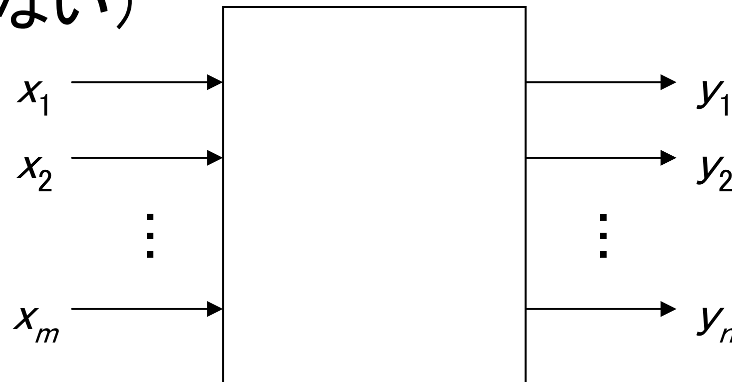
算術論理演算ユニットの設計

（教科書4.5節）

組合せ論理回路(復習)

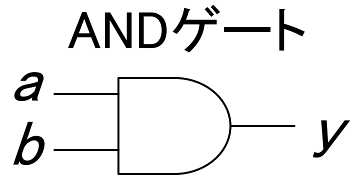
組合せ論理回路: 出力値が入力値のみの関数となっている論理回路. 論理関数 $f: \{0, 1\}^m \rightarrow \{0, 1\}^n$ を実現. (フィードバック・ループや記憶回路を含まない)



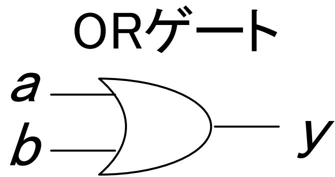
$y_i = f_i(x_1, x_2, x_3, ..., x_m)$ (for $1 \leq i \leq n$)

基本的な組合せ論理回路: インバータ, ANDゲート, ORゲート, XORゲートなど.

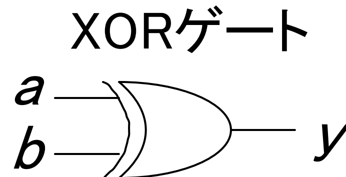
組合せ論理回路（復習）



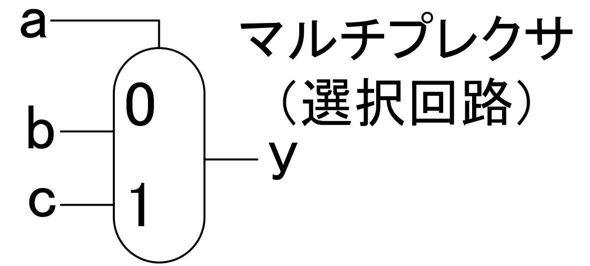
a	b	y
0	0	0
0	1	0
1	0	0
1	1	1



a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

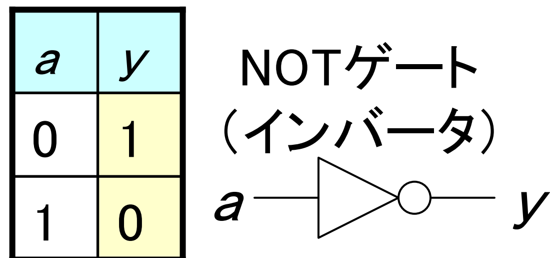


a	b	y
0	0	0
0	1	1
1	0	1
1	1	0



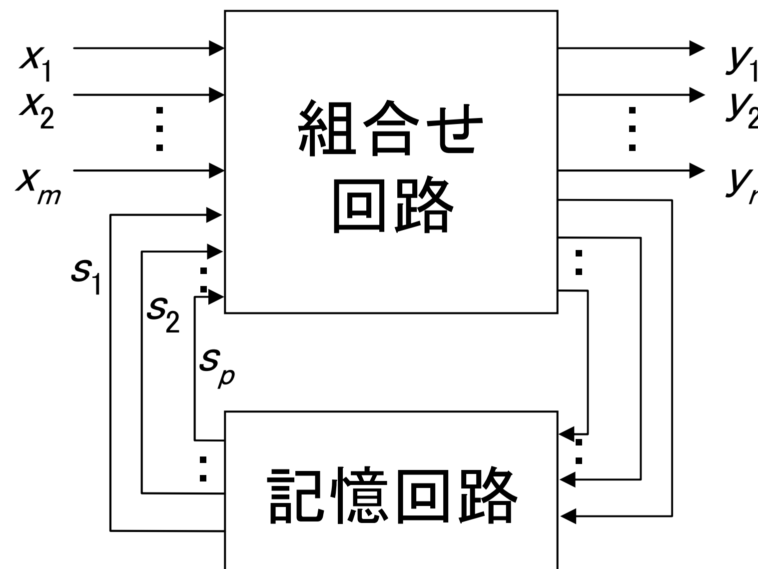
a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

a	y
0	1
1	0



順序回路(復習)

順序回路: 出力値が, 入力値と回路の状態値の関数となっている論理回路. また, 次状態値も入力値と回路の現状態値の関数となっている. 順序機械 M=(I, O, S, δ, λ) を実現.



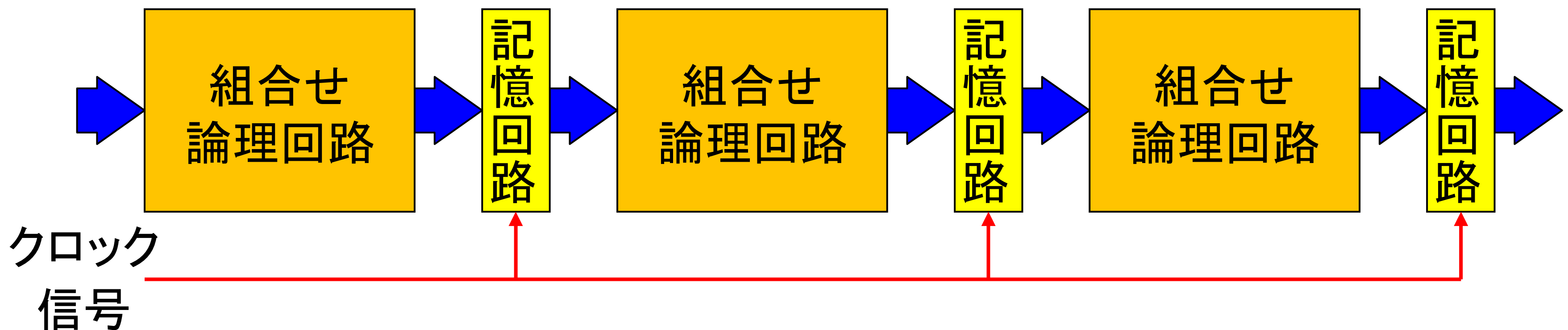
I: 入力集合
O: 出力集合
S: 状態集合
δ: 状態遷移関数
λ: 出力関数

$$y_i = f_i(x_1, x_2, \ldots, x_m, s_1, s_2, \ldots, s_p) \quad (\text{for } 1 \leq i \leq n)$$

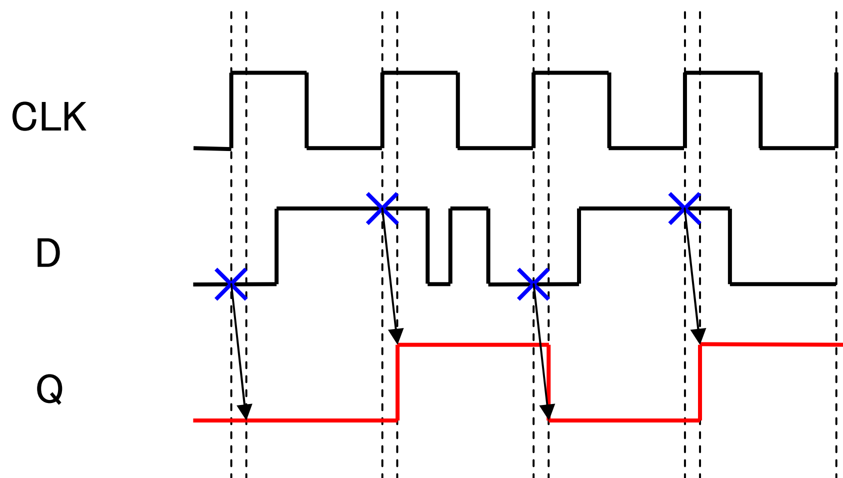
$$s_j = g_j(x_1, x_2, \ldots, x_m, s_1, s_2, \ldots, s_p) \quad (\text{for } 1 \leq j \leq p)$$

