計算機工学

第6章 演算アーキテクチャ

ALU Arithmetic and Logic Unit, 算術論理演算装置の構造について

教科書 コンピュータアーキテクチャの基礎, 柴山潔先生著(京都工芸繊維大学)

参考書 コンピュータの構成と設計, パターソン&ヘネシー

固定小数点の加減算

桁上げ carry: キャリ

借り borrow: ボロウ

小学校での筆算の知識と全く同じ.

精度と近似

- 計算機は，２進数ですべての数を表現している．
- 1/3が，10進数では循環小数になり，有限桁では正確に表現できない．２進数でも同じ．

例

３で割って，３をかけると，結果は99になる．

```
#include <stdio.h>
main()
{
  int a=100,b,c;
  b=a/3;  c=b*3;
  printf("%d %d %d\n",a,b,c);
}
```

同じことは，浮動小数点でも言える．

浮動小数点の場合

```
#include <stdio.h>
main()
{
  register float a=100,b,c=100.0/3.0,d,e;// registerは最適化を避けるため
  b=a/3.0;
  d=b*3.0;
  e=c*3.0;
  printf("a=%f b=%f c=%f d=%f e=%f\n",a,b,c,d,e);
}
```

実行結果

```
a=100.000000 b=33.333333 c=33.333332 d=100.000000 e=99.999996
```

固定小数点の加減算

$$(13)_{10} + (19)_{10} = (32)_{10}$$

(1)
$(0\ 1\ 1\ 0\ 1\ .)_2$
$(1\ 0\ 0\ 1\ 1\ .)_2$

(2)
$0\ 1\ 1\ 0\ 1$
$1\ 0\ 0\ 1\ 1$ (+
0

(3)
$0\ 0\ 1\ 1\ 0\ 1$
$0\ 0\ 1\ 1\ 0\ 1$ (+
$(1\ 0\ 0\ 0\ 0\ 0)_2$

$$(19)_{10} - (13)_{10} = (6)_{10}$$

(1)
$(1\ 0\ 0\ 1\ 1\ .)_2$
$(0\ 1\ 1\ 0\ 1\ .)_2$

(2)
$1\ 0\ 0\ 1\ 1$
$0\ 1\ 1\ 0\ 1$ (−
0

(3)
$1\ 0\ 0\ 1\ 1$
$0\ 1\ 1\ 0\ 1$ (−
$(0\ 0\ 1\ 1\ 0)_2$

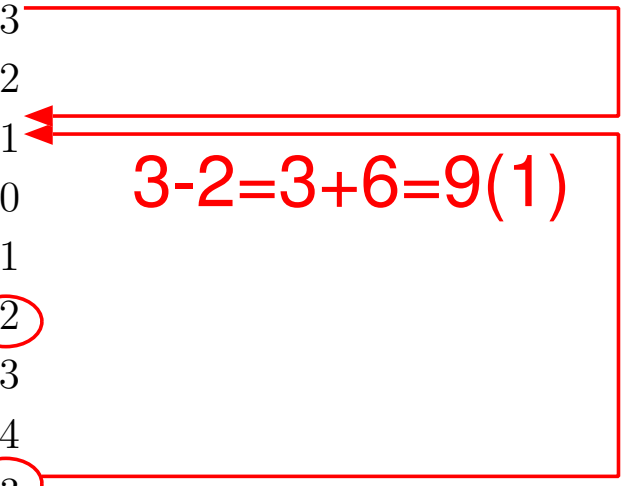
２進数の循環輪(第３章資料再掲)

10進(符号無視)	２進	２の補数	１の補数	
3(11)	1011	3	3	
2(10)	1010	2	2	
1(9)	1001	1	1	
0(8)	1000	0	0	(桁あふれは無視)
7	0111	−1	0	
6	0110	−2	−1	
5	0101	−3	−2	
4	0100	−4	−3	
3	0011	3	3	
2	0010	2	2	
1	0001	1	1	
0	0000	0	0	

- ３ビットでの２の補数表現(４ビット目は無視): ２の補数なら３から−４までを表現可能
- １の補数は全ビット反転. ２の補数＝１の補数＋1
- 補数を用いて演算することで, 加減算が同じ手続きで実行できる.

2の補数の加減算

10進	2進	2の補数
3	1011	3
2	1010	2
1	1001	1
0	1000	0
7	0111	−1
6	0110	−2
5	0101	−3
4	0100	−4
3	0011	3
2	0010	2
1	0001	1
0	0000	0



- 2の補数では，2減ずる (表で2段下に下がる) ことは，6加算する (表で6段上に上がる) ことと同じ.
- 6を−2と定義すれば (2の補数の定義)，加算=減算となる.
- ただし，桁あふれに注意.（後述)

1の補数の加減算

10進	2進	2の補数	1の補数
3(11)	1011	3	3
2(10)	1010		
1(9)	1001	1	1
0(8)	1000	0	0
	111	−1	0
6	110	−2	−1
5	101	−3	−2
4	100	−4	−3
3	011	3	3
2	010	2	2
1	001	1	1
0	000	0	0

エンドアラウンドキャリ

エンドキャリ

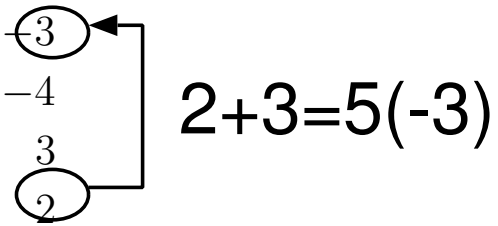
3-2=3+5=8(0)

2-3=2+4=6(-1)

- 桁あふれ(エンドキャリ)が出る場合は，キャリを1桁目に伝搬させる(エンドアラウンドキャリ)

オーバーフロー

10 進	2 進	2 の補数
3	1011	3
2	1010	2
1	1001	1
0	1000	0
7	0111	−1
6	0110	−2
5	0101	−3
4	0100	−4
3	0011	3
2	0010	2
1	0001	1
0	0000	0



2+3=5(-3)

- 2+3は，5となり，4ビットで表せる数の範囲(3から-4)を超えるので，結果が正しくなくなる．
- 正数と負数の加算では，オーバーフローは発生しない．

C言語による確認: オーバーフロー

```
int main ()
{
  short a=30000,b=30000,c;
  c=a+b;
  printf("%d\n",c);
}
```

結果

```
-5536
```

- shortは符号付き16ビットなので, 表せる範囲は,

$$(0111, 1111, 1111, 1111)_2 = 32767$$

から,

$$(1000, 0000, 0000, 0000)_2 = -32768$$

まで

- 30000+30000はオーバーフローする.

2の補数表現の固定小数点の加減算

桁あふれ(overflow)がない場合 $(-19)_{10} - (13)_{10} = -(32)_{10}$

$$\begin{array}{ll} (19)_{10} = (010011)_2 & -(19)_{10} = (101101)_2 \\ \rightarrow & \\ (13)_{10} = (001101)_2 & -(13)_{10} = \underline{(110011)_2}(+ \\ & \quad *100000 \end{array}$$

桁あふれ(overflow)がある場合 $(-10)_{10} - (7)_{10} = -(17)_{10}$

$$\begin{array}{ll} (10)_{10} = (01010)_2 & -(10)_{10} = (10110)_2 \\ \rightarrow & \\ (7)_{10} = (00111)_2 & -(7)_{10} = \underline{(11001)_2}(+ \\ & \quad *01111 \end{array}$$

演算幅の拡張

- ビット幅の異なる値同士の演算を行なうときに，値同士のビット幅をそろえる．

４ビット	→ 拡張 →	８ビット
正整数0011(3)		00000011
負整数1101(−3)		11111101(２の補数)
1100(−3)		11111100(１の補数)
正小数0.011		0.01100000
負小数1.101		1.10100000(２の補数)
1.101		1.10111111(１の補数)

正整数 足りない分だけ，上位ビットに０を補う

負整数 足りない分だけ，上位ビットに１を補う

正小数 足りない分だけ，下位ビットに０を補う

負小数 足りない分だけ，下位ビットに0(２の補数の場合)または，1(１の補数の場合)を補う

C言語における符号拡張

- C言語でも異なるバイト数の変数間の代入，演算などでは自動的に符号拡張される．

```
#include <stdio.h>
int main()
{
  char  a;
  int   b;
  a=-1;
  b=a; // 自動的に符号拡張される
  printf("a=%d b=%d\n",a,b);
  printf("a=%02x b=%08x\n",(unsigned char)a, (unsigned int)b);
}
```

実行結果

```
a=-1 b=-1
a=ff b=ffffffff
```

固定小数点の加減算機構

半加算器 (Half Adder) 入力は2個の1ビット2進数．出力は加算結果 (Sum) と桁上げ (Carry)

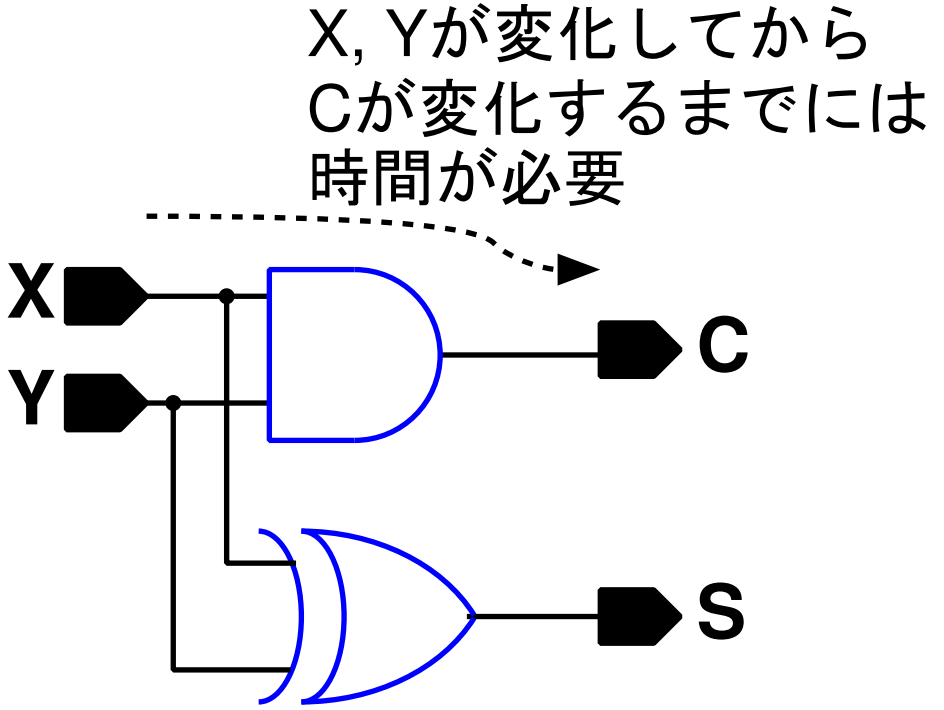
入力		出力		和
X	Y	C	S	
0	0	0	0	$(0)_{10}$
0	1	0	1	$(1)_{10}$
1	0	0	1	$(1)_{10}$
1	1	1	0	$(2)_{10}$

