はじめに

- 今回は乗算アルゴリズムと乗算器（固定小数点乗算回路）について講義します。

- 教科書の対応範囲は、以下の通りです。
  - 6.1.3項（p.180～189）

2進数の乗算

- 基本的には、我々が筆算で計算するのと同じです。
- 乗算器を設計するのに、足し算部分をどのように処理するかで、いろいろな工夫がなされます。

例）1101×1011（13×11）

```
           1101
 ×)        1011
 ---------------
           1101
          1101
         0000
        1101
 ---------------
       10001111
```

- 皆さんは、小学生のときにかけ算の九九を暗記させられたはずです。
  - 合計81(=9×9)個のパターンを憶えなければならないので、苦労した人もいるかもしれません。
- しかし、2進数の場合の九九(?)は極めてシンプルです。
- 論理演算のAND(論理積)で実現できます。

x \ y	0	1
0	0	0
1	0	1

$$z = x \times y = x \cdot y$$

算術積　論理積

- よって、部分積は、以下のように各桁の論理積（AND）で求まります。

```
      1101
×)    1011
----------
      1101
     1101
    0000
   1101
----------
  10001111
```

				x_3	x_2	x_1	x_0
×）				y_3	y_2	y_1	y_0
				$x_3 \cdot y_0$	$x_2 \cdot y_0$	$x_1 \cdot y_0$	$x_0 \cdot y_0$
			$x_3 \cdot y_1$	$x_2 \cdot y_1$	$x_1 \cdot y_1$	$x_0 \cdot y_1$	
		$x_3 \cdot y_2$	$x_2 \cdot y_2$	$x_1 \cdot y_2$	$x_0 \cdot y_2$		
	$x_3 \cdot y_3$	$x_2 \cdot y_3$	$x_1 \cdot y_3$	$x_0 \cdot y_3$			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

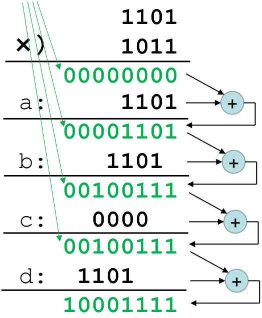
繰り返し型乗算器

- ALU(加算器)を1個用いて、加算を繰り返すことで乗算を行います。

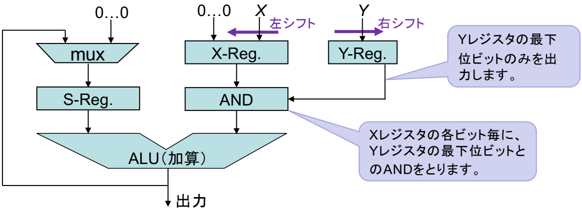
```
        1101
×)      1011
------------
a:      1101
b:     1101
c:    0000
d:   1101
------------
    10001111
```

一度にa:〜d:を全部足すのではなくて、1個ずつ、部分和に加えていきます。

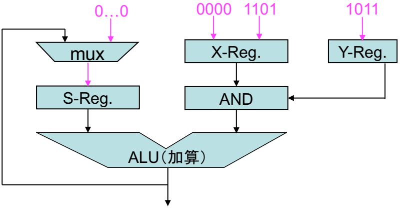
緑色で示したのが部分和です。



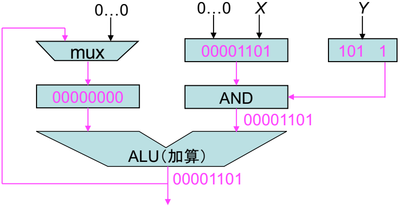
- 繰り返し型乗算器のハードウェア構成です。
  - Xレジスタ(シフトレジスタを使用)には被乗数をセットします。このとき、上位に0のビットを追加します。また、サイクル毎に、左へ1ビットシフトします。
  - Yレジスタ(シフトレジスタを使用)には乗数をセットします。サイクル毎に、右へ1ビットシフトします。
  - Sレジスタには、部分和を格納します。最初に0に初期化します。