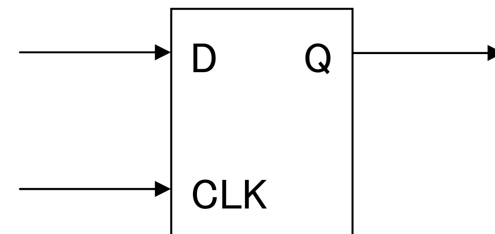
同期式順序回路（復習）

同期回路: クロックに同期して動作する順序論理回路．クロックの立ち上がり時の入力と状態で，次回クロックが立ち上がるまでの出力と状態を確定．



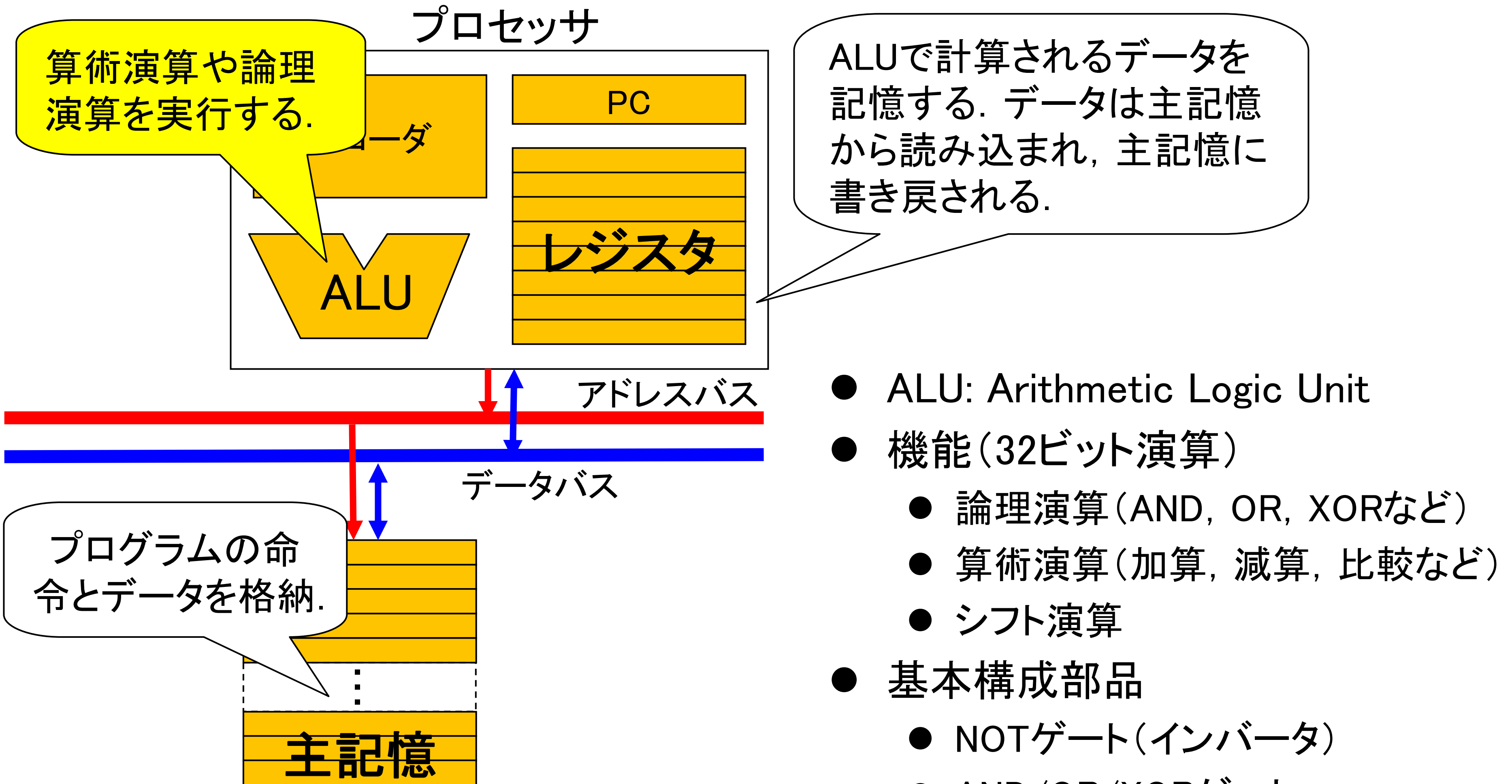
代表的なクロック同期式記憶回路:Dフリップフロップ





九州大学工学部電気情報工学科（2006年度）

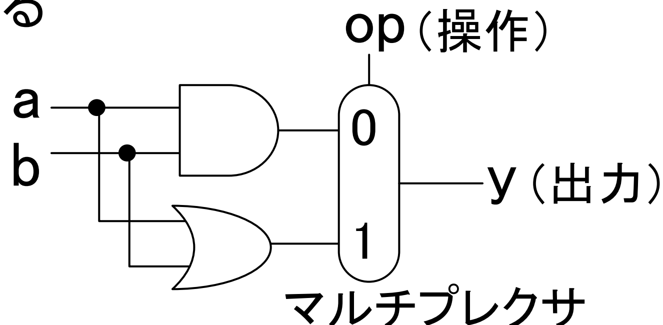
算術論理演算ユニットALU



＊)本講義では, XORならびにシフト演算は省略する

1ビット論理演算器を設計してみよう！

- 仕様
  - 入力：a, b, op（各1ビット）
  - 出力：y（1ビット）
  - 機能
    - a, bに対しる「AND」か「OR」の論理演算
    - opにより操作（ANDかORか）を決定
- 基本的な考え方
  - 論理積（AND）と論理和（OR）の両方を並列に求める
  - op信号の値に基づき何れか一方を選択してyへ出力する

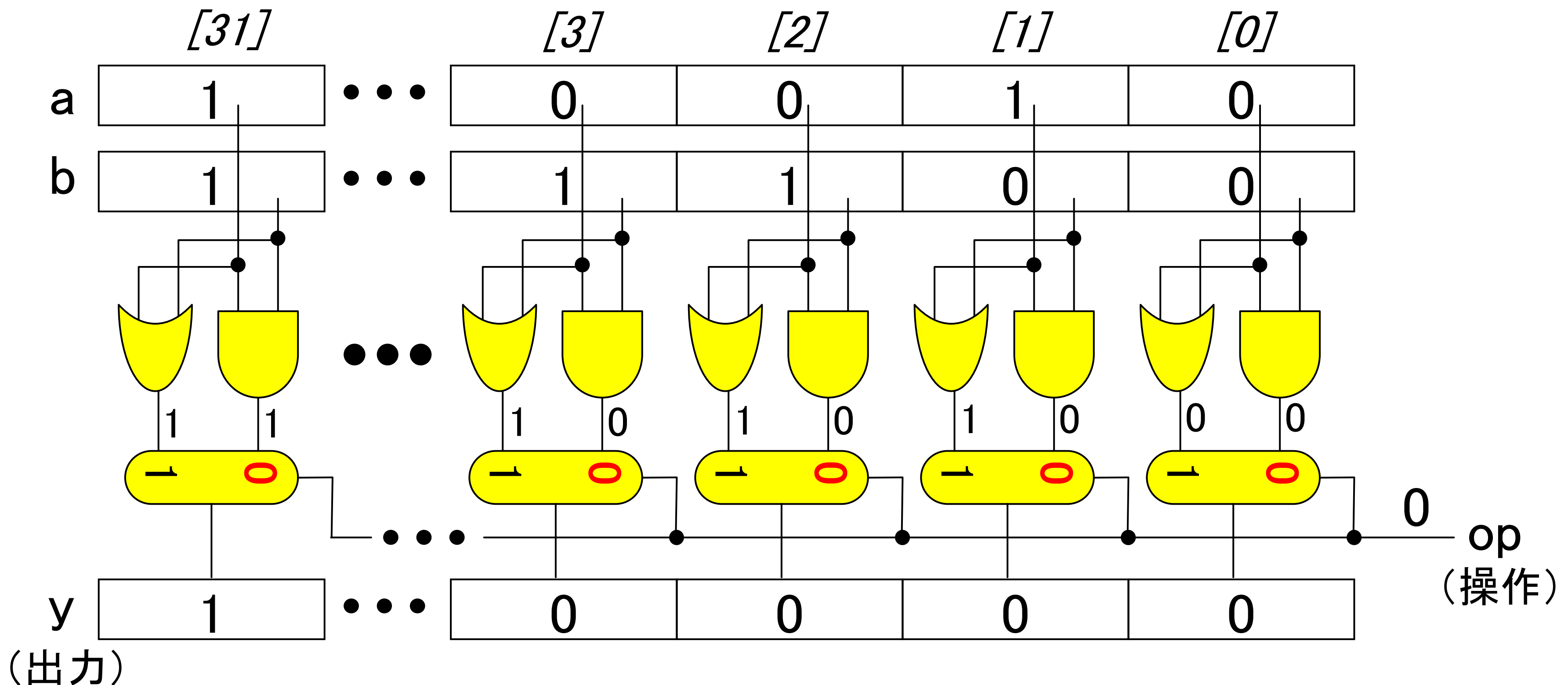


真理値表

op	a	b	y
0	0	0	0 (a & b)
0	0	1	0 (a & b)
0	1	0	0 (a & b)
0	1	1	1 (a & b)
1	0	0	0 (a or b)
1	0	1	1 (a or b)
1	1	0	1 (a or b)
1	1	1	1 (a or b)

32ビット論理演算器の設計(1)

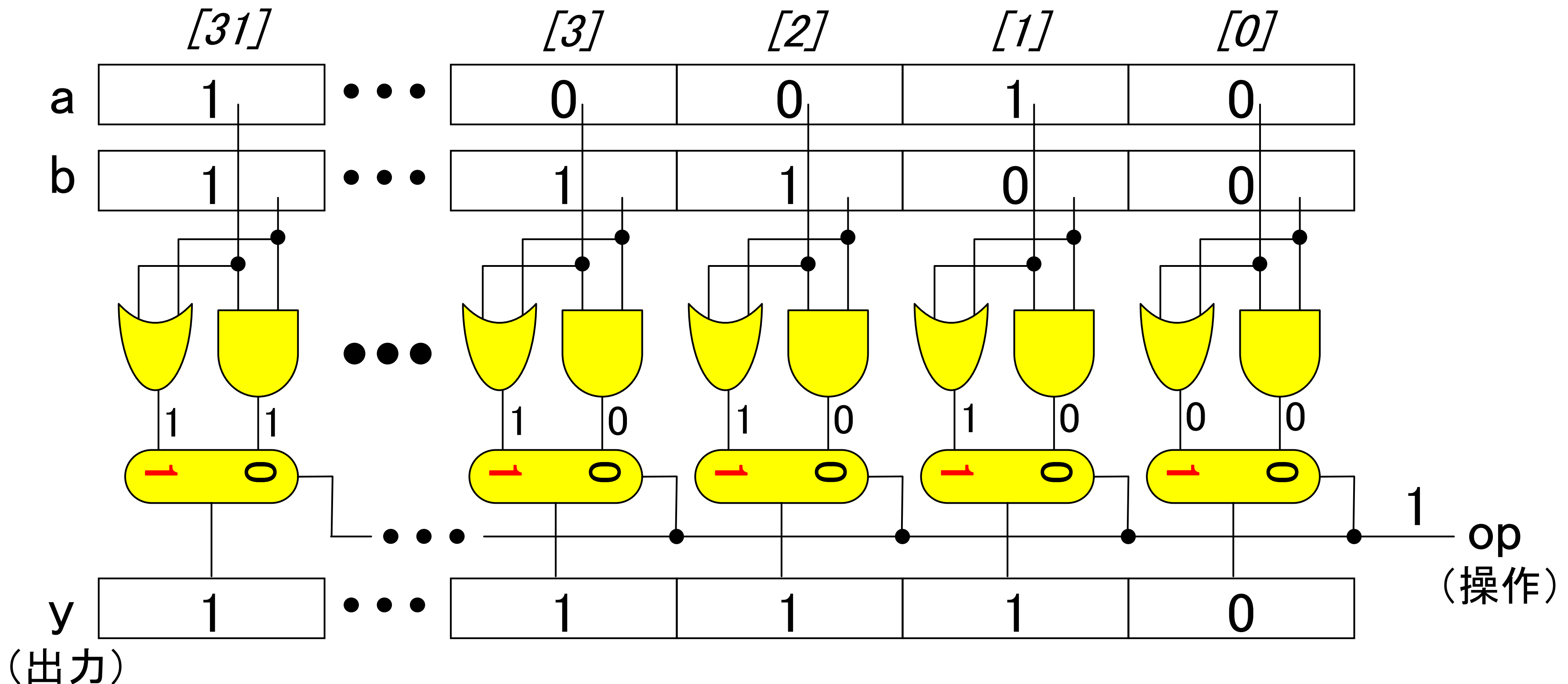
オペランドのビットごとにANDやORをとる



論理積の場合(op信号が0)