- Sは，XとYのXOR(排他的論理和)

$$S = X \oplus Y$$

- Cは，XとYのAND(論理積)

$$C = X \cdot Y$$



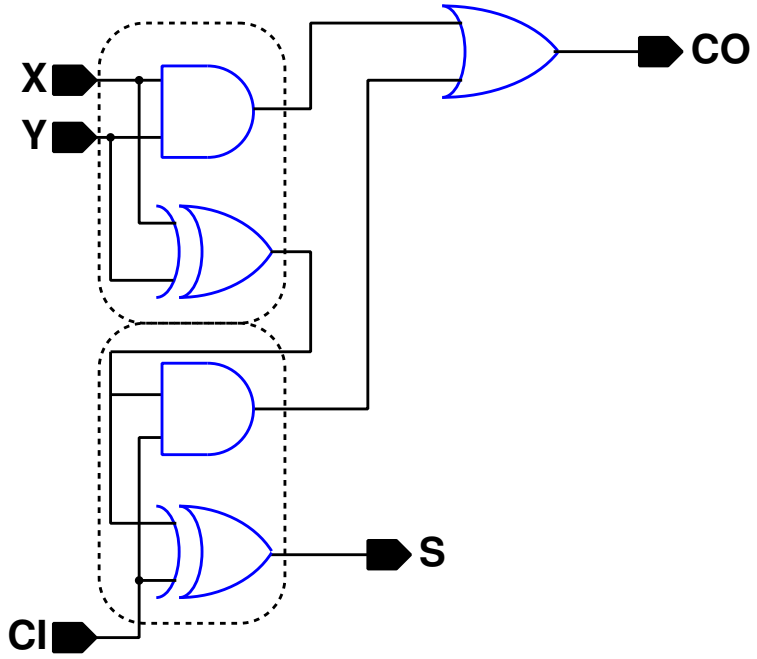
教科書P172 図6.12より

- 加算には，2つの値と**桁上げ**が入力として必要

全加算器(Full Adder)

- 入力は2個の1ビット2進数と1個の1ビット桁上げ(Carry, CI). 出力は加算結果(Sum)と桁上げ(Carry, CO)

入力			出力		和
X	Y	CI	CO	S	
0	0	0	0	0	$(0)_{10}$
0	0	1	0	1	$(1)_{10}$
0	1	0	0	1	$(1)_{10}$
0	1	1	1	0	$(2)_{10}$
1	0	0	0	1	$(1)_{10}$
1	0	1	1	0	$(2)_{10}$
1	1	0	1	0	$(2)_{10}$
1	1	1	1	1	$(3)_{10}$



教科書P173 図6.13より

- Sは, XとYのXOR $(S_0 = X \oplus Y)$とCIのXOR

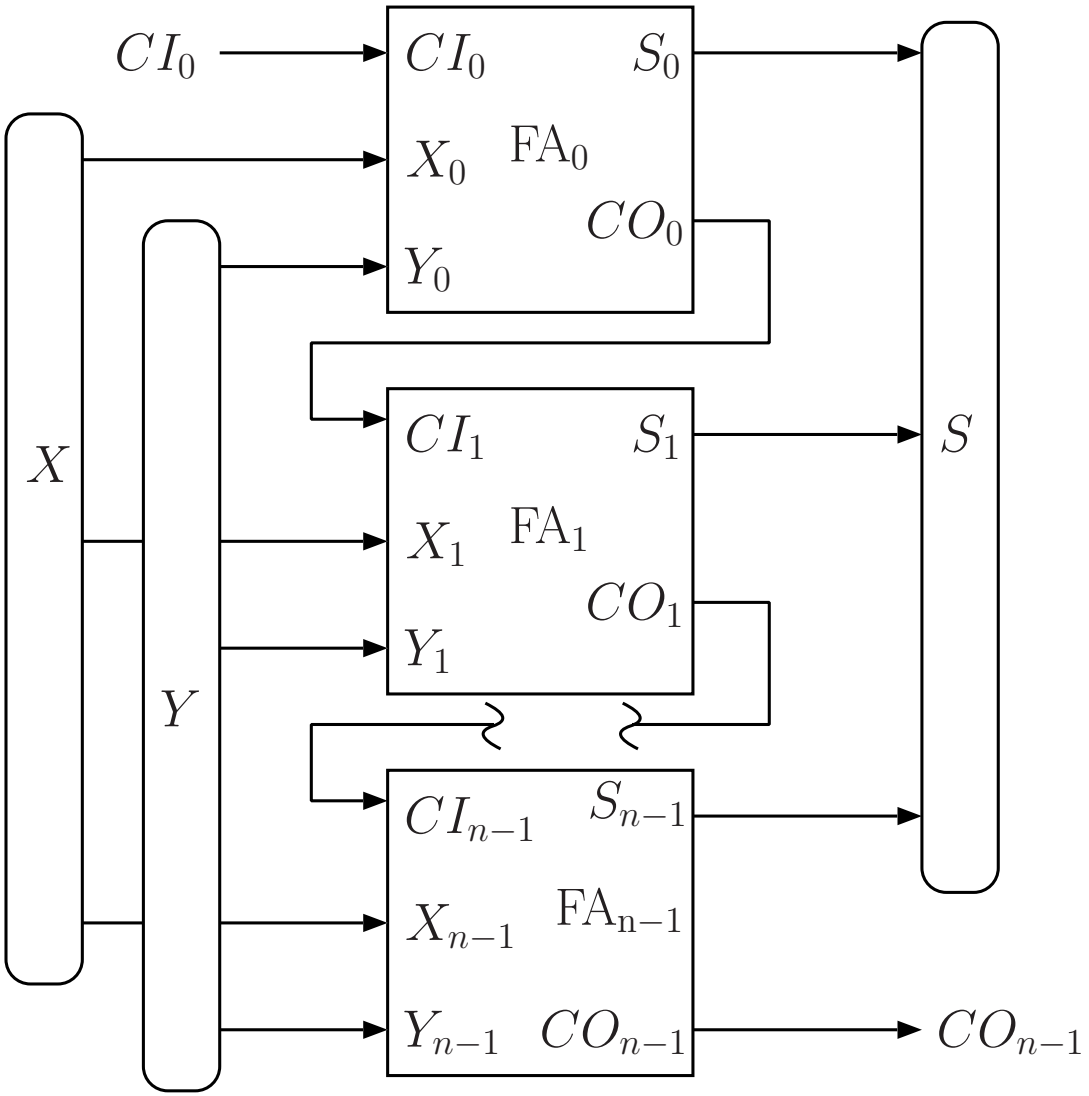
$$S = S_0 \oplus CI = (X \oplus Y) \oplus CI$$

- COは, XとYのAND $(X \cdot Y)$が1か, S_0とCIのANDが1のときに1

$$CO = X \cdot Y + S_0 \cdot CI = X \cdot Y + (X \oplus Y) \cdot CI = X \cdot Y + Y \cdot CI + CI \cdot X$$

桁上げ伝播加算器(Ripple Carry Adder: RCA)

- n 個の全加算器を直列に接続して，n ビットの加算器を作る． CO_i と，CI_{i+1} を接続
- 回路が簡単
- 桁上げ伝搬に時間がかかる．



$X = (X_{n-1}, ..X_1, X_0)_2$ 等と定義

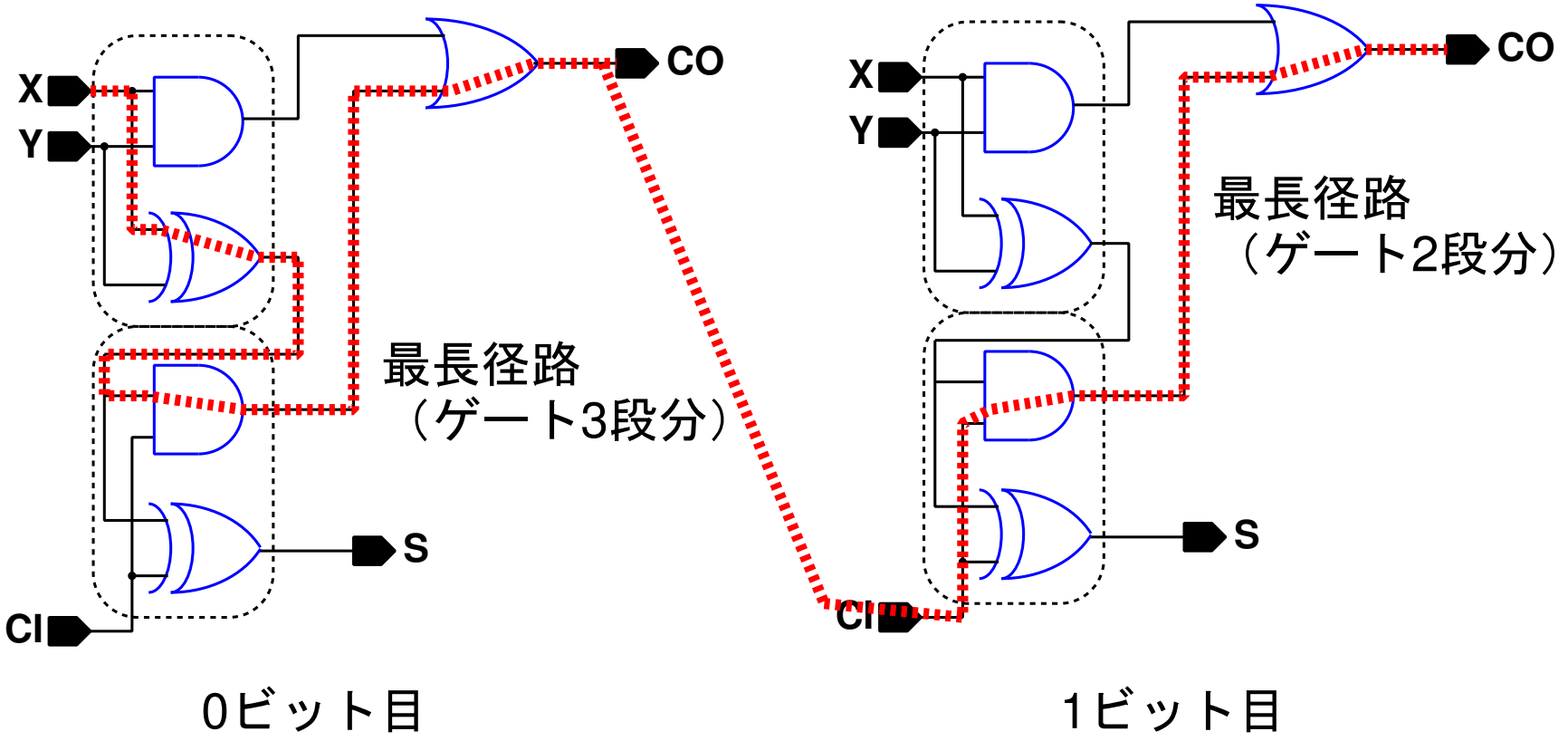
教科書 P174 図6.14より

最悪の桁上げ伝播(8ビット)

X		1	1	1	1	1	1	1	1	
$+Y, CI_0$		0	0	0	0	0	0	0	1	, 0
$X \oplus Y$		1	1	1	1	1	1	1	0	
CO		0	0	0	0	0	0	0	1	最初の時点
CO		0	0	0	0	0	0	1	1	CO_0が1ビット目に伝わると
CO		1	1	1	1	1	1	1	1	最終の時点
CO_7, S	1,	0	0	0	0	0	0	0	0	

- キャリーは, LSBからMSBまで順に伝播していく.
- キャリー伝播がクリティカルパス(もっとも遅い信号伝送経路)となる.