- 動作例
  - サイクル1
    - 被乗数、乗数を入力して、X、Yレジスタにセットします。

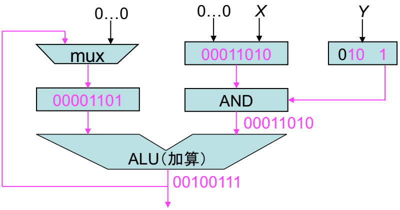


  - サイクル2
    - 部分積の加算を行い、X、Yレジスタをシフトします。



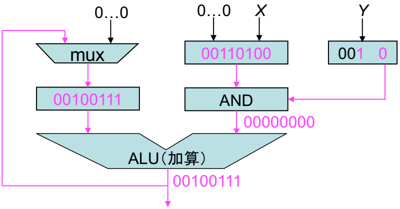
– サイクル3

- 部分積の加算を行い、X、Yレジスタをシフトします。

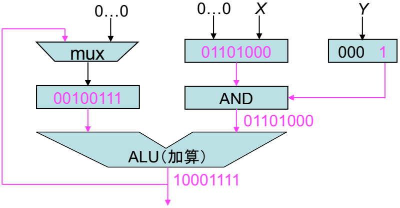


– サイクル4

- 部分積の加算を行い、X、Yレジスタをシフトします。



– サイクル5
  - 部分積の加算を行います。
  - 加算結果を乗算結果として出力します。



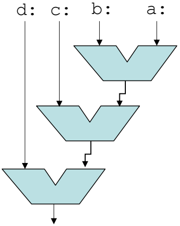
- nビットの繰り返し型乗算器では、n+1サイクルで乗算結果を出力します。

  – 次ページ以降で、もっと高速な乗算器を考えます。

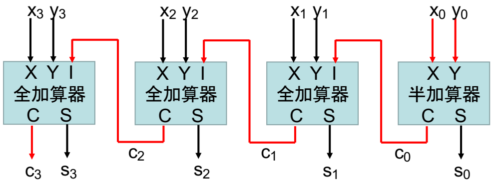
配列型乗算器

- ALU(加算器)を複数個使用することができるなら、そのまんまですが(何の工夫もありませんが)、右下図のような構成の並列乗算器が考えられます。

```
           1101
×)         1011
---------------
a:         1101
b:        1101
c:       0000
d:      1101
---------------
       10001111
```



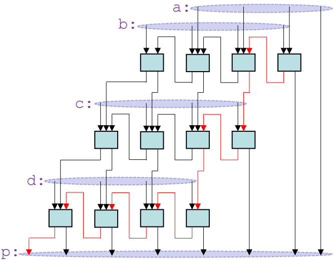
- ところで、加算器の性能面での問題は、桁上げが最下位ビットから最上位ビットまで伝播するのに時間がかかることでした。
  - ビット数が多いほど、加算結果が出るまでの時間もかかります。



  - この点を改善できると、乗算器の性能を向上させることができます。

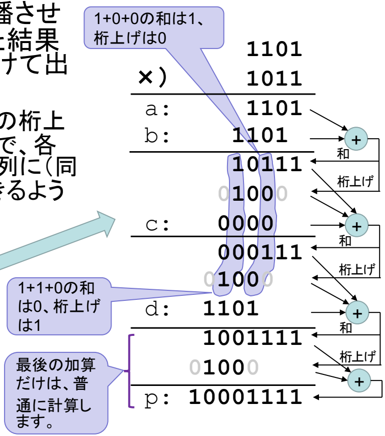
- 先の並列乗算器を、全(半)加算器のレベルでブロック図を描くと以下のようになります。
  - クリティカルパス(赤線で示しています)は、全(半)加算器が8段分となっています。

	1101
×)	1011
a:	1101
b:	1101
c:	0000
d:	1101
p:	10001111

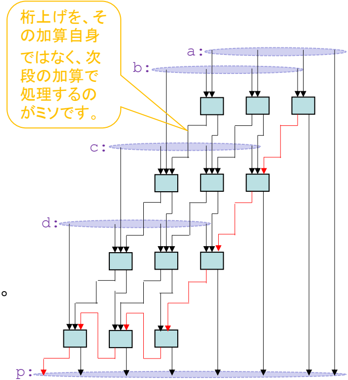


- ここで、桁上げを伝播させないように、加算した結果を、和と桁上げに分けて出力してみましょう。
  - 各ビットで下位からの桁上げを入力しないことで、各ビットでの加算を並列に(同時に)行うことができるようにします。

	1101
×)	1011
a:	1101
b:	1101
c:	0000
d:	1101
p:	10001111

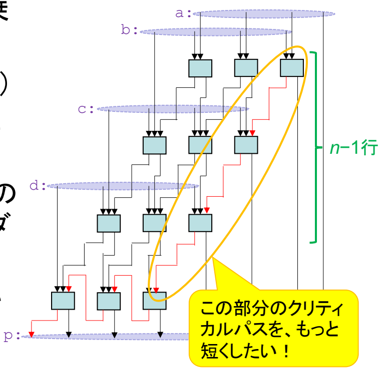


- この考え方で並列乗算器を設計すると、全(半)加算器のレベルのブロック図は右のようになります。
  - クリティカルパスは、全(半)加算器の6段分になります。
    - 最初に示した並列乗算器と比べると2段しか減っていませんが、ビット数が増えると、その差はもっと大きくなります。
- 全(半)加算器を2次元に整然と並べるので、配列型乗算器と呼ばれます。