32ビット論理演算器の設計(2)

オペランドのビットごとにANDやORをとる



論理和の場合(op信号が1)

1ビット加算器を設計してみよう！（1）

- 仕様
  - 入力：a, b, cin（各1ビット）
  - 出力：s, cout（各1ビット）
  - 機能
    - 入力a，b，ならびに，下の桁からの桁上がり（cin）を加算
    - 和（s）と上の桁への桁上がり（cout）を出力

出力：上位への桁上げ（cout）キャリー・アウト

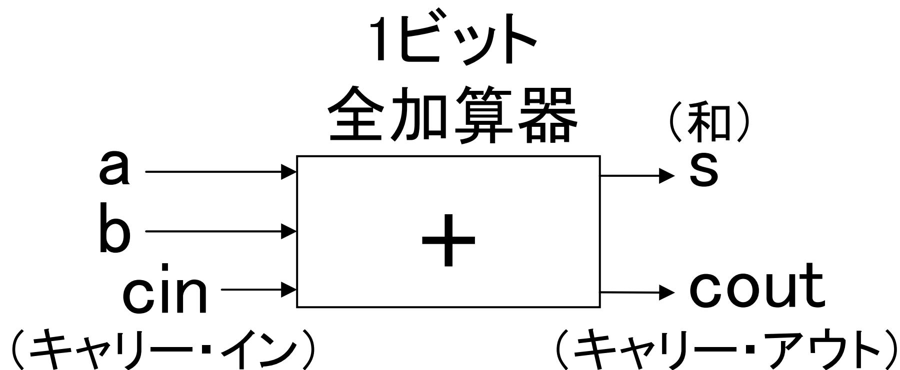
入力：下位からの桁上げ（cin）キャリー・イン

	1	1	1	0	1	0	0		
	0	1	1	1	0	1	1	0	←a
+)	0	0	1	1	0	1	0	1	←b
	1	0	1	0	1	0	1	1	

入力：足される数（aとb）

出力：和（s）

1ビット加算器を設計してみよう！（2）



真理値表

cin	a	b	s	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

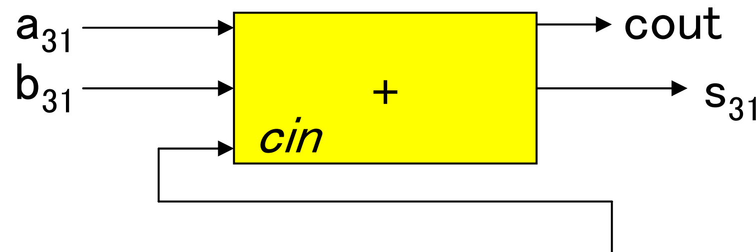
sとcoutの積和標準形

$$s = \overline{a} \cdot \overline{b} \cdot cin + \overline{a} \cdot b \cdot \overline{cin} + a \cdot \overline{b} \cdot \overline{cin} + a \cdot b \cdot cin$$

$$cout = \overline{a} \cdot b \cdot cin + a \cdot \overline{b} \cdot cin + a \cdot b \cdot \overline{cin} + a \cdot b \cdot cin$$

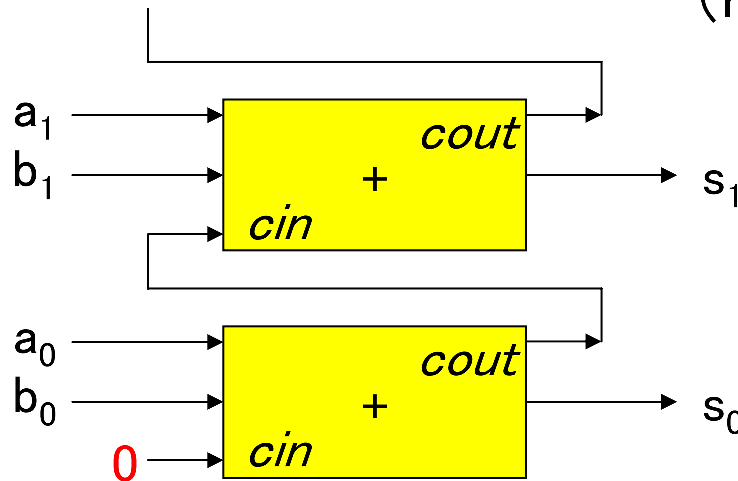
32ビット加算器の設計

1ビット加算器を使った32ビット加算器



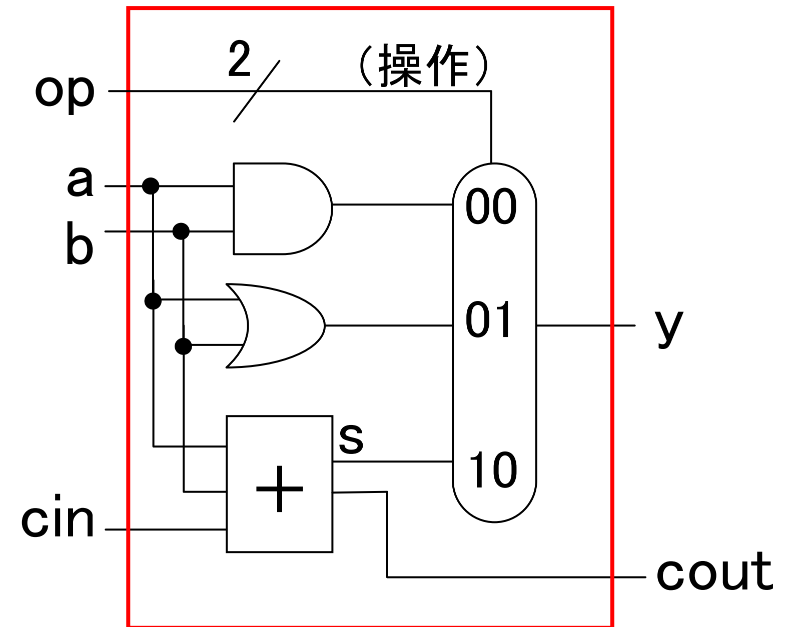
下位から上位へ桁上げが伝播

順次桁上げ加算器
(ripple carry adder)

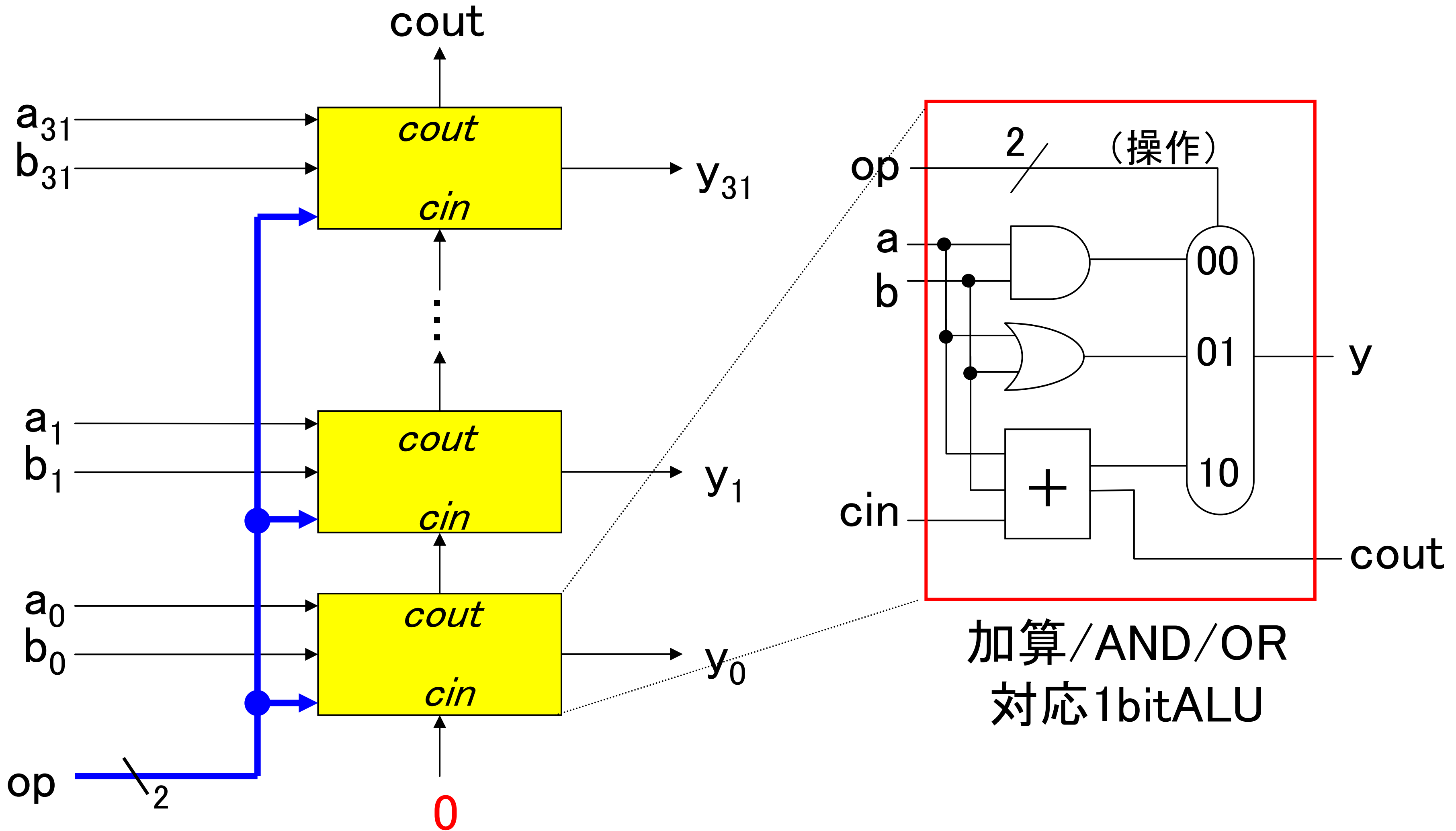


加算/AND/OR対応1ビットALUの設計

- 仕様
  - 入力：a, b, cin（各1ビット）
  - 入力：op（2ビット）
  - 出力：y, cout（各1ビット）
  - 機能
    - 「AND」か「OR」か「加算」
    - opにより操作（出力）を決定
      - op=00→aとbの論理積（AND）
      - op=01→aとbの論理和（OR）
      - op=10→aとbとcinの加算



加算/AND/OR対応32ビットALUの設計



減算器の設計(1)

減算(*b* を引く)=負数の加算(−*b* を足す)

2の補数表現の場合，符号を気にすることなく，符号なし整数の加算とまったく同じ方法で減算できる.