桁上げ伝播加算器のクリティカルパス



- X, Y は全ビット同時に与えられるとする．
- n ビットの加算器の場合クリティカルパス長 L は，

$$L = 2n + 1$$

$n = 32$ で，65段，n=4 で，9段

桁上げ先見加算器(Carry Look-Ahead Adder, CLA)

- 桁上げ信号が伝播する論理段数を減らして，高速化する．
- キャリー伝搬の条件は，「その桁の二つ(X_i, Y_i)の入力がともに1であるか，どちらかが1で，ひとつしたの桁のキャリー出力(CI_i)が1のとき」

$$
\begin{aligned}
CI_{i+1} &= COi = X_i \cdot Y_i + (X_i + Y_i) \cdot CI_i \\
&= G_i + P_i \cdot CI_i \\
G_i &= X_i \cdot Y_i \quad \text{Carry Generate} \\
P_i &= X_i + Y_i \quad \text{Carry Propagate}
\end{aligned}
$$

桁上げ伝播加算器 そのまま漸化式を順に実行する．オーダ(order，次数)は，n

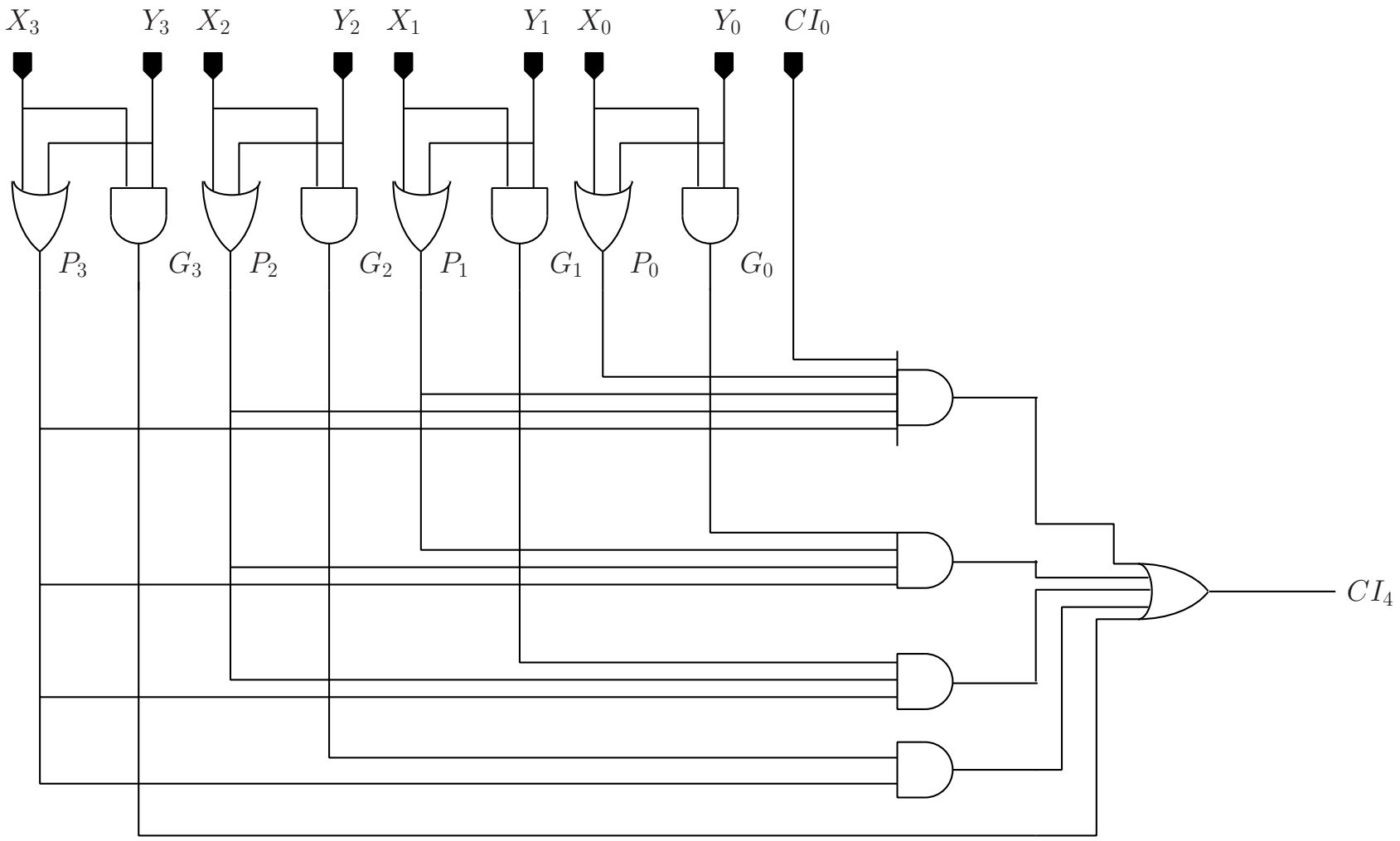
桁上げ先見加算器 漸化式を展開して実行する．オーダは，$\log_2 n$

桁上げ先見加算の原理(1)

$$
\begin{aligned}
CI_1 &= G_0 + P_0 \cdot CI_0 \\
CI_2 &= G_1 + P_1 \cdot CI_1 \\
&= G_1 + P_1 \cdot (G_0 + P_0 \cdot CI_0) \\
&= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot CI_0 \\
CI_3 &= G_2 + P_2 \cdot CI_2 \\
&= G_2 + P_2 \cdot (G_1 + P_1 \cdot CI_1) \\
&= G_2 + P_2 \cdot (G_1 + P_1 \cdot (G_0 + P_0 \cdot CI_0)) \\
&= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot CI_0 \\
CI_4 &= G_3 + P_3 \cdot CI_3 \\
&= G_3 + P_3 \cdot (G_2 + P_2 \cdot CI_2) \\
&= G_3 + P_3 \cdot (G_2 + P_2 \cdot (G_1 + P_1 \cdot CI_1)) \\
&= G_3 + P_3 \cdot (G_2 + P_2 \cdot (G_1 + P_1 \cdot (G_0 + P_0 \cdot CI_0))) \\
&= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot CI_0 \\
&= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot CI_0
\end{aligned}
$$

- CI_nを，それぞれ組合せ回路で実現する．

CI_4を実現する組合せ回路

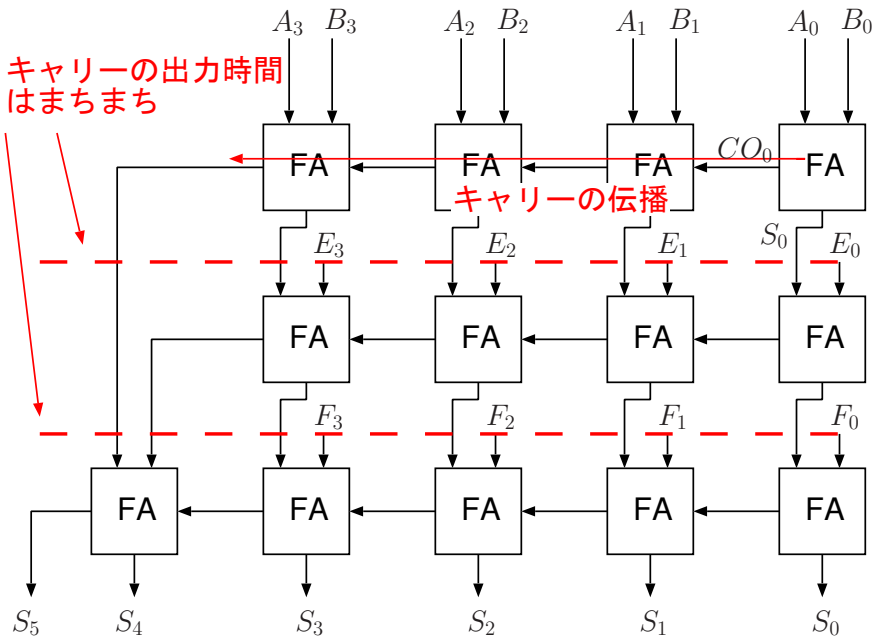


- $n = 4$で, 3段=$(1+\log_2 4)$.
- 実際には, 論理ゲートの入力は最大4まで.

