

# 計算機構成論

## Lecture 8

# 計算機での算術演算

2023年度前期

情報理工学部 Rクラス担当

越智裕之

※ このレジュメの中で「教科書」とは、一昨年度まで教科書に指定されていた書籍のことです（Lecture 0 参照）。この書籍を入手できなくても支障なく学習できるよう、レジュメに加筆修正を行っています。

- **2進数の加算と減算** (教科書3.2節, 付録(B.2, B.3, B.5)の一部 + $\alpha$ )
  - **実行の様子** ポイント: 最上位からの桁上げは無視してOK
  - オーバフロー (オーバーフローというのが一般的かもしれないが教科書に合わせます)
  - ハードウェアの概要 (加算器 $\Rightarrow$ ALU)
- **乗算の実現方法** (教科書3.3節)
  - 基本的な乗算アルゴリズムとそのハードウェア
  - その改良
- **除算の実現方法** (教科書3.4節)
  - 基本的な除算アルゴリズムとそのハードウェア
- **浮動小数点形式による実数の演算** (教科書3.5節)
  - 仮数、指数、正規化
  - IEEE754浮動小数点規格
  - 演算精度と丸め
  - 浮動小数点演算の実現方法

2022年度は  
試験範囲外

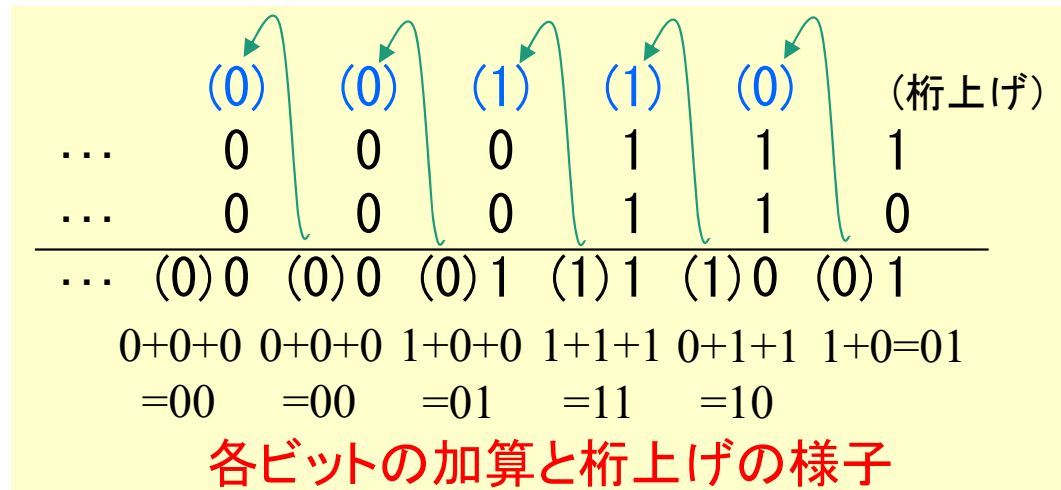
# 加算の実行

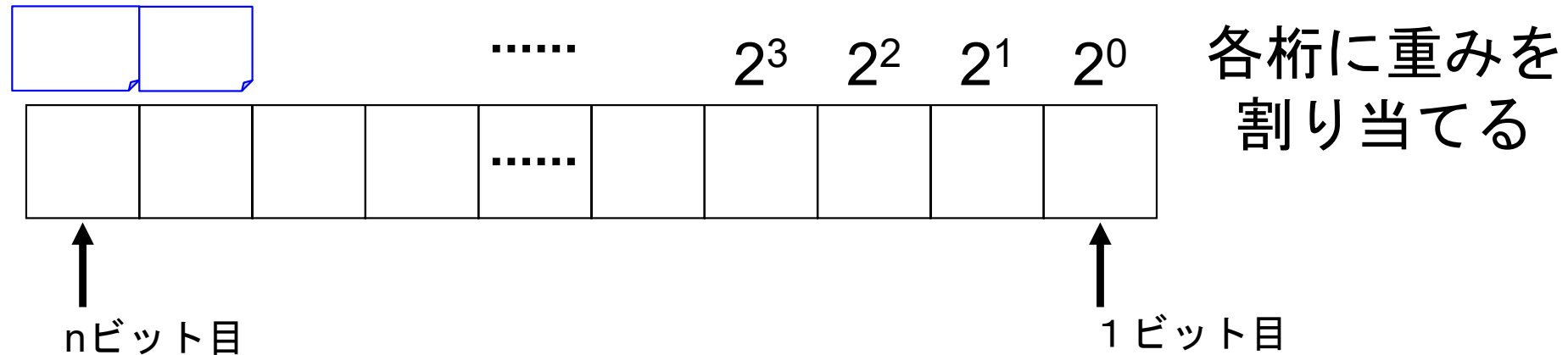
## 2進数の加算

例)  $6_{10}$ に $7_{10}$ を2進数で加える. **10進の加算と同じように桁上げをする.**

$$\begin{array}{r} 0000\ 0111 = 7_{10} \\ +\ 0000\ 0110 = 6_{10} \\ \hline =\ 0000\ 1101 = 13_{10} \end{array}$$