キャリー	~~1~~	1	1			1	1		
		0	1	1	0	0	1	1	0	$102_{(10)}$
+)		1	1	1	0	0	0	1	1	$-29_{(10)}$
		0	1	0	0	1	0	0	1	$73_{(10)}$

減算器の設計(2)

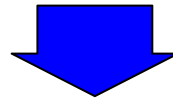
2の補数表現による負数のビット表現の簡単な求め方:

① 2進数の 0 と 1 を反転する.

0000 0000 0000 0101 → 1111 1111 1111 1010

② ①で得られた2進数をひとつカウントアップする.

1111 1111 1111 1010 → 1111 1111 1111 1011



a − b を求めるには:

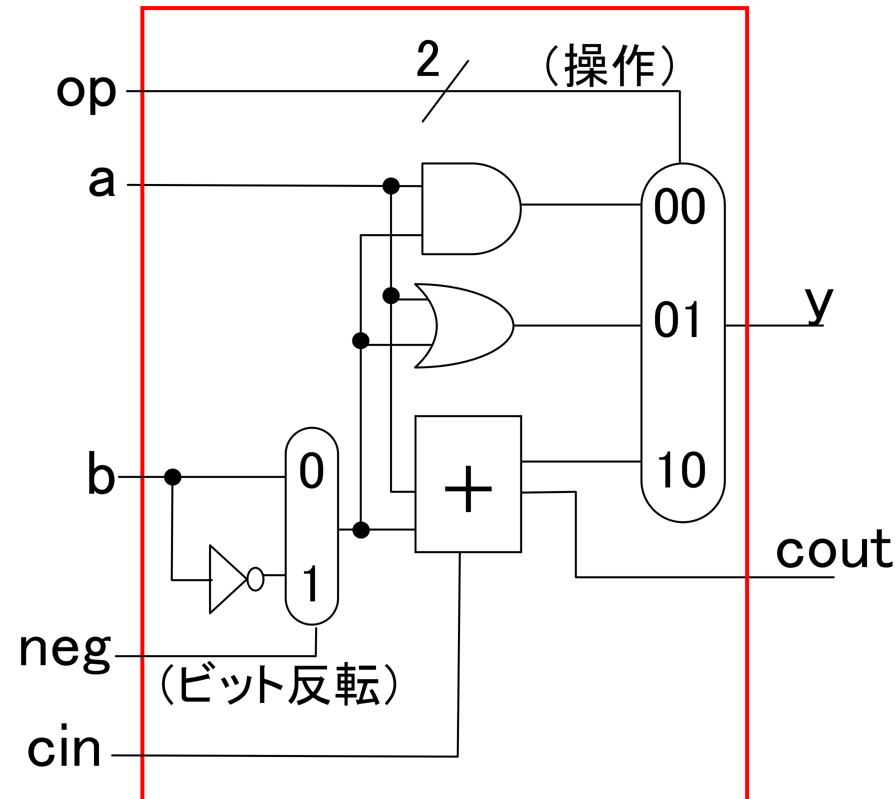
① b の 0 と 1 を反転する.
② ①の結果に 1 を加算する.

} 「－b」の２の補数表現を求める

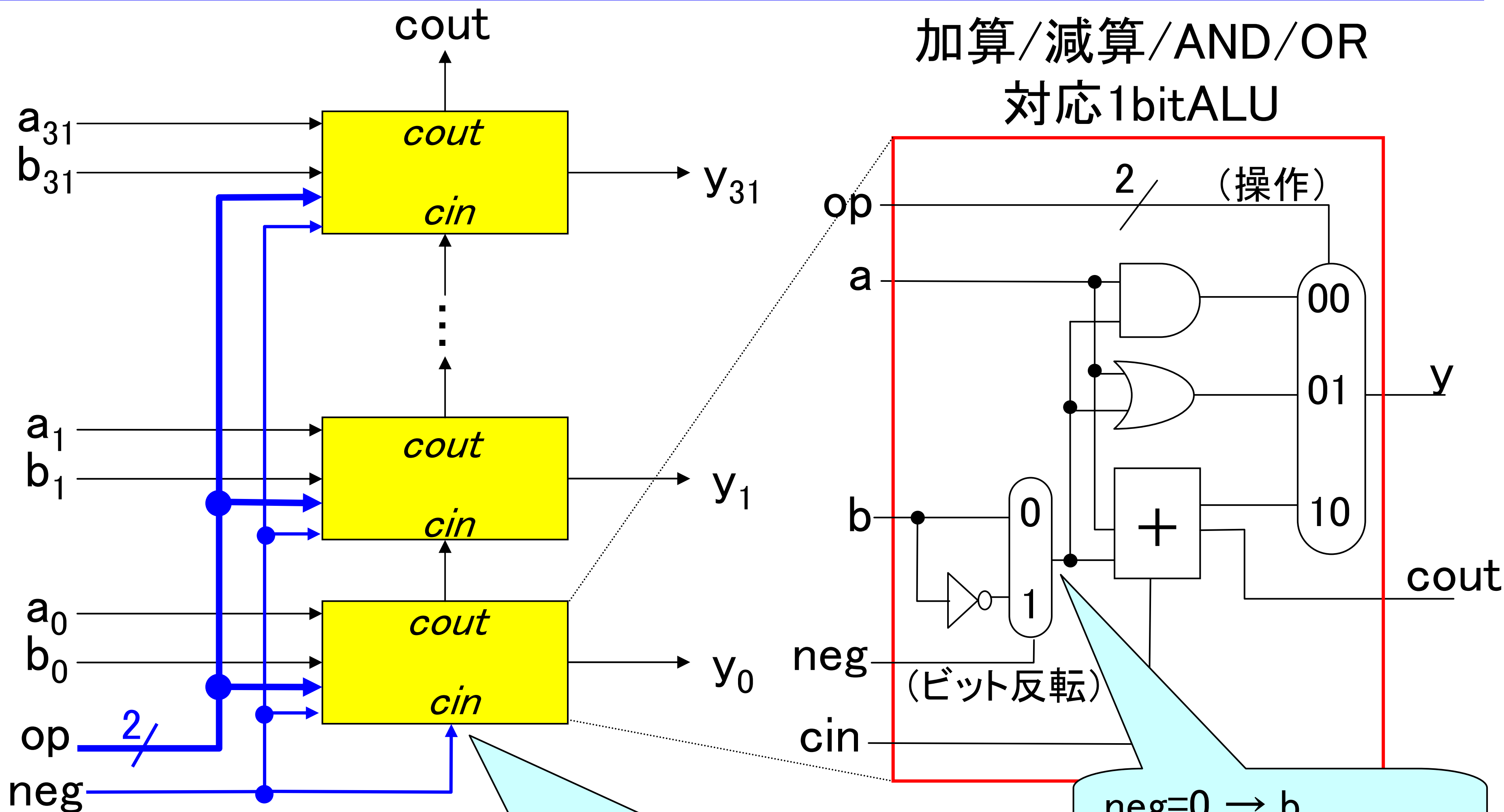
③ a と②の結果を加算する. } a＋(－b)を計算する

加算/減算/AND/OR対応1ビットALUの設計

- 入力：a, b, cin（各1ビット）
- 入力：op（2ビット）, neg（1ビット）
- 出力：y, cout（各1ビット）
- 機能
  - 「AND」か「OR」か「加算」か「減算」
  - opにより操作を決定
    - op=00→論理積（AND）
    - op=01→論理和（OR）
    - op=10→加算または減算
  - negにより入力bを反転するか否か決定
    - neg=0→反転なし（AND/OR/加算）
    - neg=1→反転（減算）



加算/減算/AND/OR対応32ビットALUの設計



neg=0 → cinは0
neg=1 →cinは1（つまり＋1）

neg=0 → b
neg=1 →bの反転

op=10, neg=1の時
「a＋（－b）」を出力

オーバーフロー(1)

オーバーフロー: 算術演算の結果が表現可能な値の範囲を超えること.

4bit 加算の場合:

① 正(0000～0111) + 正(0000～0111) → 0000～1110
(0～7) + (0～7) で結果は 0 ～ 14. オーバーフローの可能性あり.
結果が 1000(8)～1110(14) のとき(＝負のとき), オーバーフロー.

② 正(0000～0111) + 負(1000～1111) → 1000～0110
(0～7) + (−8～−1) で結果は −8～6. オーバーフローはない.

③ 負(1000～1111) + 正(0000～0111) → 1000～0110
(−8～−1) + (0～7) で結果は −8～6. オーバーフローはない.

④ 負(1000～1111) + 負(1000～1111) → 0000～1110
(−8～−1) + (−8～−1) で結果は −16～−2. オーバーフローの可能性あり.
結果が 0000～0111 のとき(＝正のとき), オーバーフロー.

オーバーフロー(2)

4bit 減算の場合:

⑤　正(0000～0111) − 正(0000～0111)

正(0000～0111) + 負(1000～1111) と同じ. オーバーフローなし.

⑥　正(0000～0111) − 負(1000～1111)

正(0000～0111) + 正(0000～0111) と同じ.

結果が負のとき, オーバーフロー.