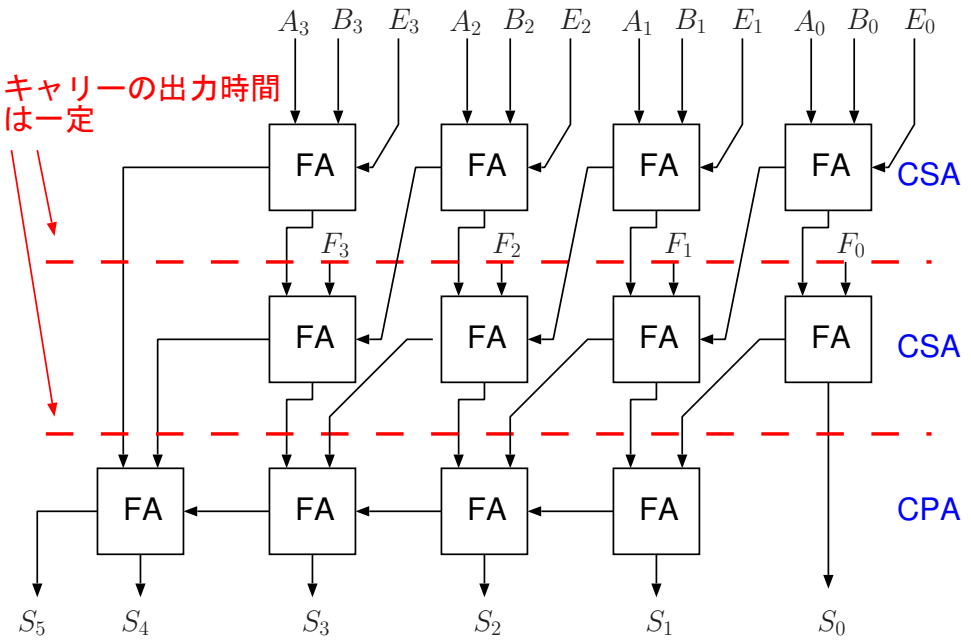
桁上げ保存加算器（Carry Save Adder, CSA）

- 多項の加算のさいに，全ビットのキャリー伝播を1回にする．
- パイプライン化しやすい．
- 乗算器に用いられる．

$S = A + B + E + F$: CO_5は，Sの最上位ビット(S_5)，CI_0は0



桁上げ伝搬加算器による多項の加算



桁上げ保存加算器による多項の加算

教科書P178 図6.17より

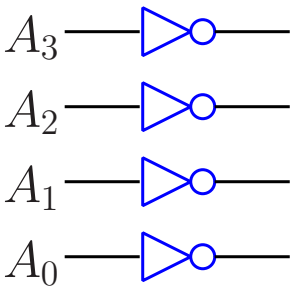
補数器

1の補数 単に，否定を取るだけ．

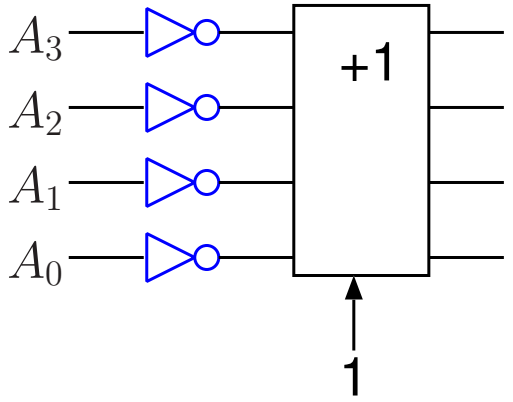
- インバータ (NOT ゲート) だけで良い．

2の補数 否定を取って1を足す．

- +1 を行なう演算回路 (インクリメンタ，incrementor) も必要．
- インクリメンタはほとんど，加算器と同じ回路規模．



1の補数器



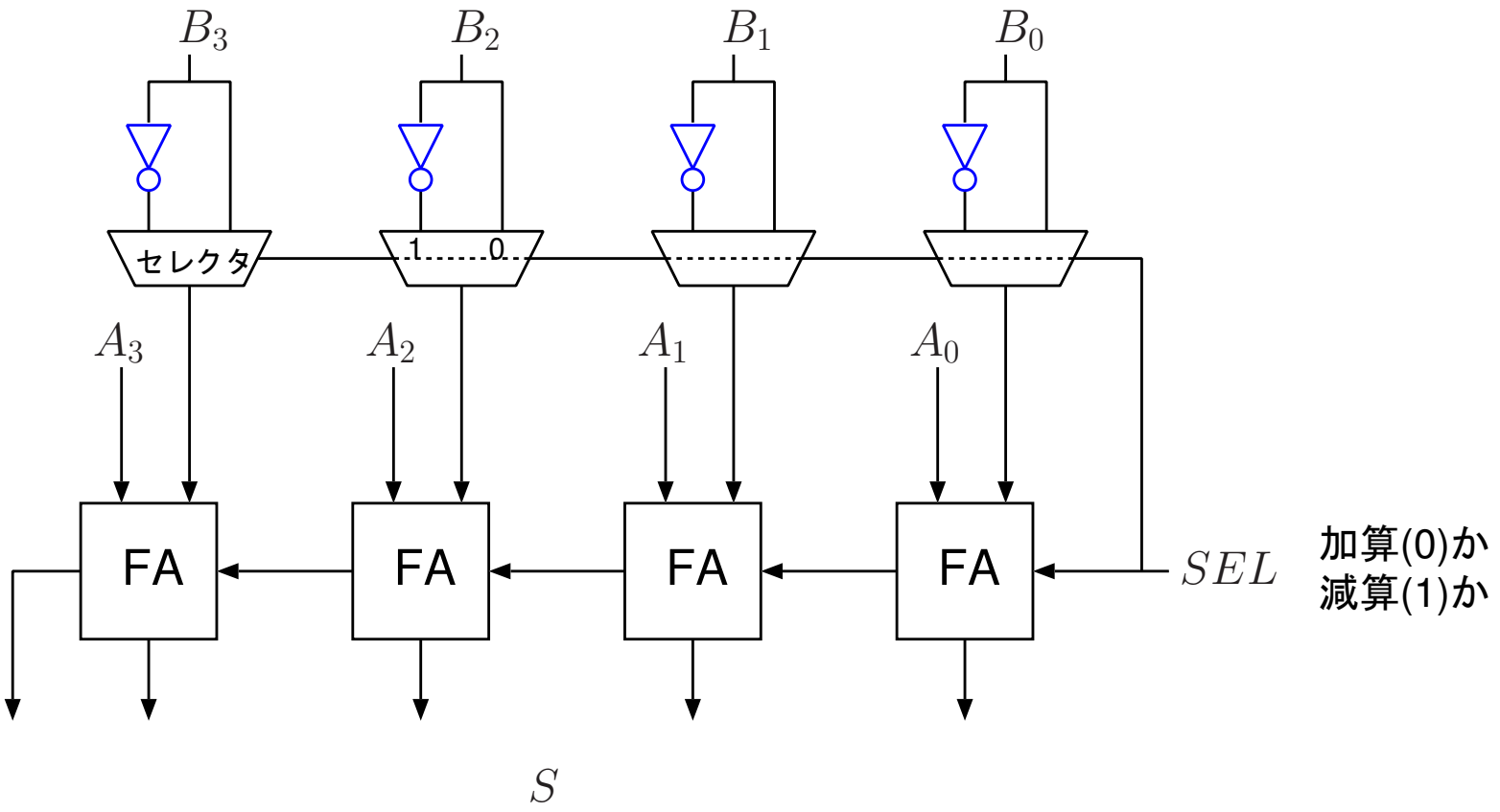
$\overline{A}+1$

2の補数

2の補数器

教科書P179 図6.18より

２の補数を用いた加減算器



- SELが０の時は，$S = A + B$
- SELが１の時は，$S = A + \overline{B} + 1 = A +$ (Bの２の補数)

固定小数点加減算機構のコンディション(Condition)

$$CO, F = X + Y$$

オーバーフロー(overflow: OF) 入力値の最上位ビットが同じ(同符号)で，演算結果の最上位ビットが入力と異符号になるときに発生.
($X_{n-1} == Y_{n-1}$ かつ $F_{n-1} \neq X_{n-1}$)

- 演算結果がそのビット幅で表現できる範囲を超える場合． 16 ビット符号付き(short)の場合，30000+30000．(VF)

桁上げ(Carry : C) $CO = C_{n-1}$ が 1 の場合．(CF)

符号(sign : S) F_{n-1} (NF)

ゼロ(zero : Z) 演算結果(F)が 0 の場合 (ZF)

括弧内最後のアルファベットは，kuechip2 でのフラグ名

8ビットマイクロプロセッサ(Kuechip2)の命令コード表

Bcc	0 0 1 1 cc	◎	Branch cc	条件が成立すれば

cc : Condition Code			
A	0 0 0 0	Always	常に成立
VF	1 0 0 0	on oVerFlow	桁あふれ $VF = 1$
NZ	0 0 0 1	on Not Zero	$\neq 0$ $ZF = 0$
Z	1 0 0 1	on Zero	$= 0$ $ZF = 1$
ZP	0 0 1 0	on Zero or Positive	≥ 0 $NF = 0$
N	1 0 1 0	on Negative	< 0 $NF = 1$
P	0 0 1 1	on Positive	> 0 $(NF \vee ZF) = 0$
ZN	1 0 1 1	on Zero or Negative	≤ 0 $(NF \vee ZF) = 1$
NI	0 1 0 0	on No Input	$IBUF\_FLG\_IN = 0$
NO	1 1 0 0	on No Output	$OBUF\_FLG\_IN = 1$
NC	0 1 0 1	on Not Carry	$CF = 0$
C	1 1 0 1	on Carry	$CF = 1$
GE	0 1 1 0	on Greater than or Equal	≥ 0 $(VF \oplus NF) = 0$
LT	1 1 1 0	on Less Than	< 0 $(VF \oplus NF) = 1$
GT	0 1 1 1	on Greater Than	> 0 $((VF \oplus NF) \vee ZF) = 0$
LE	1 1 1 1	on Less than or Equal	≤ 0 $((VF \oplus NF) \vee ZF) = 1$

固定小数点の乗算機構

積(Product) = 被乗数(Multiplicand) × 乗数(Multiplier)

- 乗算を，筆算(加算の繰り返し)で行なう．
- 10進の場合は，筆算の場合に，乗算(九九の知識)が必要であるが，2進の場合は，不要(×1のみ)．
- nビットの値同士の積は，$2n$ビットになる．

			1	5	1	2				1	0	1	1	(11)10
		×	2	0	4	3			×	1	1	0	1	(13)10
			4	5	3	6				1	0	1	1	部分積
		6	0	4	8				0	0	0	0	0	部分積
	0	0	0	0				1	0	1	1	0	0	部分積
+	3	0	2	4			+	1	0	1	1	0	0	0	部分積
	3	0	8	9	0	1	6	1	0	0	0	1	1	1	1	(143)10

直接乗算法

繰り返し乗算法

$$P = X \times Y$$

$P = 0;$
`for`$(i = 0; i < n; i++)\{$

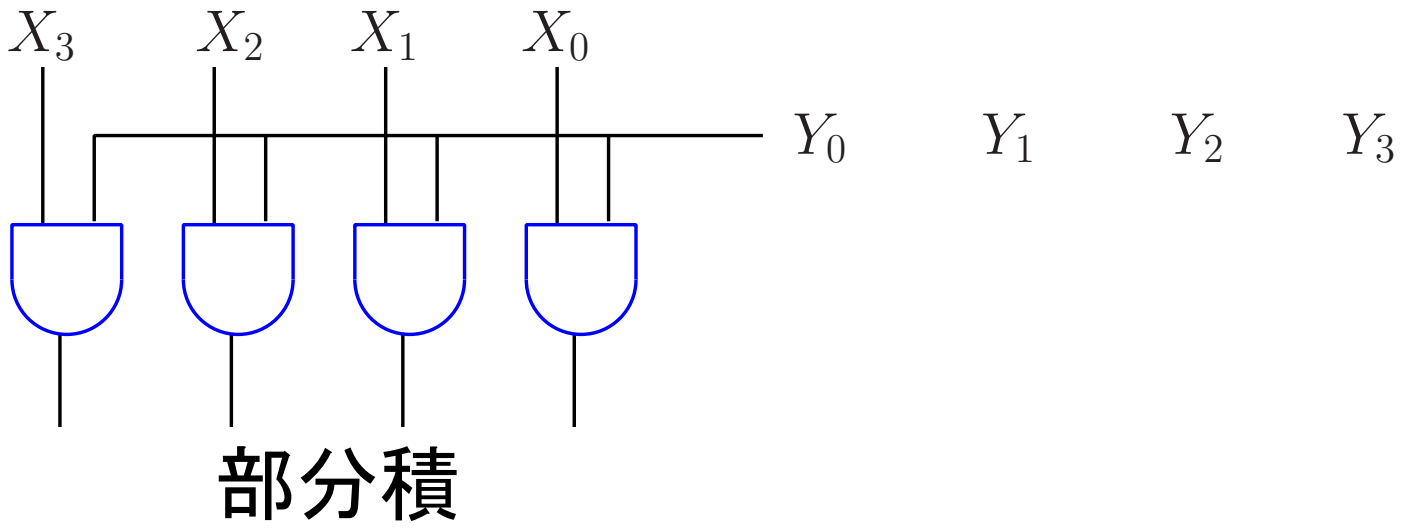
1. if$(Y_i == 0)$ $PP = 0$;//乗数の第iビット(Y_i)が0なら，部分積は0
2. `else` $PP = X$; //乗数の第iビットが1なら，部分積はX
3. $P+ = (PP << i)$; // 部分積をiビットシフトさせて積に加える

$\}$

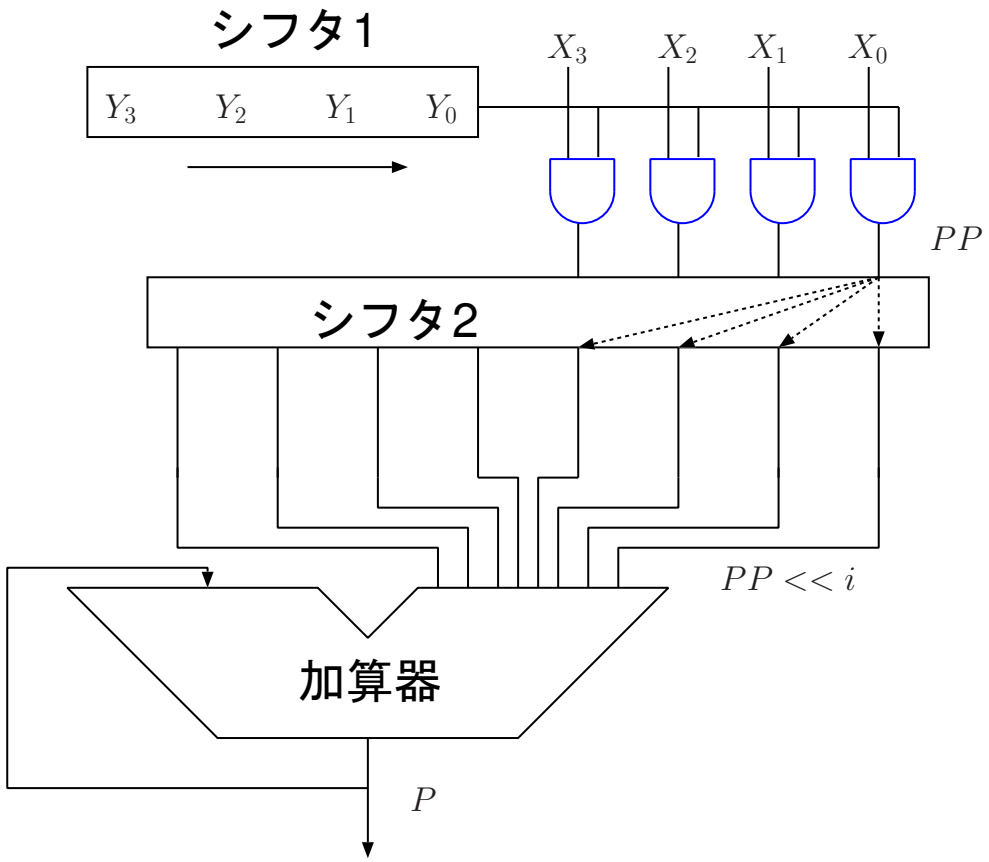
部分積発生回路

下記の `if/else` を，AND ゲートで実現.