数値の各ビットを下位ビットから加え、桁上がりを隣の上位ビットに加える





## 復習ミニクイズ：2の補数の定義は？

### 定義

nビットのXの2の補数がY ⇔

### 求め方

各ビット反転して、 と2の補数となる

# 減算の実行：2の補数を足すだけでOK

## 2進数の減算 引く数を2の補数表現にしてから加算

例)  $7_{10}$  から  $6_{10}$  を2進数で引く

$$7_{10} - 6_{10} = 7_{10} + (-6_{10})$$

$$\begin{array}{r} 0000\ 0111 = 7_{10} \\ +\ 1111\ 1010 = -6_{10} \\ \hline =\ 10000\ 0001 = 1_{10} \end{array}$$

ポイント

2の補数表現を用いることで、  
非負数と負数の加算が特別な  
演算をすることなく、実現できる

他の符号付き整数表現では、特別な演算が必要

最上位か  
ら出てい  
く桁上げ  
は無視

	(1)	(1)	(1)	(1)	(0)	(桁上げ)
...	0	0	0	1	1	1
...	1	1	1	0	1	0
...	(1)0	(1)0	(1)0	(1)0	(1)0	(0)1
	1+0+1	1+0+1	1+0+1	1+1+0	0+1+1	1+0=01
	=10	=10	=10	=10	=10	

各ビットの加算と桁上げの様子

ポイント：最上位からの桁上げは無視しても  
(オーバフローが起こらない時は) OK

# 自分で確認する問題1/2

2の補数表現（8ビット）で、 $A+B=C$ をするとき、 $A, B$ の正負にかかわらず、単に足すだけで $C$ が正しい値となることを確認せよ。具体的には、 $A$ と $B$ の符号ビットが、①ともに0、②ともに1、③1と0、の3つの場合について以下の例で確認せよ。  
\*ただし、（どんなやり方をしてもそもそも）結果が正しくない値の時もある。それはどのような場合か？ 以下の例を自分で確認せよ。

## ①正+正

$$\begin{array}{r} \text{0} \\ \curvearrowright \\ \begin{array}{r} 00111111 \quad +63 \\ +) 00000011 \quad +3 \\ \hline \textcircled{0}01000010 \quad +66 \end{array} \end{array}$$

最上位から出ていく桁上げは無視

$$\begin{array}{r} \text{0} \\ \curvearrowright \\ \begin{array}{r} 00111111 \quad +63 \\ +) 01000011 \quad +67 \\ \hline \textcircled{0}10000010 \quad +130 \text{ とならない} \end{array} \end{array}$$

最上位から出ていく桁上げは無視

## 自分で確認する問題2/2

### ②負+負

$$\begin{array}{r} \overset{1}{\curvearrowright} \\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \quad -1 \\ +) \ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \quad -125 \\ \hline \textcircled{1}\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \quad -126 \end{array}$$

$$\begin{array}{r} \overset{1}{\curvearrowright} \\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1 \quad -65 \\ +) \ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \quad -125 \\ \hline \textcircled{1}\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0 \quad -190 \text{ とならない} \end{array}$$

最上位から出ていく桁上げは無視

### ③負+正

符号ビット

$$\begin{array}{r} \overset{1}{\curvearrowright} \\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \quad -1 \\ +) \ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \quad +3 \\ \hline \textcircled{1}\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \quad +2 \end{array}$$

符号ビット

$$\begin{array}{r} \overset{0}{\curvearrowright} \\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1 \quad -65 \\ +) \ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \quad +3 \\ \hline \textcircled{0}\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 0 \quad -62 \end{array}$$

最上位から出ていく桁上げは無視

①正+正, ②負+負では, 値がおかしくなることがある!  
(以下の, オーバフロー)