⑦　負(1000～1111) − 正(0000～0111)

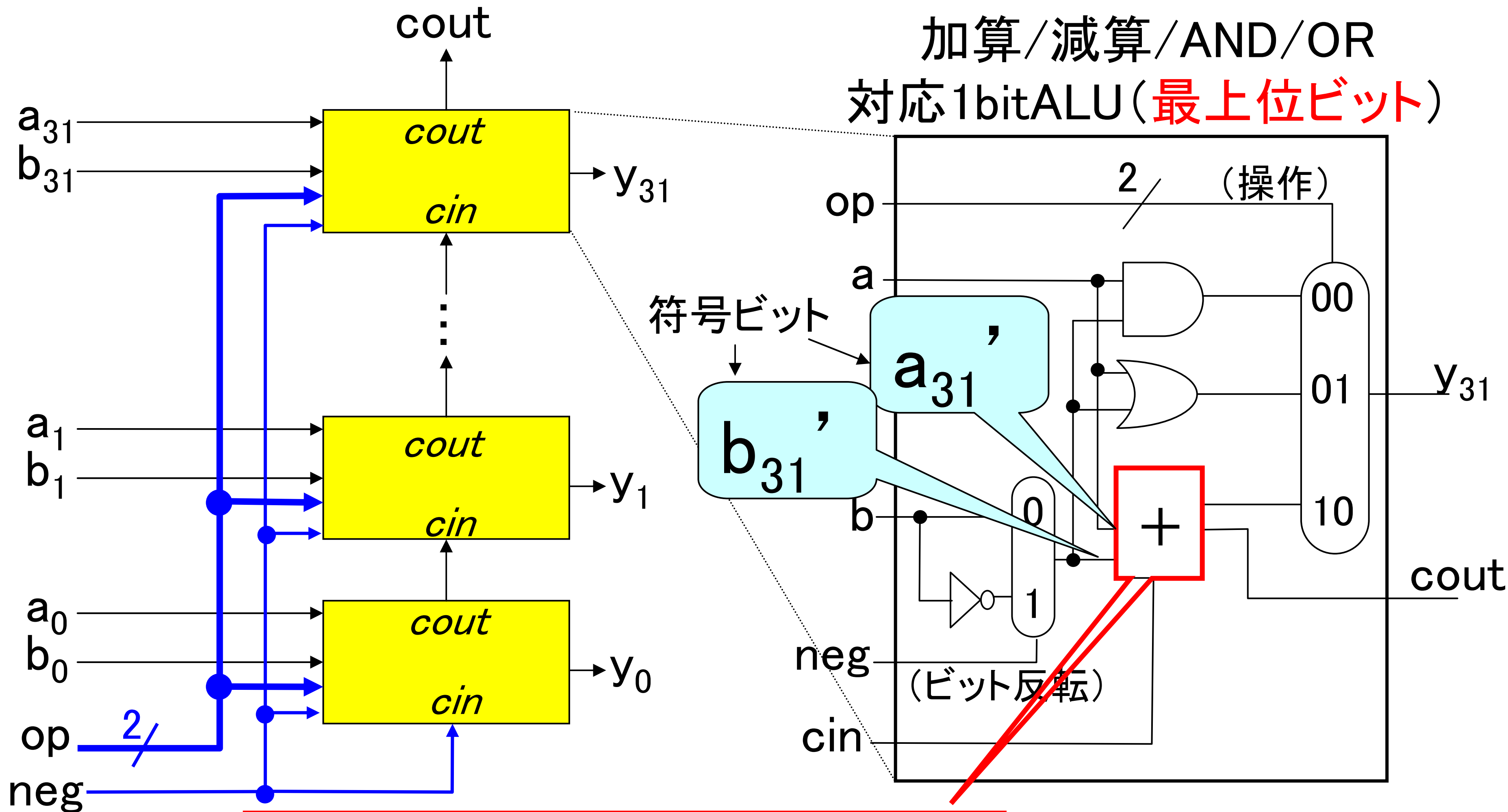
負(1000～1111) + 負(1000～1111) と同じ.

結果が正のとき, オーバーフロー.

⑧　負(1000～1111) − 負(1000～1111)

負(1000～1111) + 正(0000～0111) と同じ. オーバーフローなし.

オーバーフロー(3)



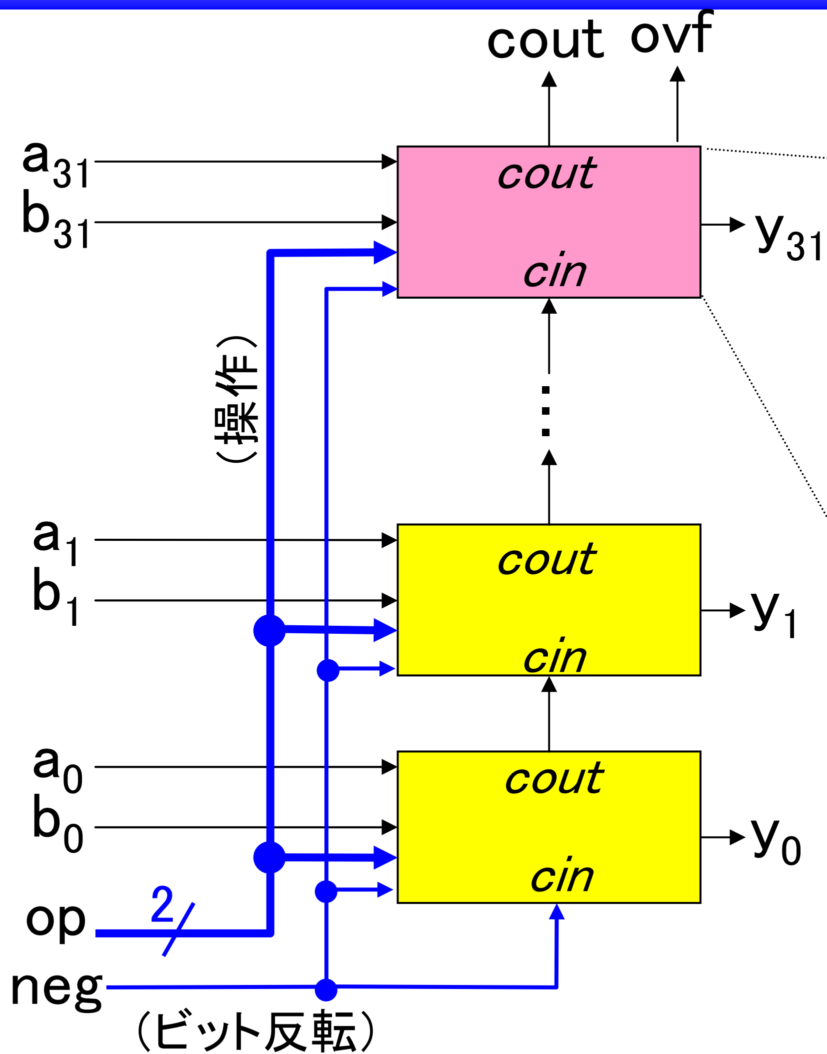
正+正=負, 負+負=正のとき
オーバーフロー発生

オーバーフロー(4)

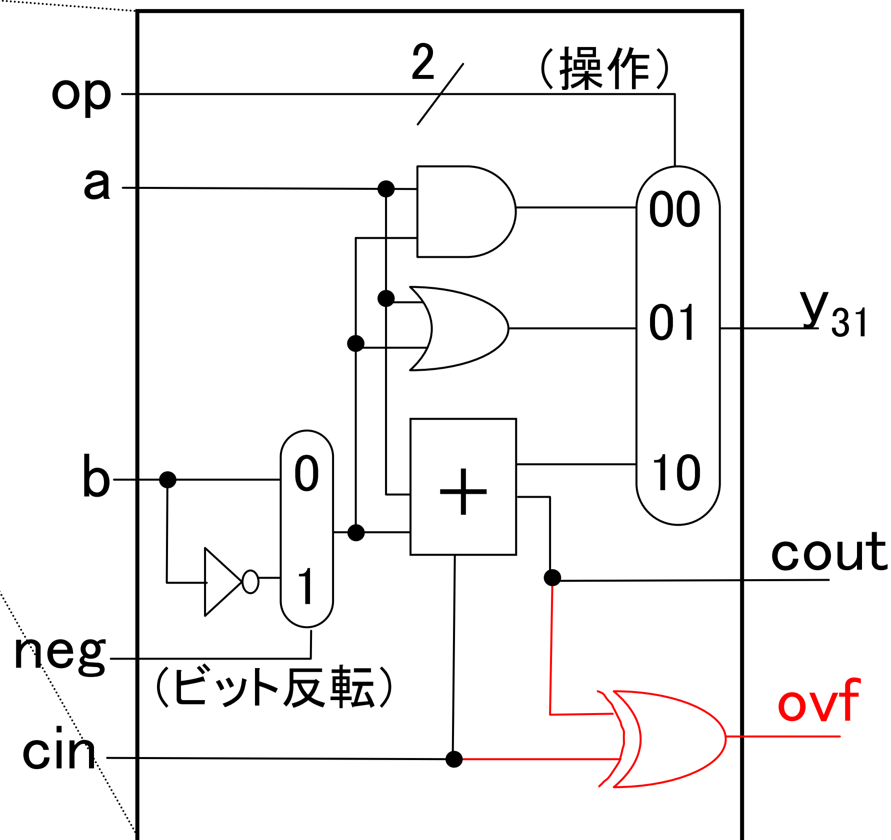
a_{31}'	b_{31}'	cin	y_{31}	cout	備考
0	0	0	0	0	①正＋正＝正／⑤正ー負＝正
0	0	1	1	0	①正＋正＝負／⑤正ー負＝負
0	1	0	1	0	②正＋負＝負／⑥正ー正＝負
0	1	1	0	1	②正＋負＝正／⑥正ー正＝正
1	0	0	1	0	③負＋正＝負／⑦負ー負＝負
1	0	1	0	1	③負＋正＝正／⑦負ー負＝正
1	1	0	0	1	④負＋負＝正／⑧負ー正＝正
1	1	1	1	1	④負＋負＝負／⑧負ー正＝負

cin $\neq$ cout ならばオーバーフロー

オーバーフロー(5)



加算/減算/AND/OR
対応1bitALU(最上位ビット)