1. if($Y_i == 0$) $PP = 0$;//乗数の第iビット(Y_i)が0なら，部分積は0
2. else $PP = X$; //乗数の第iビットが1なら，部分積はX



繰り返し乗算器



教科書P183 図6.24より

1. シフタ1により，Yを1ビットずつシフトさせて，ANDゲートに入力
2. シフタ2により，PPをiビットシフトさせて，加算器に入力
3. 加算器は，　現在の積 P と，$PP << i$を加算する．

C言語でまじめに実装すると (multi.c)

```
#include<stdio.h>
main ()
{
  unsigned char x=45,y=58;// x=00101101=45, y=00111010=58
  unsigned short i,p=0,a=0;
  for(i=0;i<8;i++){
    if((y&(1<<i))!=0){// if 1
      p+=(x<<i);
      a++;
    }
    printf("p(%d)=%d\n",i,p);
  }
  printf("p=%d, number of addition=%d\n",p,a);
}
```

高速乗算法

- 繰り返し乗算法は，乗数のビット幅に比例した時間がかかる．
- 乗算を高速かつ小面積で行なうための様々な方法が提案されている．
  - ブース (Booth) の方法
  - 並列乗算器
  - ウォリス (ワレス，Wallace) 木

ブース(Booth)の方法

- $252 \times 999 = 252 \times (1000 - 1) = 252 \times 1000 - 252$ のように計算する.
- 10進数での×10は, 2進数では×2.
- ×2は, 単なるシフト演算と同じ.

$$
\begin{aligned}
(18)_{10} \times (2)_{10} &= (36)_{10} \\
(10010)_2 \times (10)_2 &= (100100)_2
\end{aligned}
$$

- ブースのアルゴリズムにより, 部分積の数が減らせる. ↑ 加算の数が減らせる. ↑ 高速になる.

$$
\begin{aligned}
(26)_{10} \times (14)_{10} &= 26 \times (8 + 4 + 2) \quad \text{3個の部分積} \\
&= 26 \times (16 - 2) \quad \text{2個の部分積}
\end{aligned}
$$

ブース(Booth)の方法(2)

$$
\begin{aligned}
X \times Y &= (26)_{10} \times (14)_{10} \\
&= (11010)_2 \times (01110)_2 \\
&= (11010)_2 \times ((10000)_2 - (00010)_2) \\
&= (26)_{10} \times ((16)_{10} - (2)_{10})
\end{aligned}
$$

アルゴリズム

1. ($Y_{-1} = 0$,) $Y_0 = 0$なので, $PP_0 = 0$ (パタン1, 00)
2. $Y_0 = 0$, $Y_1 = 1$なので, $PP_1 = -X << 1, P+ = PP_1$ (パタン2, 10)
3. $Y_1 = 1$, $Y_2 = 1$なので, $PP_2 = 0$ (パタン1, 11)
4. $Y_2 = 1$, $Y_3 = 1$なので, $PP_3 = 0$ (パタン1, 11)
5. $Y_3 = 1$, $Y_4 = 0$なので, $PP_4 = X << 4, , P+ = PP_4$ (パタン3, 01)

- 一つ前のビットの値と現在のビットの値によって, パタン1から3まで, 処理を変える.

ブースの方法(3)

$$
\begin{aligned}
X \times Y &= (14)_{10} \times (26)_{10} \\
&= (01110)_2 \times (11010)_2 \\
&= (01110)_2 \times ((100,000)_2 - (001,000)_2 + (000,100)_2 - (000,010)_2 \\
&= (14)_{10} \times ((32)_{10} - (8)_{10} + (4)_{10} - (2)_{10}
\end{aligned}
$$

- 前ページの方法でやると，部分積が４個になってしまう．
- 通常の乗算 (部分積３個) より多い．
- 010 の場合は，通常の乗算で行なわなければならない．

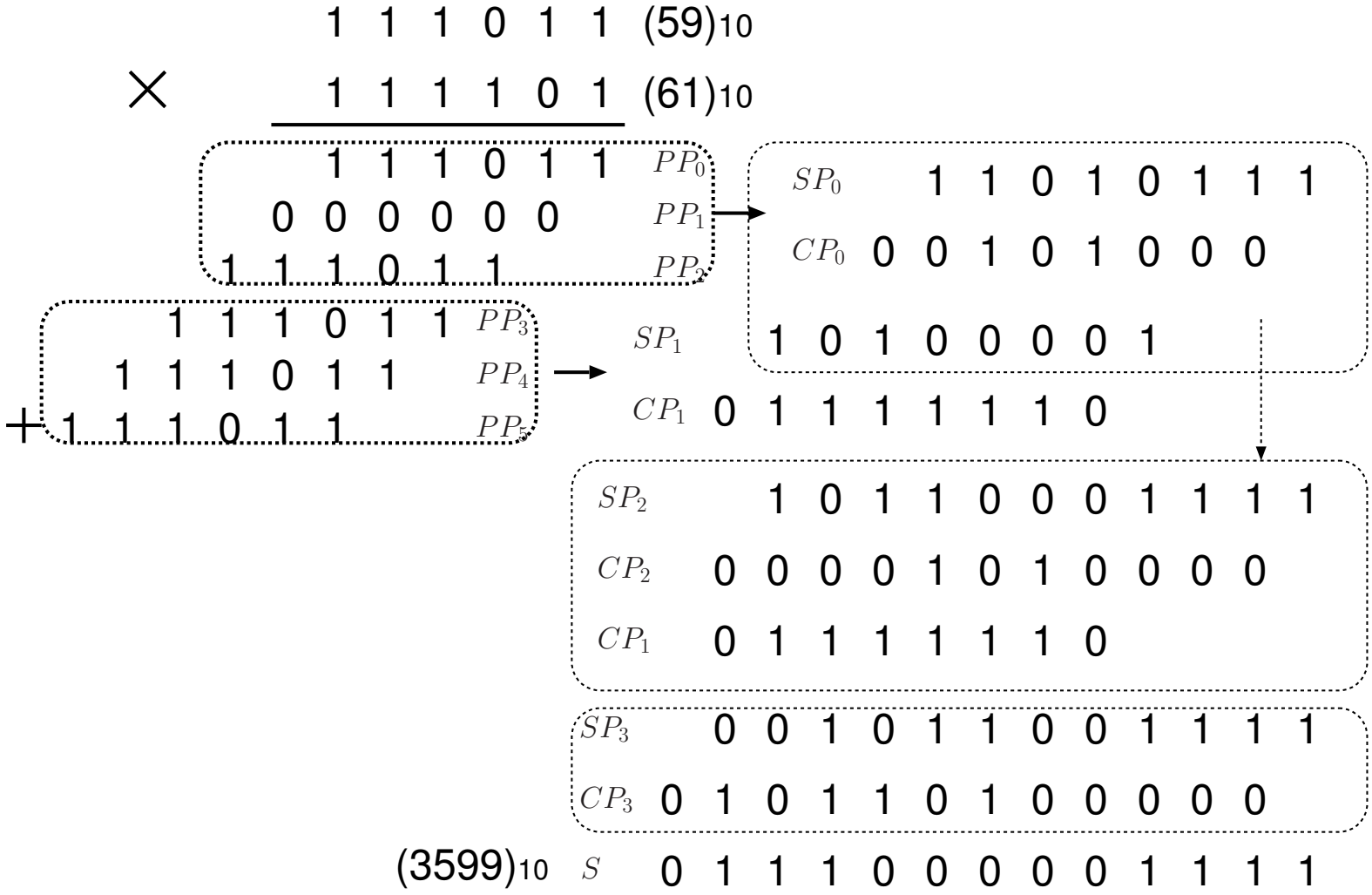
  010 通常の乗算
  01+0 ブースの方法 (1+は，１が２回以上続くという意味)

並列乗算器

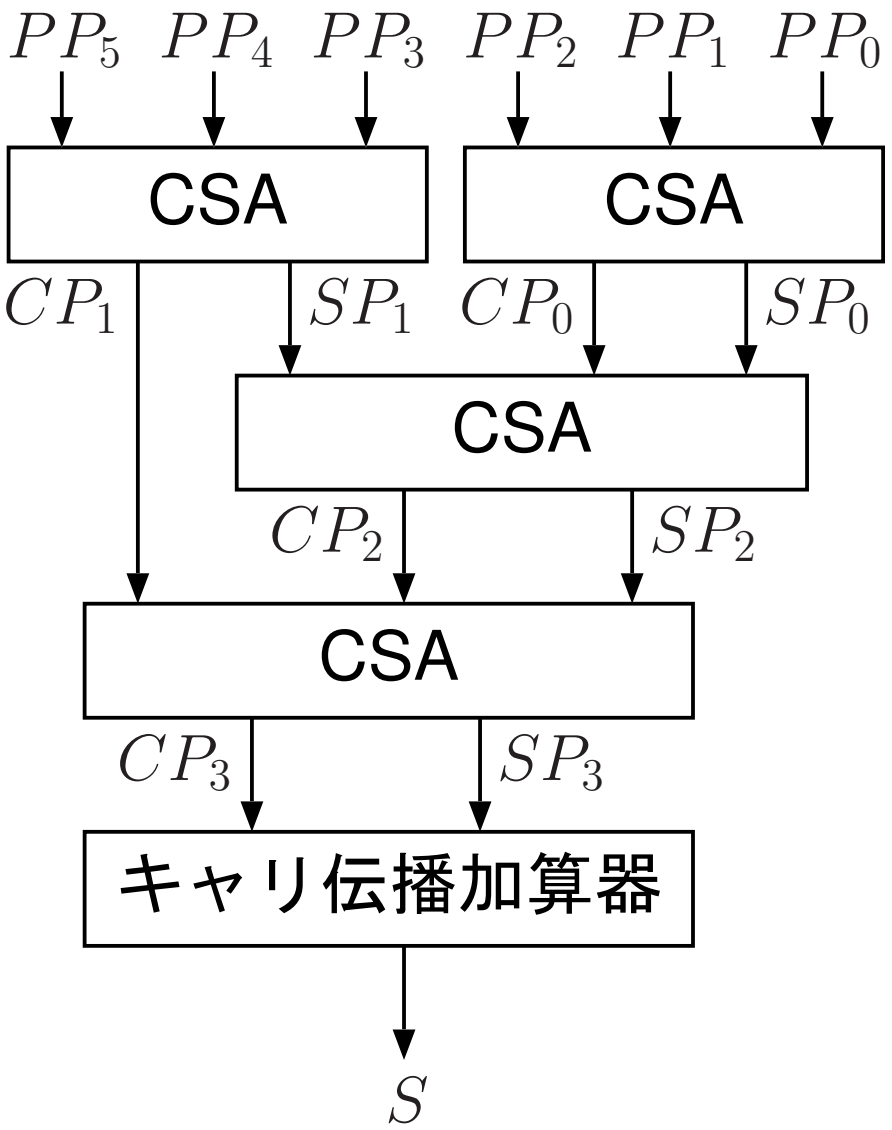
- 繰り返し乗算器を展開して，一気に行なう．
- 高速だが，面積が莫大になるため，あまり使われない．