Wallace tree

- *n*ビットの配列型乗算器では、最後の行を除くと、全（半）加算器を*n*−1行並べます。
  - つまり、この部分の遅延時間のオーダは、*O*(*n*)です。
- もっと速くできないでしょうか？



- 6ビットの乗算器を例に考えます。
  - 並列乗算器の場合と同様、和と桁上げを分けて出力する加算器を使用します。

```
          101110
×)        110101
----------------
a:        101110
b:       000000
c:      101110
d:     000000
e:    101110
f:   101110
----------------
p: 100110000110
```

3行ずつ、2つに分けて加算を行います。

```
a:        101110
b:       000000
c:      101110
-------------------------
g:      10010110   和
h:      010100     桁上げ
```

```
d:     000000---
e:    101110
f:   101110
-------------------------
k:   11100100---   和
m:   001100        桁上げ
```

これら2つの加算は、同時に(独立に)計算できます。

```
a:       101110
b:      000000
c:     101110
g:     10010110
h:     010100
```

a:～f:の6行が、4行に減りました。

```
g:     10010110
h:     010100
k:  11100100
m:  001100
```

次は、この3行を加算します。

```
d:     000000---
e:    101110
f:   101110
k:   11100100---
m:   001100
```

```
g:       10010110
h:       010100
k:   11100100
n:   11111100110   和
q:      000100     桁上げ
```

さらに、m:、n:、q:の3行を加算します。

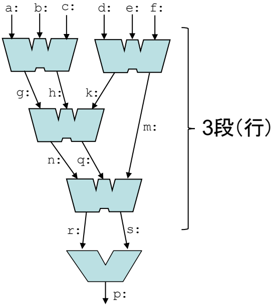
次ページへ

– よって、乗算器の構成は以下のようになります。

```
m:   001100
n:   11111100110
q:       000100
r:   11001000110  和
s:  00110100      桁上げ
```

最後の2行は普通に加算します。

```
r:   11001000110
s:  00110100
p:  100110000110
```



- 最後の加算器を除くと、3段（個）分の加算器の遅延時間で計算できます。配列型乗算器の場合は、この部分が5段になりますので、Wallace treeを用いたほうが、より短い時間で計算できることがわかります。

- *n*ビットの配列型乗算器では、最後の加算器を除いた部分の遅延時間のオーダがO(*n*)であるのに対し、Wallace treeを用いた場合のオーダはO(log *n*)となります。
  - 32ビットの整数乗算器の場合、最後の加算器を除くと、配列型乗算器では31段なのに対して、Wallace treeでは8段となります。
  - また、倍精度の浮動小数点乗算器内部の乗算ブロック（53桁として計算）では、配列型の52段に対して、Wallace treeでは9段です。

- ビット数が多いと、圧倒的に加算器の段数が少なくできるWallace treeですが、欠点もあります。
  - 配列型乗算器の場合は、LSI上で全(半)加算器を規則正しく並べることができ、短い配線で全(半)加算器間を接続できます。
  - しかし、Wallace treeの場合は、全(半)加算器の配置や配線パターンが不規則になります。
  - 特に、現在のLSIでは、微細化が進んだために配線遅延が性能上の問題として顕在化しているため、配線長が長くなるWallace treeにとっては不利です。
  - したがって、常にWallace treeの方が高速であるとは限りません。
  - 配列型とするかWallace treeを用いるかは、加算器の段数と配線遅延とを考慮して決定する必要があります。

Boothの乗算アルゴリズム

- これまでは、暗黙のうちに、整数を0または正の数として（つまり、符号なしの整数として）考えてきました。
  - 浮動小数点数の乗算の場合は、それで何の問題もありません。
  - しかし、固定小数点数の乗算の場合は、2の補数で表現した負の数についても考えておかなければなりません。
    - 負の数を正の数に変換してから乗算して、乗算結果をまた負の数に戻す、というのでも計算できなくはありませんが、正負の判定と反転に余分な時間がかかってしまいます。
- ところが、Boothという人（コンピュータの専門家ではなく、化学が専門でした）が、正か負かを判定することもなく、負の数であってもそのまま乗算できる方法を発明しました。