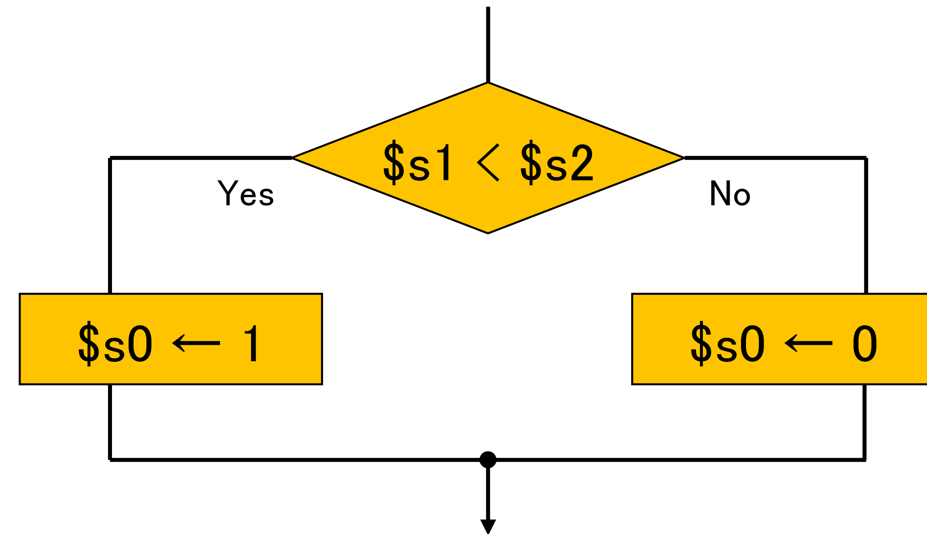
ovf:オーバーフロー出力

比較器(slt：set-on-less-than)の設計

MIPSでの比較命令の例

slt $s0, $s1, $s2

レジスタ$s1の値と$s2の値を比較して，$s1<$s2であれば$s0に値「1」を，そうでなければ値「0」を格納（分岐条件の設定に利用）



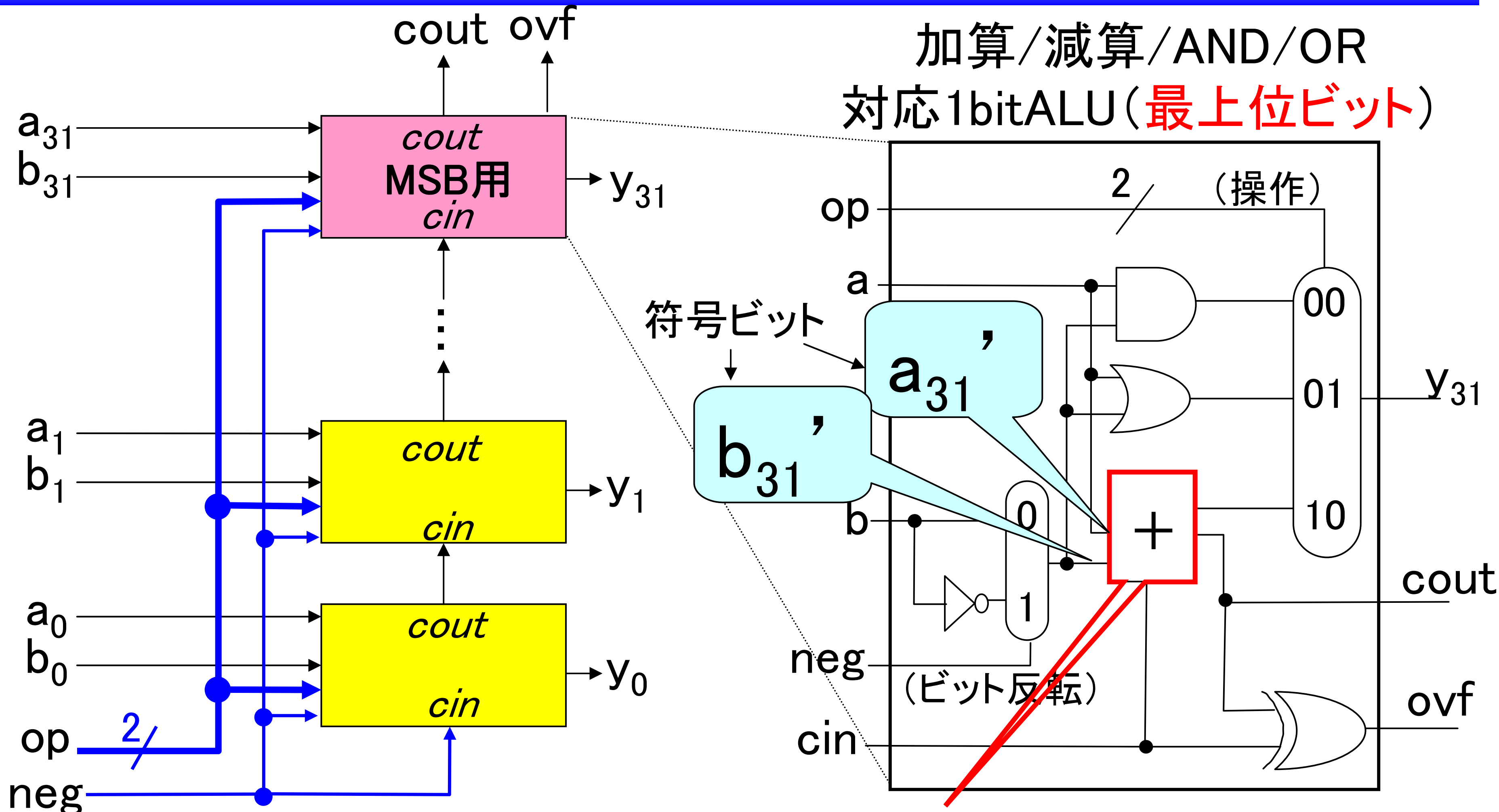
ALUに要求される機能

①32ビット入力aとbを比較
- ･「a－b<0」か否かを判定

②比較結果に基づき0か1を出力
- ･a<bの場合：32ビットの000･･･0001
- ･a>=bの場合：32ビットの000･･･0000

比較結果に依存するのは最下位ビットのみ

減算に基づく大小比較(1)



- 減算結果の符号に基づき判定(a−bの結果が負→a<b)
- 減算におけるオーバーフローに注意

減算に基づく大小比較(2)

a_{31}	b_{31}	a_{31}'	b_{31}'	cin	y_{31}	cout	ovf	備考
0	1	0	0	0	0	0	0	⑤正ー負＝正
0	1	0	0	1	1	0	1	⑤正ー負＝負
0	0	0	1	0	1	0	0	⑥正ー正＝負(a＜b)
0	0	0	1	1	0	1	0	⑥正ー正＝正
1	1	1	0	0	1	0	0	⑦負ー負＝負(a＜b)
1	1	1	0	1	0	1	0	⑦負ー負＝正
1	0	1	1	0	0	1	1	⑧負ー正＝正(a＜b)
1	0	1	1	1	1	1	0	⑧負ー正＝負(a＜b)

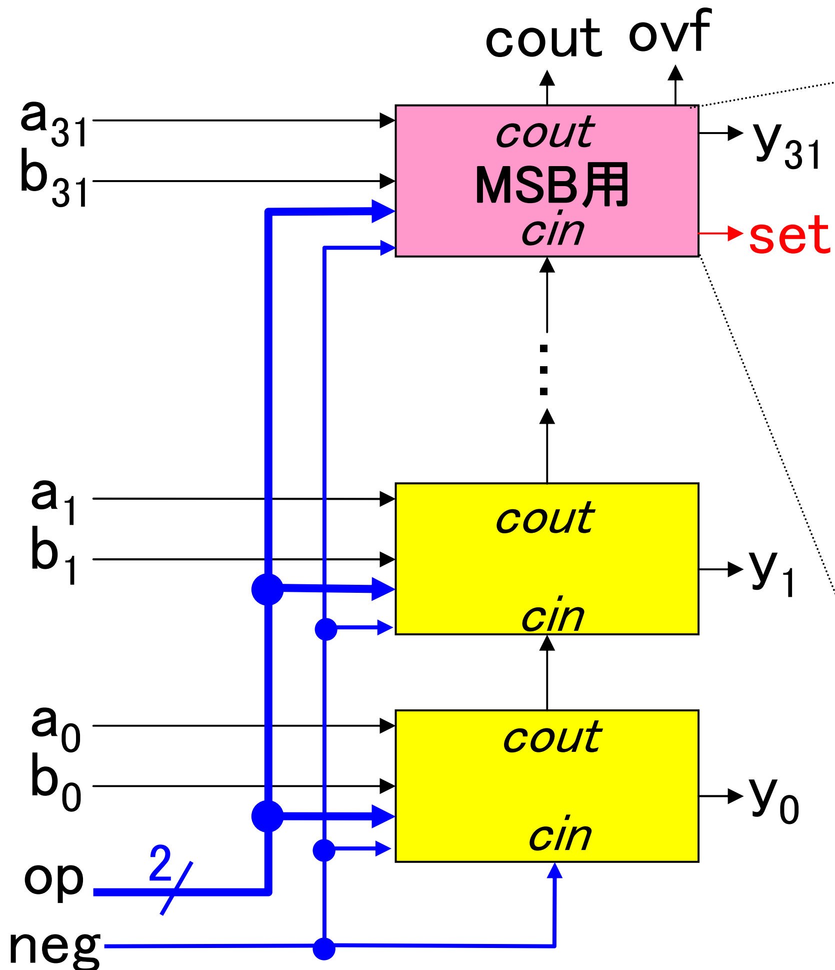
オーバーフローが生じなくて(ovf = 0), 結果が負(y_{31}=1) → a ＜ b

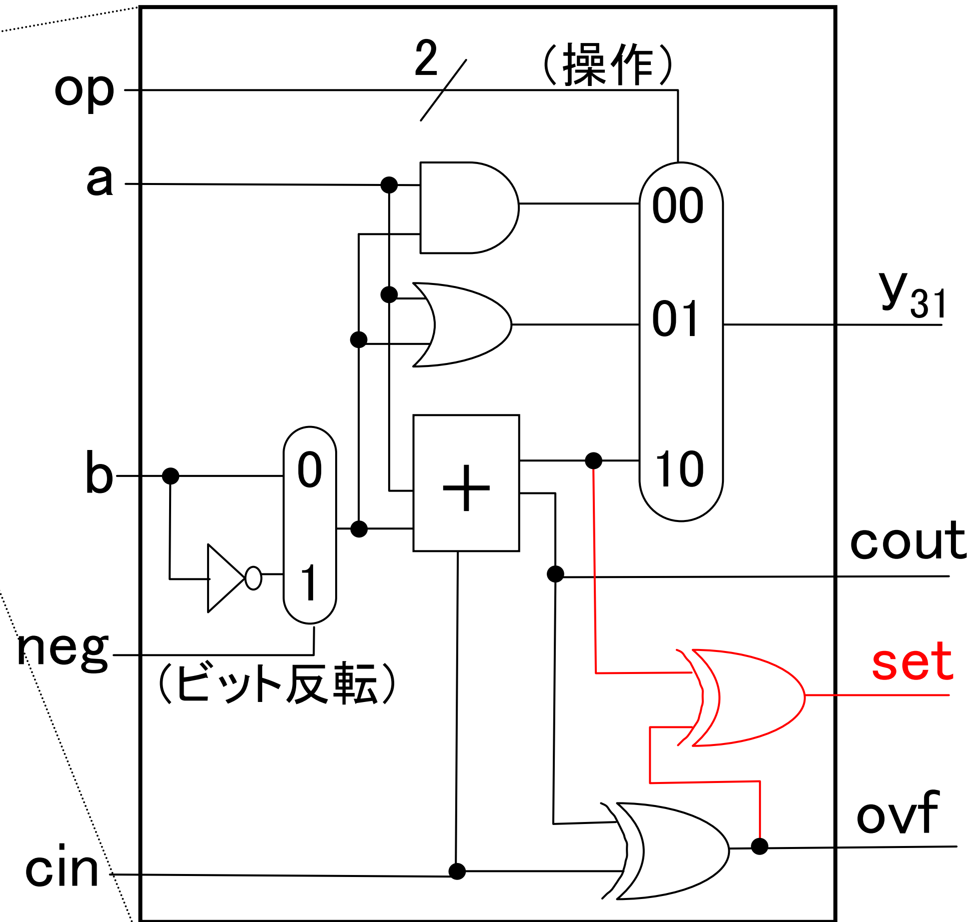
オーバーフローが生じて(ovf =1), 結果が正(y_{31}=0) → a ＜ b