ウォリス木

- 部分積を３個ずつ，CSA で加算していく．



ウォリス木乗算器

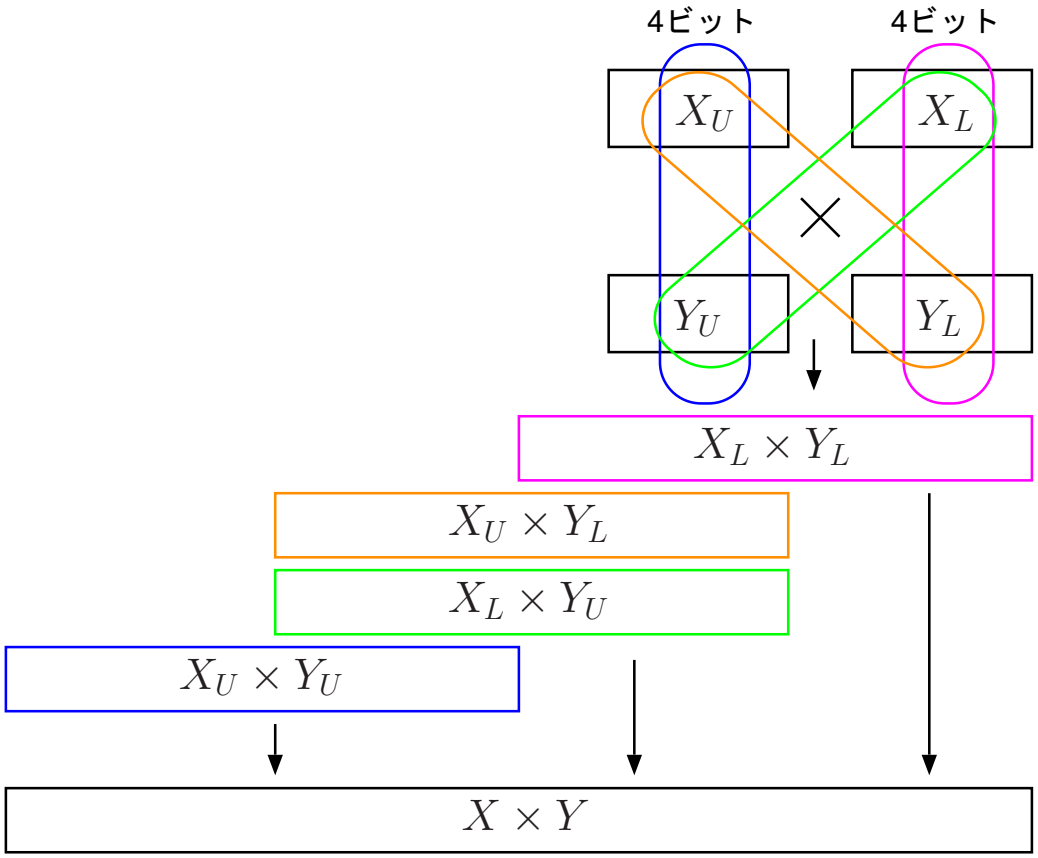


教科書P187 図6.27より

乗算の演算幅の拡張

- 多ビットの乗算をそれより少ないビットの乗算器を使って実行する.

$$
\begin{aligned}
1216 \times 2135 &= (1200 + 16) \times (2100 + 35) \\
&= 1200 \times 2100 + 1200 \times 35 + 16 \times 2100 + 16 \times 35 \\
X \times Y &= (X_U + X_L) \times (Y_U + Y_L)
\end{aligned}
$$



教科書P188 図6.28より

乗算機構のコンディション

- 乗数，被乗数のどちらかが0だったら，演算結果は0．(当たり前)
- どちらかが0だったら，直ちに演算結果を0にできる．
- 0判定は，全ビットのANDを取ることで可能．
- nビットの値同士の乗算は，$2n$ビットの結果を産む．
  - 32ビット計算機において，C言語のshort同士の乗算は，オーバーフローしない
  - int同士の乗算はオーバーフローする．

固定小数点の除算機構

- 小学校で，乗算の次に，除算を習うように，計算機内でも，乗算より除算の方が難しい．

  (被除数, dividend) ÷ (除数, divisor) ＝ (商, quotient) · · · (剰余, remainder)

  (被除数) ＝ (除数) × (商) ＋ (剰余)

- 基本的に，除算も，筆算を用いて行なう．

  − 2進数なので，九九の知識は不要．

$$
\begin{array}{rr}
 & 1\ 0\ 9 \\
\hline
14\,) & 1\ 5\ 3\ 0 \\
 & -1\ 4 \\
\cline{2-3}
 & \quad 1\ 3 \\
 & \quad -\ 0 \\
\cline{2-3}
 & \quad 1\ 3\ 0 \\
 & -\ 1\ 2\ 6 \\
\cline{2-3}
 & \qquad\quad 4
\end{array}
$$

固定小数点の除算

$$(98)_{10} \div (9)_{10} = (10)_{10} \cdots (8)_{10}$$

$$(01100010)_2 \div (1001)_2 = (1010)_2 \cdots (1000)_2$$

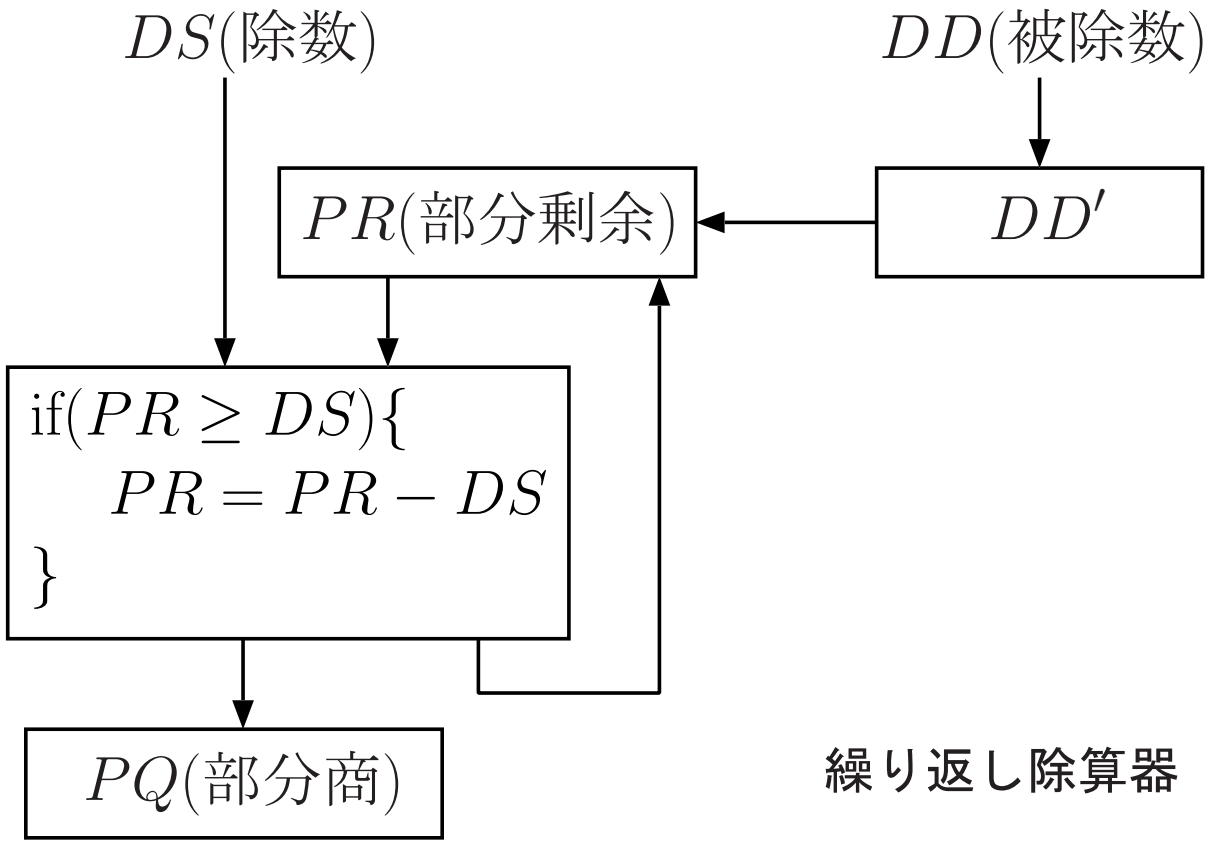
$$
\begin{array}{rcccccccc}
 & & & 0 & 1 & 0 & 1 & 0 & PQ_0, PQ_1, PQ_2, PQ_3, PQ_4 \\
\cline{4-8}
1001\,) & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
 & -0 & 0 & 0 & 0 & & & & \\
\cline{2-5}
 & & 1 & 1 & 0 & 0 & & & PR_0 \\
 & - & 1 & 0 & 0 & 1 & & & \\
\cline{3-6}
 & & & 0 & 1 & 1 & 0 & & PR_1 \\
 & & - & 0 & 0 & 0 & 0 & & \\
\cline{4-7}
 & & & & 1 & 1 & 0 & 1 & PR_2 \\
 & & & - & 1 & 0 & 0 & 1 & \\
\cline{5-8}
 & & & & & 1 & 0 & 0 & 0 \quad PR_3 \\
 & & & & - & 0 & 0 & 0 & 0 \\
\cline{6-9}
 & & & & & 1 & 0 & 0 & 0 \quad R
\end{array}
$$

$$(\text{被除数}) = (\text{除数}) \times (PQ_0, PQ_1, PQ_2, PQ_3, PQ_4) + R$$

固定小数点の基本除算機構

$$DD(\text{被除数}) \div DS(\text{除数}) = Q(\text{商}) \cdots R(\text{剰余})$$

$$DD = Q \times DS + R \qquad (R < DS)$$



繰り返し除算器

繰り返し除算器の問題点

```
if(PR ≥ DS){
        PR = PR − DS
}
```

- $PR \geq DS$は, PRとDSの減算を行なっている.
- $PR = PR - DS$も, PRとDSの減算を行なっている.
- 減算を2回も行なうのは無駄.
- 最初の比較で, 減算してしまえ! ↑ 引き戻し法, 引き放し法

引き戻し法

$$
\begin{aligned}
&PR = PR - DS \\
&\mathsf{if}(PR < 0)\{ \\
&\qquad PR = PR + DS \\
&\}
\end{aligned}
$$

- 減算して，負だったら，元に戻す．
- 先ほどの方法と，根本的に加減算の回数は変わらない．
- 下記のように行なう．

$$
\begin{aligned}
&PR' = PR \\
&PR = PR - DS \\
&\mathsf{if}(PR < 0)\{ \\
&\qquad PR = PR' \\
&\}
\end{aligned}
$$