- まず、負の数の表現と、その実際の値との関係について確認しておきます。
  - nビットの符号付き整数を考えます。
  - A(≧0)に対して、$-A$の表現Bは $B=2^n-A$ です。
  - これより、$-A=B-2^n$
    - つまり、負の数の場合は、その表現から2^n を引いた値が実際の値である、ということです。
      - 例えば、$n=3$のとき、-2の表現は$110_{(2)}=6$ です。この6から$2^3=8$を引くと、実際の値$6-8=-2$となります。
- では、$X\times Y$ について考えます。
  - Y の各ビットをy_iとして、$(y_{n-1}\,y_{n-2}\,\ldots y_1\,y_0)_2$と表されているとします。

- $Y \geqq 0$のとき、y_{n-1}=0で、

$$Y = \sum_{i=0}^{n-2} y_i \cdot 2^i$$

- $Y < 0$のとき、y_{n-1}=1で、

$$\begin{aligned} Y &= \sum_{i=0}^{n-1} y_i \cdot 2^i - 2^n \\ &= -2^n + \underbrace{y_{n-1}}_{=1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} y_i \cdot 2^i \\ &= -2^{n-1} + \sum_{i=0}^{n-2} y_i \cdot 2^i \end{aligned}$$

- y_{n-1}を利用して1つにまとめると、

$$Y = -y_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} y_i \cdot 2^i$$

- これを以下のように変形します。

$$\begin{aligned} Y &= -y_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} y_i \cdot (2^{i+1} - 2^i) \\ &= -y_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} y_i \cdot 2^{i+1} - \sum_{i=0}^{n-2} y_i \cdot 2^i \\ &= \sum_{i=1}^{n-1} y_{i-1} \cdot 2^i - \sum_{i=0}^{n-1} y_i \cdot 2^i \end{aligned}$$

– ここで、$y_{-1}\cdot 2^0$（ただし、$y_{-1}=0$とおく）をYに加えます。

$$Y = \sum_{i=1}^{n-1} y_{i-1}\cdot 2^i - \sum_{i=0}^{n-1} y_i \cdot 2^i + y_{-1}\cdot 2^0$$

$$= \sum_{i=0}^{n-1} y_{i-1}\cdot 2^i - \sum_{i=0}^{n-1} y_i \cdot 2^i$$

$$= \sum_{i=0}^{n-1} (y_{i-1} - y_i)\cdot 2^i$$

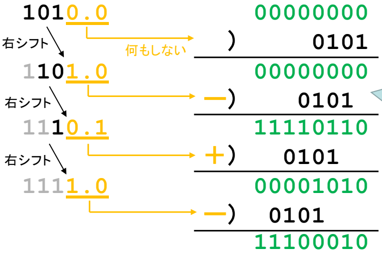
- よって、

$$X \times Y = X \times \sum_{i=0}^{n-1} (y_{i-1} - y_i) \cdot 2^i$$

$$= \sum_{i=0}^{n-1} (y_{i-1} - y_i) \cdot X \cdot 2^i$$

- これより、$Y=(\ldots\ y_i y_{i-1}\ \ldots)_2$の隣り合うビットを調べて、以下のように操作すればよいことがわかります（ただし、$y_{-1}=0$です）。
  - $y_i y_{i-1}$=01なら、部分和に$X \cdot 2^i$を足す。
  - $y_i y_{i-1}$=10なら、部分和から$X \cdot 2^i$を引く。
  - $y_i y_{i-1}$=00 or 11なら、何もしない。

- では、実際に計算してみましょう。
  - n=4ビットで、5×(−6)を計算します。
    - $5_{(10)}=0101_{(2)}$、$-6_{(10)}=1010_{(2)}$
    - 積は8ビットに拡張します。$-30_{(10)}=11100010_{(2)}$



この例では、被乗数が正ですが、負の場合は符号拡張に注意してください。

– n=4ビットで、(−6)×5 を計算します。

- $-6_{(10)}=1010_{(2)}$、$5_{(10)}=0101_{(2)}$
- $-30_{(10)}=11100010_{(2)}$
- −6 の符号拡張に注意してください

0101.0		00000000
右シフト	−)	11111010
0010.1		00000110
右シフト	+)	1111010
0001.0		11111010
右シフト	−)	111010
0000.1		00010010
	+)	11010
		11100010

- 前ページの計算例では、繰り返し型の乗算器で処理する場合を想定して計算しました。

- 引き算は補数をとって足すことで実現しますので、先に説明した配列型乗算器を改良して、Boothの方法を実装する並列乗算器を設計することができます。