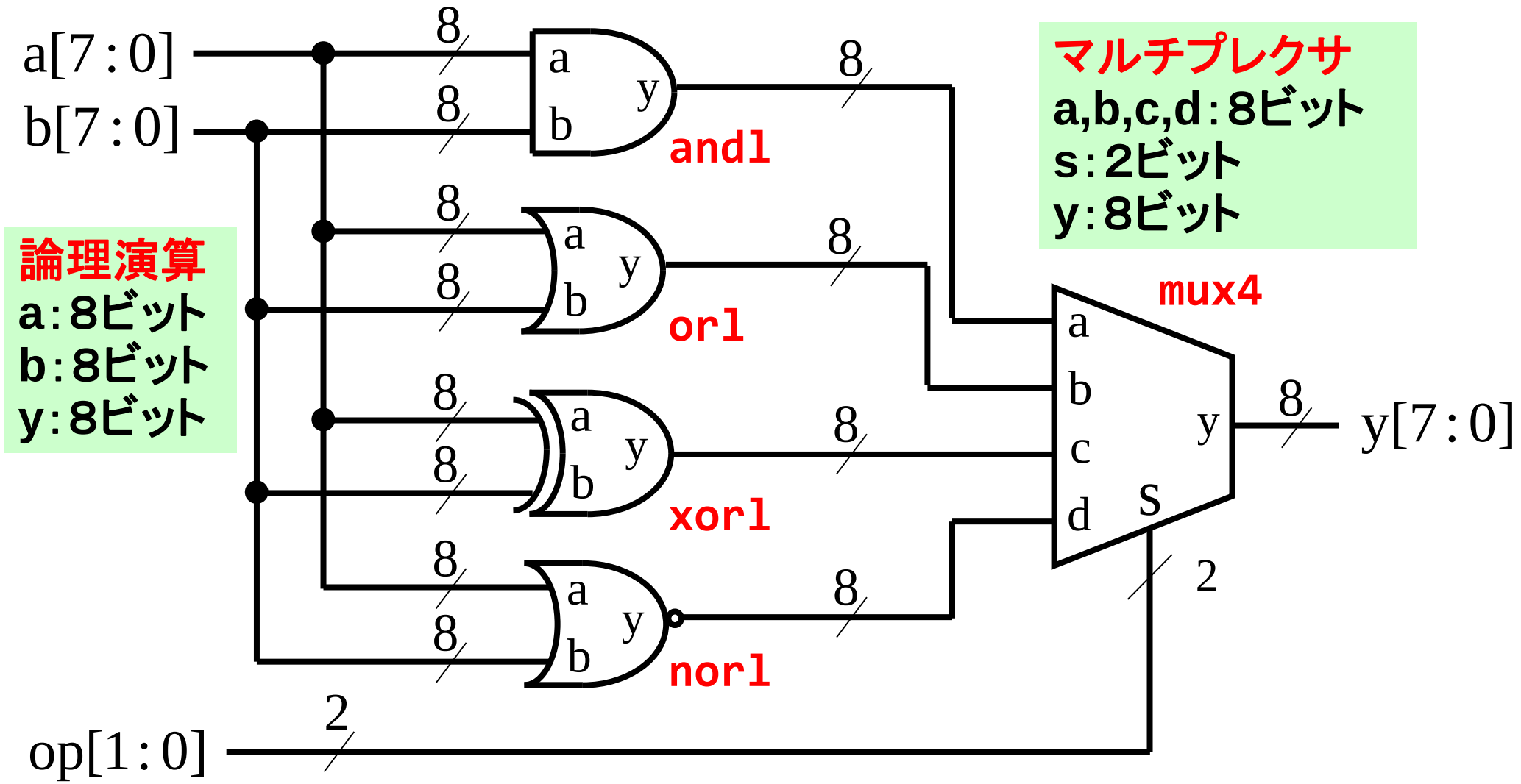
`define name const

定数(const)と対応する名前(name)を定義

定義した名前を使用するときは`name のように記述

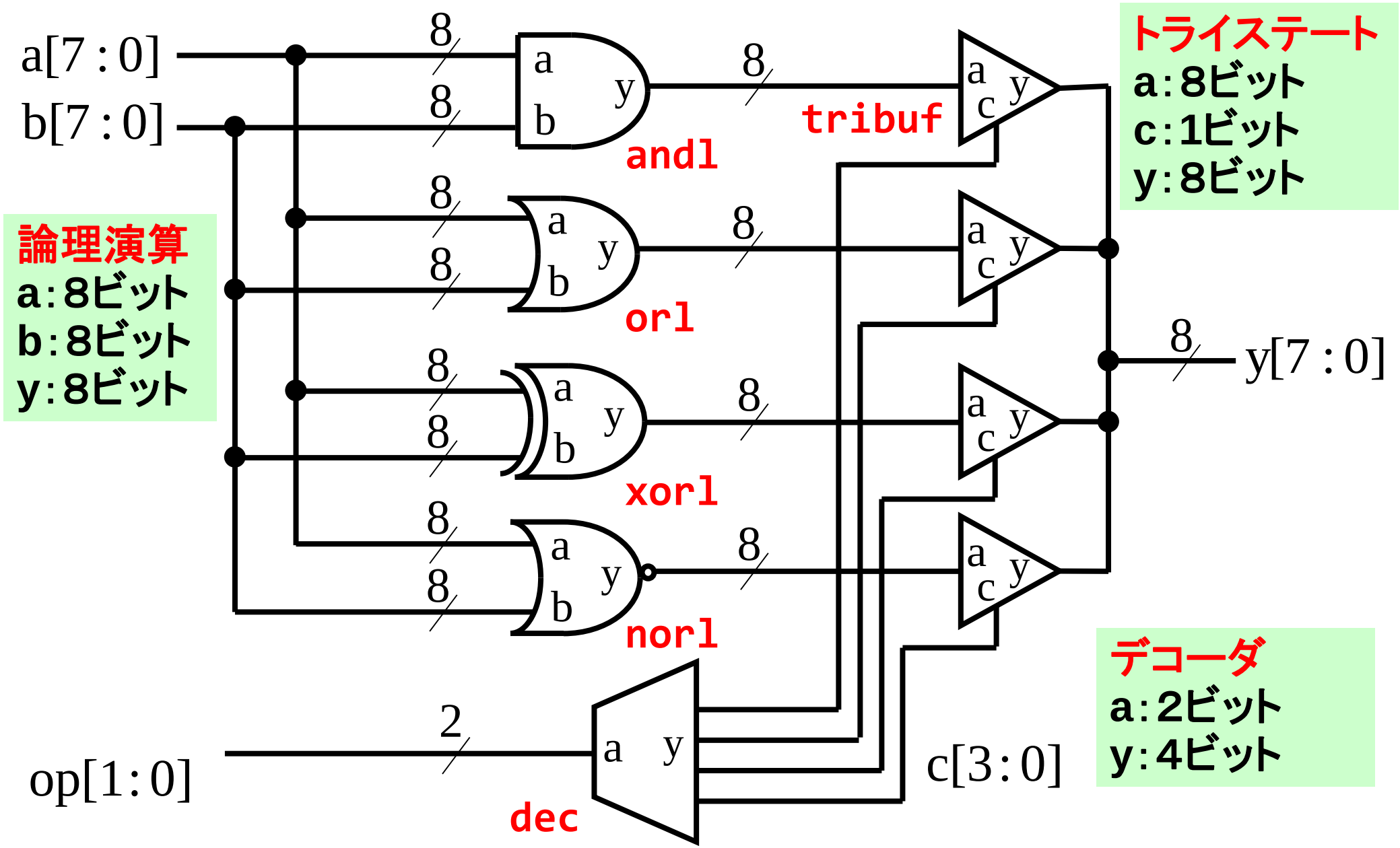
設計方法(2)

各論理演算回路をそれぞれ個別のモジュールで定義
4入力マルチプレクサで出力を選択



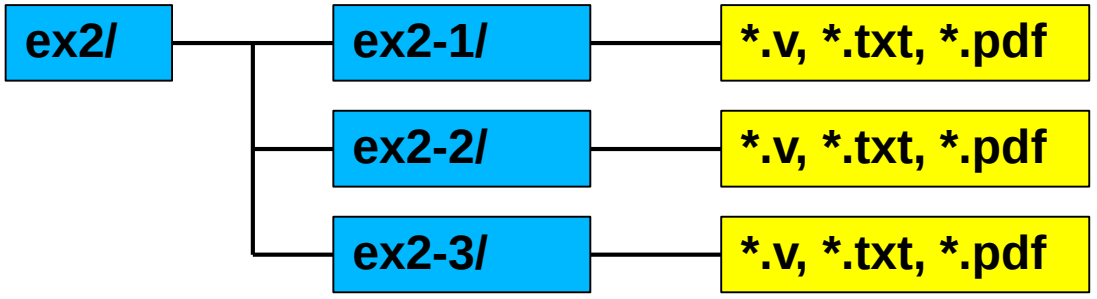
設計方法(3)

いずれか1つのトライステートのみが接続されるようにデコーダを設計



提出すべきファイル

- 課題の回路のHDLソースファイル（テストベンチを含む）
  各設計方法（1）～（3）に対してディレクトリ（ex2-?）を作成すること
- 課題の回路のシミュレーション実行結果
  （ iverilog の実行結果をリダイレクトして txt ファイルを作成）
- 課題の回路のシミュレーション波形
  （gtkwaveから pdf ファイルを作成）
- 設計方法ごとに圧縮アーカイブファイル（ex2-?.tar.gz）を作成して
  Web上から提出せよ




```
cd ex2
tar zcvf ex2-1.tar.gz ex2-1/
tar zcvf ex2-2.tar.gz ex2-2/
tar zcvf ex2-3.tar.gz ex2-3/
```