複数ビットの部分商の同時生成

- $\div 2$ではなく，$\div n$を行ない，繰り返し数を減らす．
- 2ビット同時行なう場合は，除数より2ビット多いPRと，$\frac{3DS}{2}$，DS，$\frac{DS}{2}$の比較を同時行なうことで，商を求める．
  - 教科書の，$\frac{PR}{2}$は正しくないと思われる．
- $\frac{3DS}{2}$，$\frac{DS}{2}$は簡単に求められる．

$$\frac{3DS}{2} = DS + DS >> 1$$
$$\frac{DS}{2} = DS >> 1$$

複数ビットの同時生成

- 最初だけ，除数のビット幅＋1より始める．
- その後は2ビットずつずらして計算する．

$$(47)_{10} \div (6)_{10} = (7)_{10} \cdots (5)_{10}$$

$$(101111)_2 \div (110)_2 = (111)_2 \cdots (101)_2$$

$$
\begin{array}{rcccccc}
 & & & 0 & 1 & 1 & 1 \\
\cline{4-7}
110) & 0 & 1 & 0 & 1 & 1 & 1 \\
 & -1 & 0 & 0 & 1 & .0 & \\
 & -0 & 1 & 1 & 0 & .0 & \\
 & -0 & 0 & 1 & 1 & .0 & \\
\cline{2-6}
 & & 0 & 1 & 0 & 1 & \\
 & & 0 & 1 & 0 & 1 & 1 & 1 \\
 & & - & 1 & 0 & 0 & 1 & .0 \\
 & & - & 0 & 1 & 1 & 0 & .0 \\
 & & - & 0 & 0 & 1 & 1 & .0 \\
\cline{3-8}
 & & & & 0 & 1 & 0 & 1
\end{array}
$$

$$
\begin{array}{l}
\\
\\
\\
\\
\frac{3DS}{2} = 9 \\
DS = 6 \\
\frac{DS}{2} = 3 \\
PR - DS \\
\text{2ビット分付け足す} \\
\frac{3DS}{2} \\
DS \\
\frac{DS}{2} \\
PR - \frac{3DS}{2}
\end{array}
$$

その他の除算法

引き放し法 回復せずに，次回の減算時に足す．

乗算収束型除算法 乗算を用いて，商を近似する．

配列型除算器 配列型乗算器の除算器版

オーバフローとアンダーフロー

オーバーフロー 演算結果が最大値を上回る (over) こと．

アンダーフロー 演算結果が最小値を下回る (under) こと．

除算機構のコンディション

- 除算は, その他の演算と異なり, 0で割ることができない.
- 0で割ると, 割り込みがかかり, プログラムが終了する.

```
#include <stdio.h>
main()
{
  register int a,b,c;
  a=1;
  b=0;
  c=1/b;
  printf("%d\n",c);
}
```

実行結果

```
Floating exception（英語環境の場合）
```
演算例外（日本語環境の場合）

浮動小数点の算術演算装置

- 浮動小数点演算は, 加減算がもっとも難しい.
  - 桁あわせをしなければならないので.

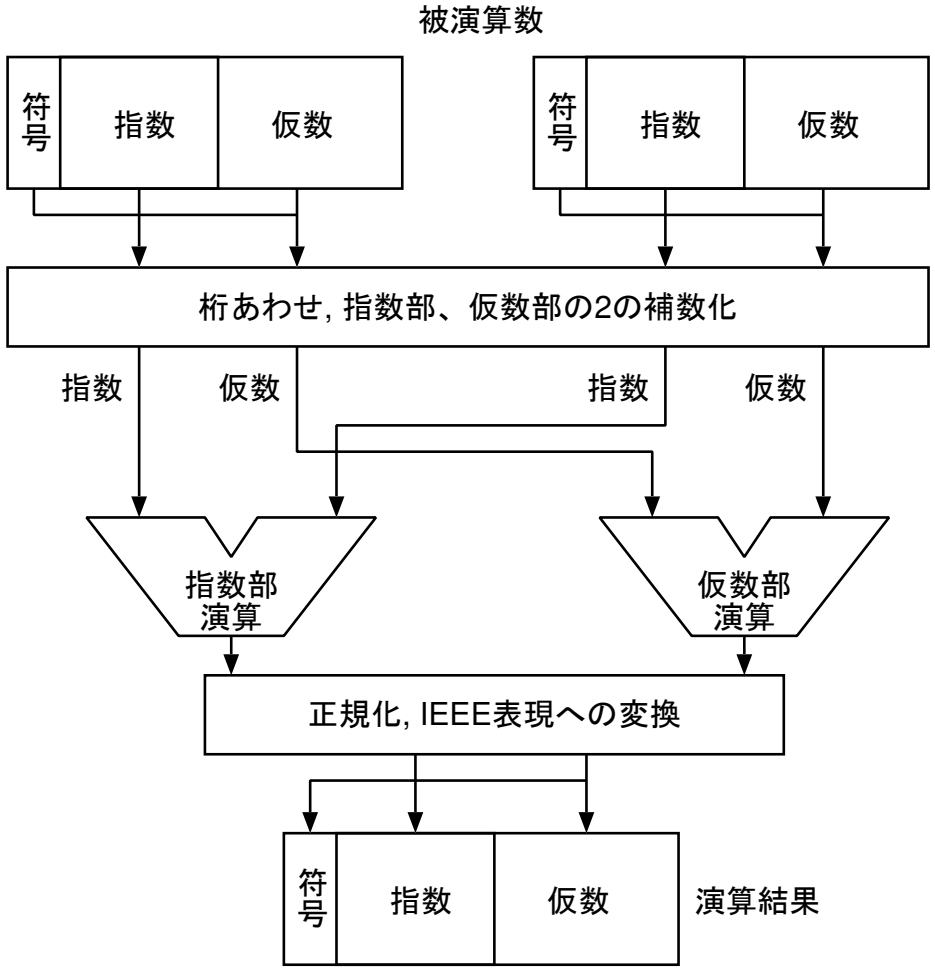
$$(7.25)_{10} + (1.125)_{10} = (8.375)_{10}$$

$$(1.1101 \times 2^2)_2 + (1.001 \times 2^0)_2 = (1000.011)_2$$

$$\begin{array}{rrl} & 1.1101 \times 2^2 & \text{正規化} \\ + & 1.001 \times 2^0 & \\ \hline & 1.11010 \times 2^2 & \\ + & 0.01001 \times 2^2 & \text{桁合わせ} \\ \hline & 10.00011 \times 2^2 & \\ & 1000.011 \times 2^0 & \\ & (8.375)_{10} & \end{array}$$

浮動小数点演算

- 乗除算は，桁あわせの必要はないので，加減算より簡単.



教科書P204 図6.42より

加減算時
- 被演算数の桁あわせを行なう.
- 指数演算は何もなし.
- 仮数は加減算を行なう.
- 演算終了後正規化

乗除算時
- 被演算数の桁あわせは不要.
- 指数は，加減算を，仮数は乗除算を行なう.
- 演算終了後正規化.

演算結果の丸め

- 浮動小数点演算は，「丸め (rounding)」による誤差がでる．
- INTEL の x86 では，浮動小数点レジスタは 80 ビット．IEEE 形式にする際に，丸める．