つまり、ovf と y_{31}が不一致の場合はa<b

大小比較

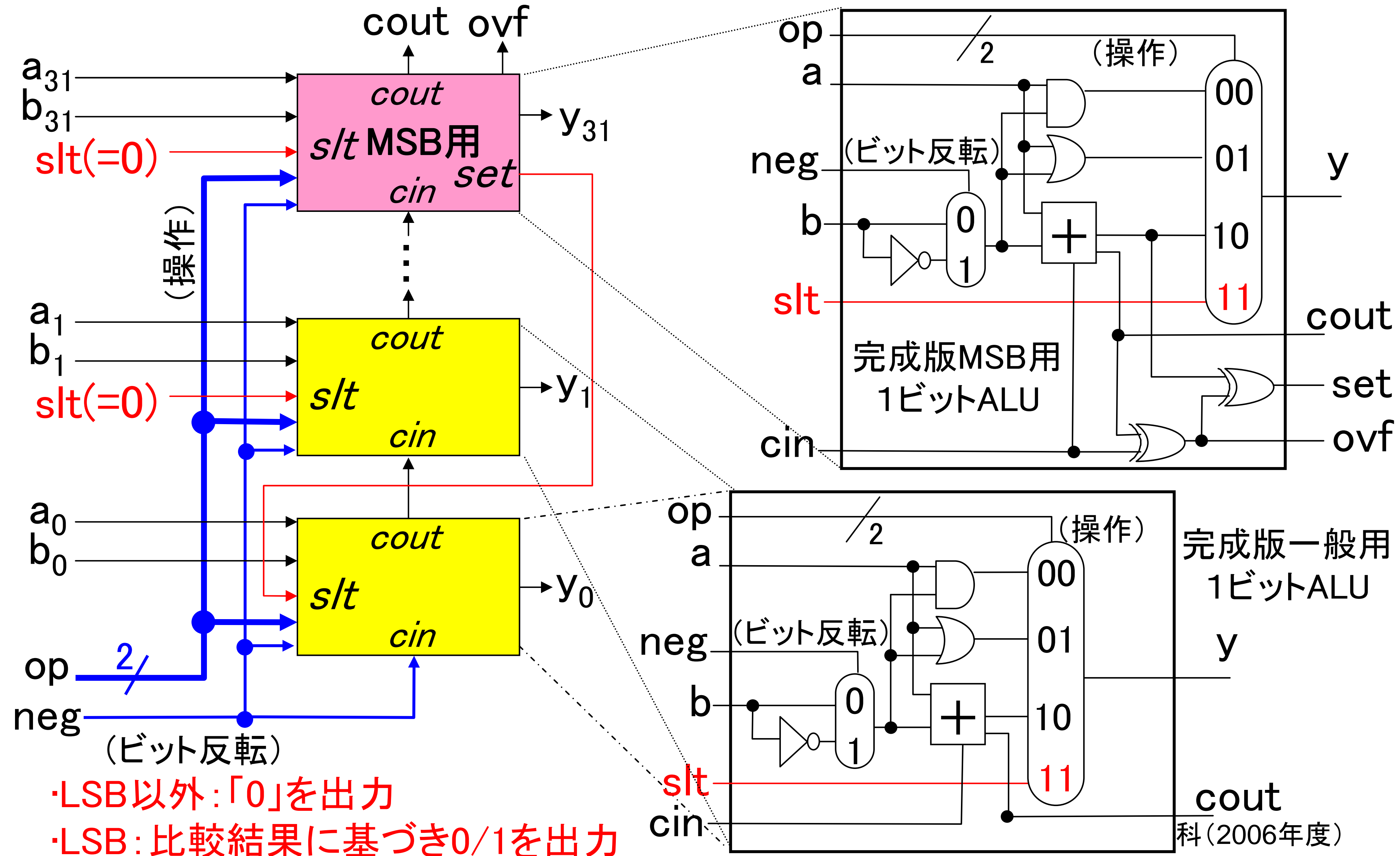
「a < b」時に‘1’となる出力信号setを生成



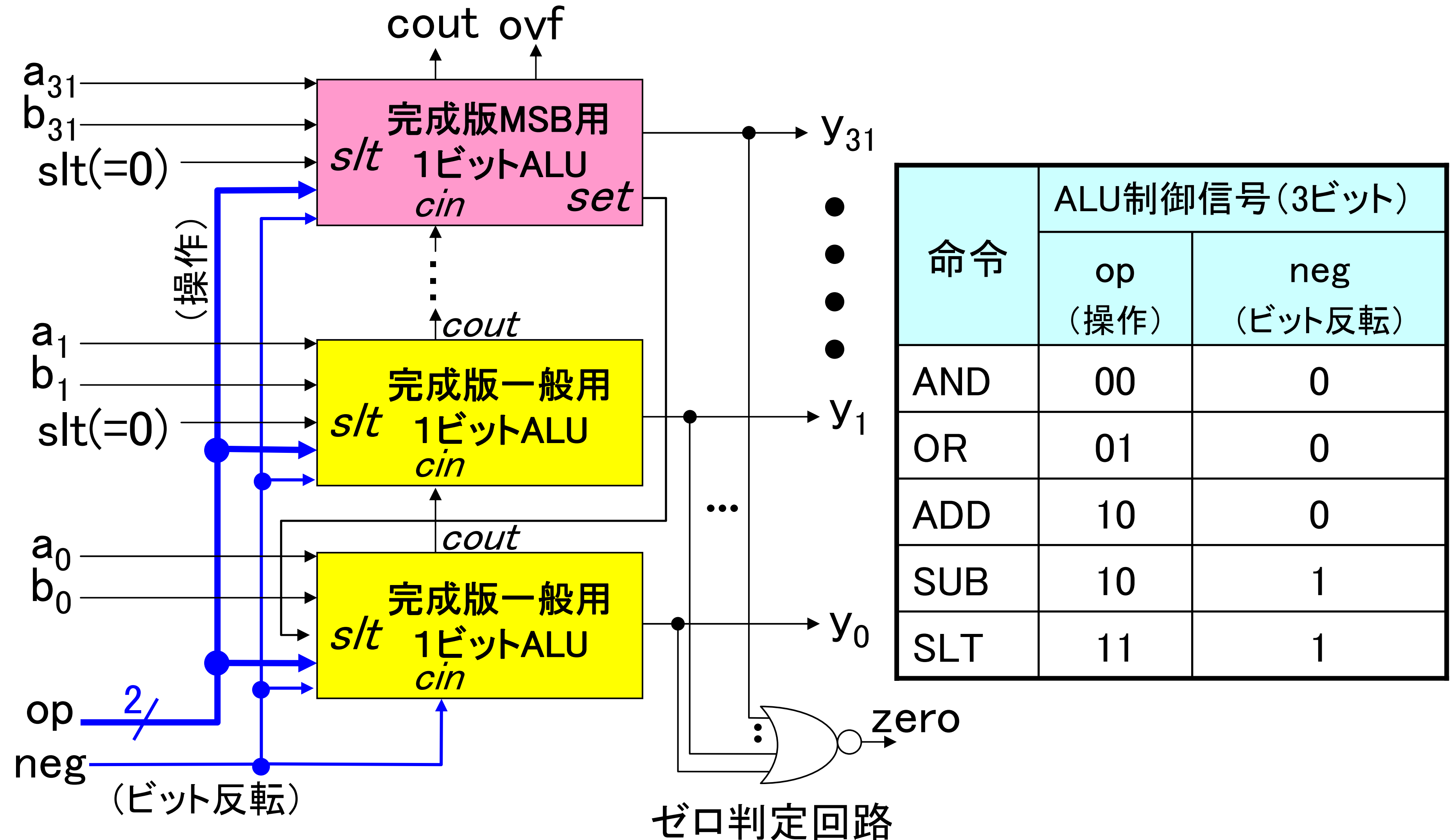


九州大学工学部電気情報工学科(2006年度)

比較器の設計(出力の生成)



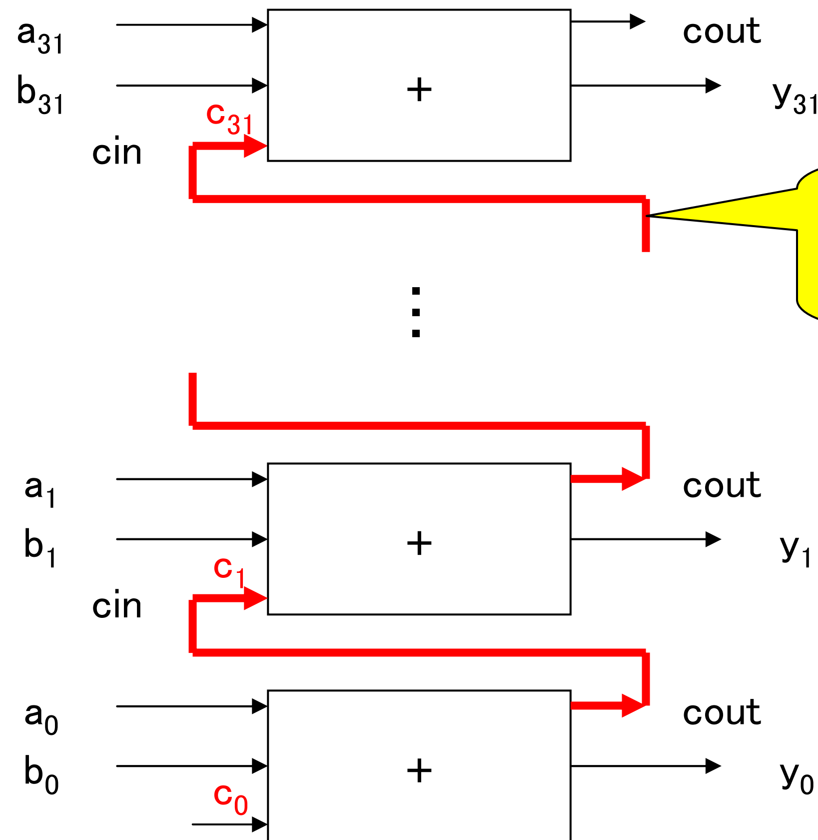
完成版32ビットALU



命令	ALU制御信号(3ビット)	
	op (操作)	neg (ビット反転)
AND	00	0
OR	01	0
ADD	10	0
SUB	10	1
SLT	11	1