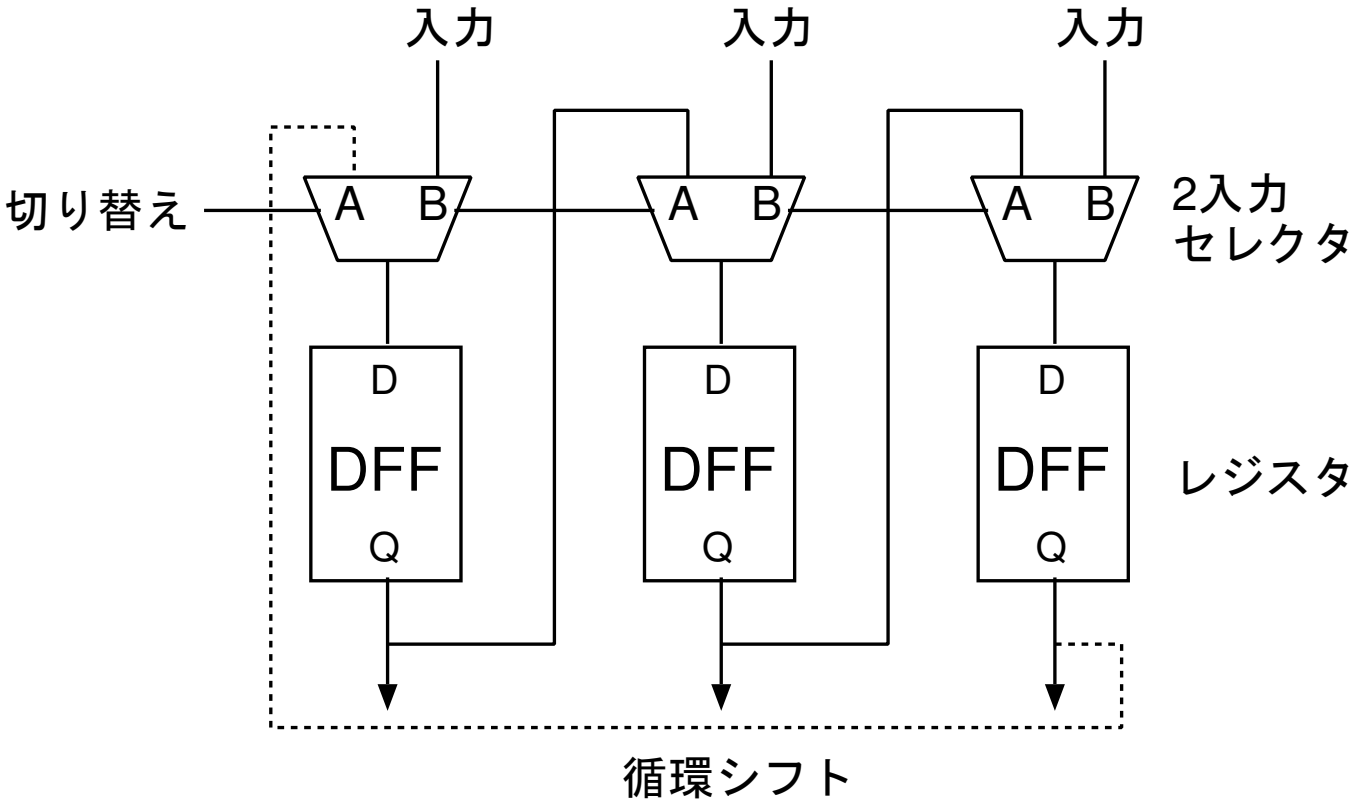
切り捨て 絶対値の小さい方に近似．$1.5 \simeq 1.0$

切り上げ 絶対値の大きい方に近似．$1.5 \simeq 2.0$

R 丸め m と M の中央値 C 未満は，切り捨て，C 以上は切り上げ． 四捨五入 (round off)．

シフタ

- ２進数では，１ビット右シフト(>> 1)は，÷2と同じ．
- ２進数では，１ビット左シフト(<< 2)は，×2と同じ．



３ビットシフトレジスタ

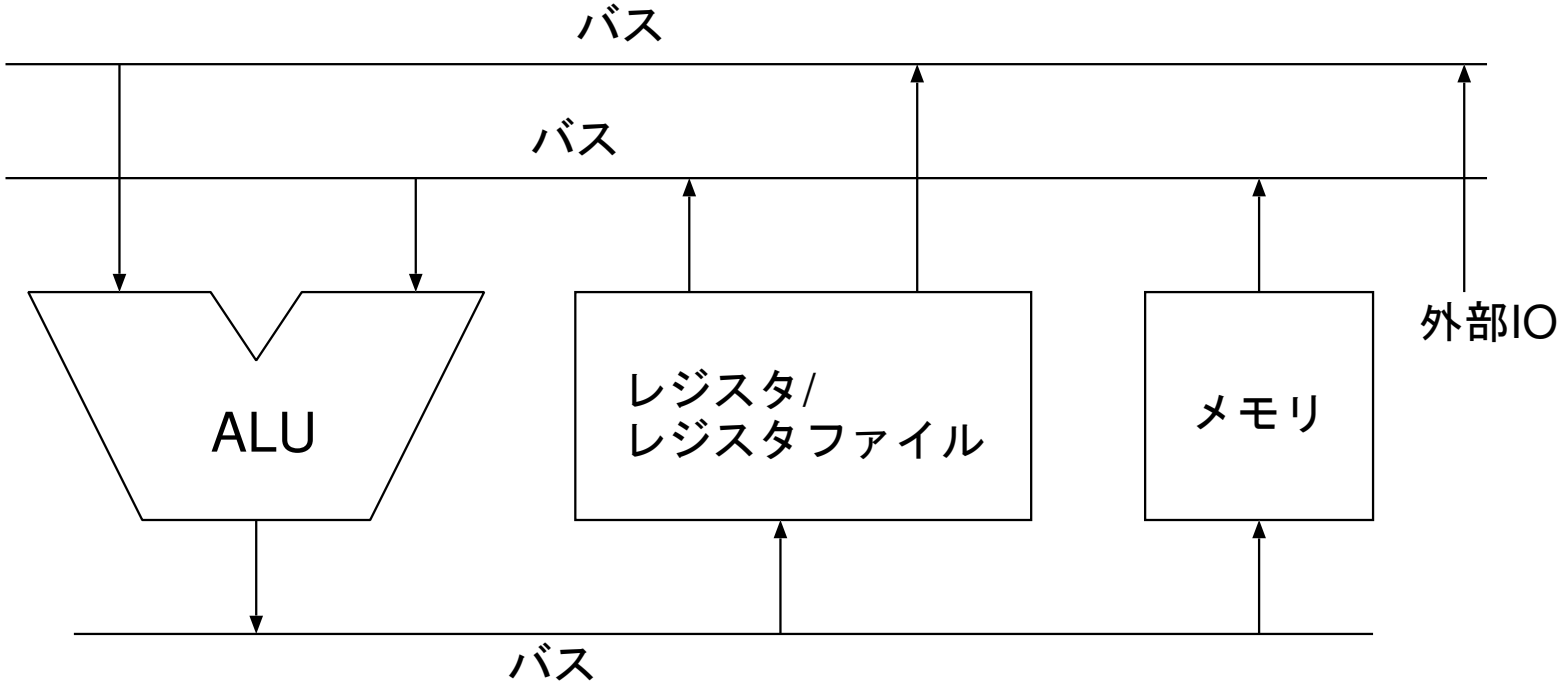
逐次シフタとバレルシフタ

逐次シフタ　１ビットごとにシフトするシフタ

バレルシフタ (barrel shifter) 任意のビット数シフトできるシフタ.

- バレルシフタは, ハードウエアが巨大になるので, めったに使われない.
- プロセッサの命令は, ほとんど1ビットシフトのみ

ALUアーキテクチャ

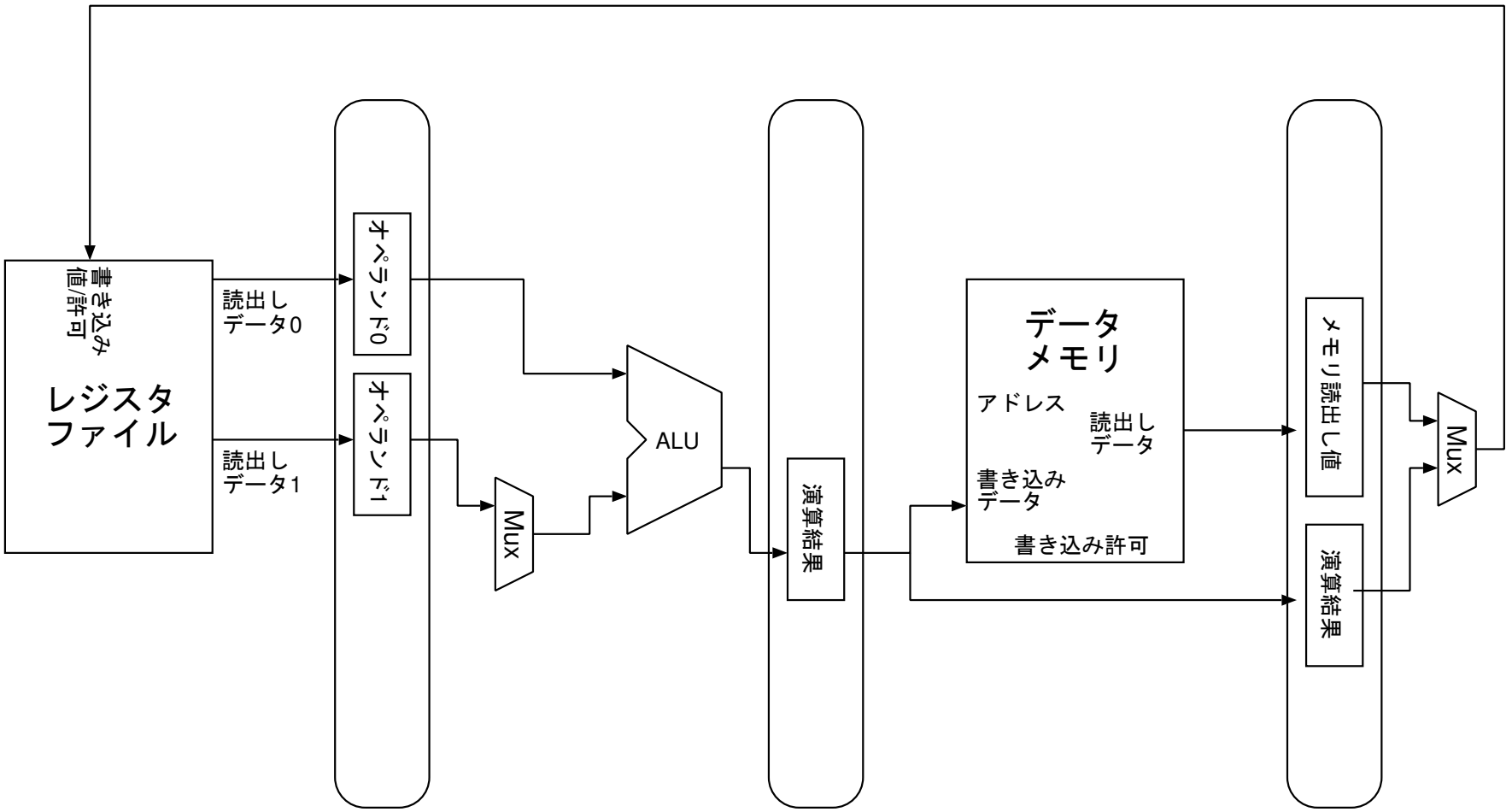


CISCプロセッサの構造

教科書P213 図6.48より

バス 多数の入出力が接続された配線．(単に多ビットの信号線のことをバスと言うこともある．)

ALUアーキテクチャ



RISCプロセッサの構造

(ALU，レジスタファイル，パイプラインレジスタのみ抜き出す.)

複合演算

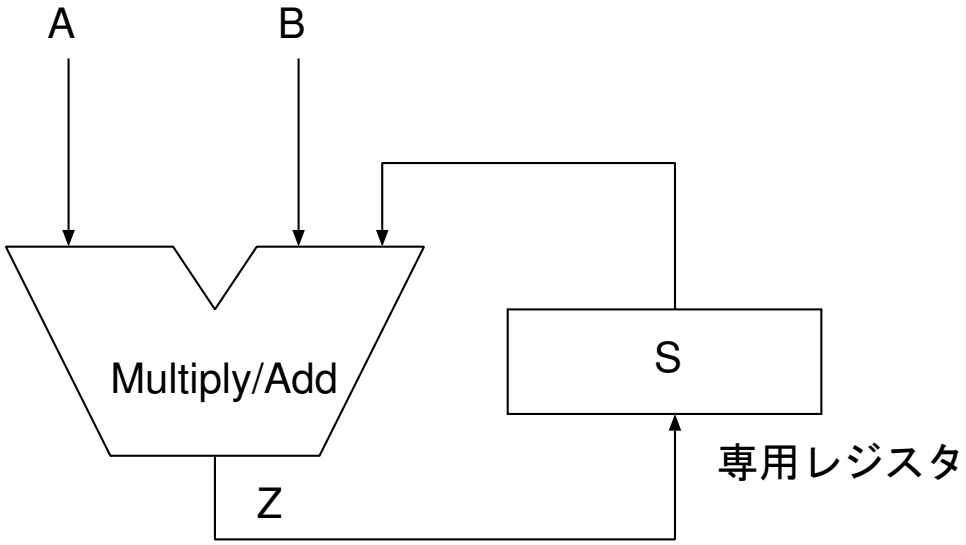
- 信号処理などでは，下記の積和演算が多用される．FIRフィルターなど．

$$S \;=\; S + A \times B$$

- 通常のALUで実装すると，遅い．

```
multiply A,B,Z
add S,Z,S
```

- 専用の演算器を設けて，高速化する．

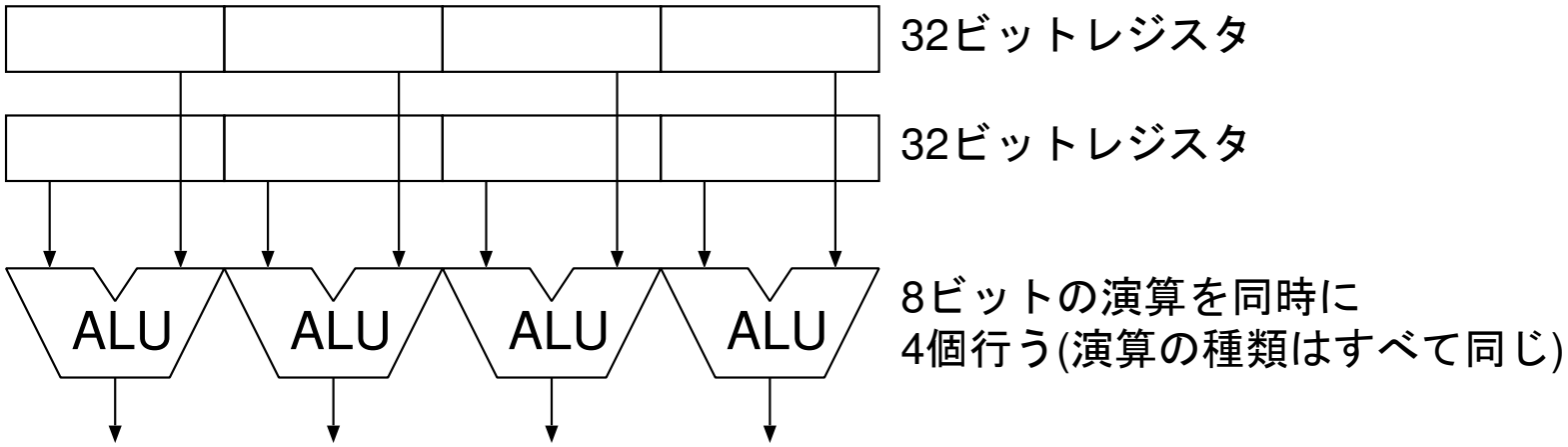


並列演算

命令並列 異なる命令を同時に並列に行う．VLIW, スーパスカラ．MIMD (Multiple Instruction stream Multiple Data stream)

データ並列 同じ命令を異なるデータに対して同時に並列に行う．SIMD (Single Instruction stream Multiple Data stream)

MMX, SSE, 3Dnow! SIMD型の演算をプロセッサ内部で行う．



教科書P215 図6.50より
MMX命令の概念図．