加算器の高速化(1)

順次桁上げ加算器(Ripple Carry Adder)



加算器の高速化(2)

真理値表

a_0	b_0	c_0	y	c_1
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

$$c_1 = a_0 \cdot c_0 + b_0 \cdot c_0 + a_0 \cdot b_0 = (a_0 + b_0) \cdot c_0 + a_0 \cdot b_0$$

加算器の高速化(3)

32個の各加算器の回路は同じであるので,

$$c_1 = a_0 \cdot c_0 + b_0 \cdot c_0 + a_0 \cdot b_0 = (a_0 + b_0) \cdot c_0 + a_0 \cdot b_0$$

$$c_2 = a_1 \cdot c_1 + b_1 \cdot c_1 + a_1 \cdot b_1 = (a_1 + b_1) \cdot c_1 + a_1 \cdot b_1$$

…

$$c_{31} = a_{31} \cdot c_{31} + b_{31} \cdot c_{31} + a_{31} \cdot b_{31} = (a_{31} + b_{31}) \cdot c_{31} + a_{31} \cdot b_{31}$$

c_2 の右辺の c_1, c_3 の右辺の c_2, …を順次置換すると,

$$c_2 = ((a_0 + b_0) \cdot c_0 + a_0 \cdot b_0) \cdot (a_1 + b_1) + a_1 \cdot b_1$$

☺ c_1 がわからなくても, c_0 から c_2 が求められる.

$$c_3 = (((a_0 + b_0) \cdot c_1 + a_0 \cdot b_0) \cdot (a_1 + b_1) + a_1 \cdot b_1) \cdot (a_2 + b_2) + a_2 \cdot b_2$$

☺ c_2 がわからなくても, c_0 から c_3 が求められる.

…

☹ビット数が増えるほど, 指数関数的に式が長くなる(＝回路が大きくなる).

加算器の高速化(4)

$g_i = a_i \cdot b_i$, $p_i = a_i + b_i$ とすると,

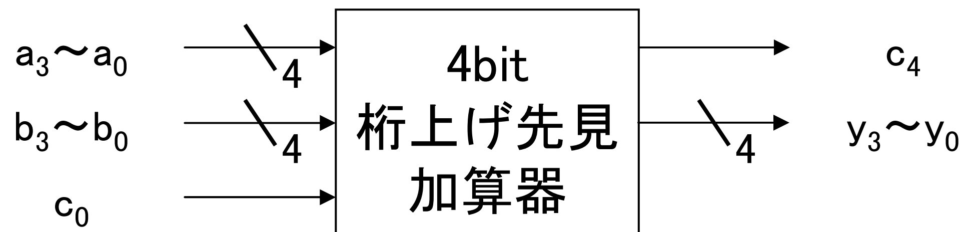
$$c_1 = g_0 + p_0 \cdot c_0$$

$$c_2 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$$

$$c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

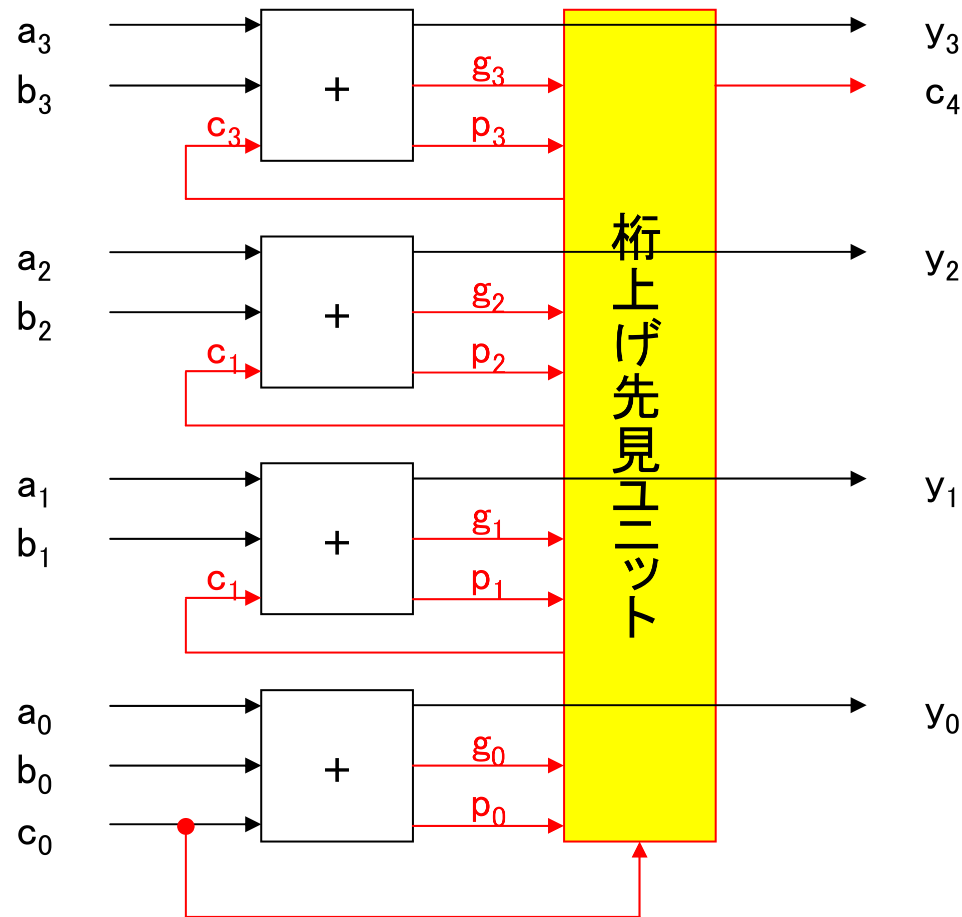
$$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

4bit 桁上げ先見加算器(Carry Look Ahead Adder)



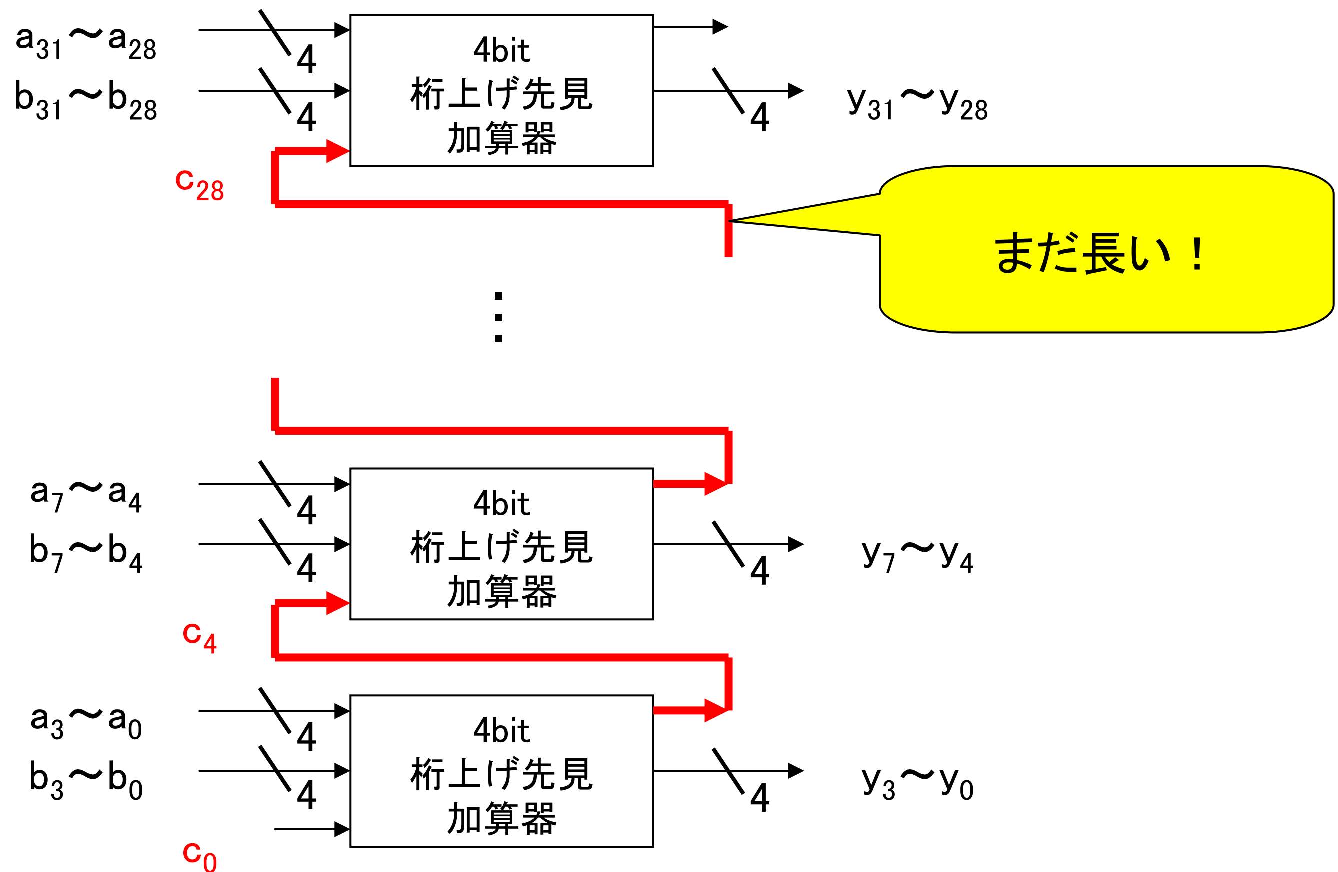
加算器の高速化(5)

4bit 桁上げ先見加算器(Carry Look Ahead Adder)



加算器の高速化(6)

32bit 加算器



加算器の高速化(7)

8個の各4bit桁上げ先見加算器の回路は同じであるので,

$$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

$$c_8 = g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4 + p_7 \cdot p_6 \cdot p_5 \cdot p_4 \cdot c_4$$

…

$$c_{32} = g_{31} + p_{31} \cdot g_{30} + p_{31} \cdot p_{30} \cdot g_{29} + p_{31} \cdot p_{30} \cdot p_{29} \cdot g_{28} + p_{31} \cdot p_{30} \cdot p_{29} \cdot p_{28} \cdot c_{28}$$

$P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0$, $P_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4$, …, $G_0 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$, $G_1 = g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4$, …として, c_8 の右辺の c_4, c_{12} の右辺の c_8, …を順次置換すると,

$$c_4 = G_0 + P_0 \cdot c_0$$

$$c_8 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_0$$

$$c_{12} = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0$$

…

加算器の高速化(8)

32bit 桁上げ先見加算器(Carry Look Ahead Adder)