復習ミニクイズ：32ビットの2の補数表現で表現可能な数の最大値と最小値は？

最小	最大
<input type="text"/>	<input type="text"/>

上記の最小値より小さい負の数，最大値より大きい正の数は，そもそも表現できないので，加算の結果がそうなる場合は，仕方がない．（以下のオーバフロー）

逆に，そうならない場合は，最上位(符号)ビットからの桁上げを無視しても加算の結果は正しい（演習②で考えます）



- 2進数の加算と減算（教科書3.2節, 付録(B.2, B.3, B.5)の一部 + $\alpha$ )
  - 実行の様子 ポイント：最上位からの桁上げは無視してOK
  - オーバフロー（オーバーフローというのが一般的かもしれないが教科書に合わせます）
  - ハードウェアの概要（加算器⇒ALU）
- 乗算の実現方法（教科書3.3節）
  - 基本的な乗算アルゴリズムとそのハードウェア
  - その改良
- 除算の実現方法（教科書3.4節）
  - 基本的な除算アルゴリズムとそのハードウェア
- 浮動小数点形式による実数の演算（教科書3.5節）
  - 仮数、指数、正規化
  - IEEE754浮動小数点規格
  - 演算精度と丸め
  - 浮動小数点演算の実現方法

2022年度は  
試験範囲外

# 加減算におけるオーバーフロー

\* 教科書ではオーバーフローと書かれていますが、オーバーフローが普通のようにです

## オーバーフロー

計算結果の数値がハードウェアに用意されているビット数に収まりきらないこと

### 加算時のオーバーフローの例 (符号付き8bit整数の場合)

同じ符号の数を加算しているのに、  
違う符号の演算結果が得られている

符号付き8bit整数は、-128～127の整数  
しか表現できず、それ以外の範囲の演算  
結果の場合にはオーバーフロー

#### 正数 + 正数

$$\begin{array}{rcl} 01100001 & = & 97_{10} \\ + 01001000 & = & 80_{10} \\ \hline 10101001 & = & -87_{10} \end{array}$$

#### 負数 + 負数

$$\begin{array}{rcl} 10011111 & = & -97_{10} \\ + 10111000 & = & -80_{10} \\ \hline 01010111 & = & 87_{10} \end{array}$$

\* Cではオーバーフローは無視される

## 2 の補数表現時の加減算のオーバフロー判定

操作	オペランドA	オペランドB	結果
A+B	$A \geq 0$	$B \geq 0$	$A+B < 0$
A+B	$A < 0$	$B < 0$	$A+B \geq 0$
A-B	$A \geq 0$	$B < 0$	$A-B < 0$
A-B	$A < 0$	$B \geq 0$	$A-B \geq 0$

教科書図3.2

つまり, 3つの数字の符号ビットから判定可能(教科書の方法)

しかし, 実際にはもっと簡単に判定できる。  
それを次ページから考えてみましょう。

# 正か負の数の加算 (エラーがないとき)

負の数を2の補数表現にすると決めておけば、単に足し算するだけでOK

符号ビット

1 最上位に入ってくる桁上げ

$$\begin{array}{r} 11111111 \\ +) 00000011 \\ \hline (1)00000010 \end{array}$$

最上位から出ていく桁上げ

$$\begin{array}{r} 11111111 \\ +) 10000011 \\ \hline (1)10000010 \end{array}$$

1 最上位に入ってくる桁上げ

1 最上位から出ていく桁上げ

-1

+3

+2

-125

-126

符号ビット

0 最上位に入ってくる桁上げ

$$\begin{array}{r} 10111111 \\ +) 00000011 \\ \hline (0)11000010 \end{array}$$

最上位から出ていく桁上げ

0 最上位に入ってくる桁上げ

$$\begin{array}{r} 00111111 \\ +) 00000011 \\ \hline (0)01000010 \end{array}$$

0 最上位から出ていく桁上げ

-65

+3

-62

+63

+3

+66

が同じ場合はo.k.

# 正か負の数の加算 (オーバーフローの場合)

が違う場合は、オーバーフローとしてエラーを報告

符号ビット

最上位に入ってくる桁上げ

$$\begin{array}{r} 01111111 \quad +127 \\ +) 00000011 \quad +3 \\ \hline \textcircled{0}10000010 \quad -126 \end{array}$$

最上位から出ていく桁上げ

実際は +130のはず

符号ビット

最上位に入ってくる桁上げ

$$\begin{array}{r} 10111111 \quad -65 \\ +) 10000011 \quad -125 \\ \hline \textcircled{1}01000010 \quad +66 \end{array}$$

最上位から出ていく桁上げ

実際は -190のはず

8ビットで表現できるのは、-128 から +127 まで

# 演習問題 その①

自分で考えられるはず

答えを見ないで解いてみよう

次の演算を2の補数表現（8ビット）で計算せよ？オーバーフローが起こっているのはどちらか？それをハードウェア的にはどうやって検出すればよいか考えよ。  
教科書より効率のいい方法を考えよ

$$-67+35$$

符号ビット ☐ 最上位に入ってくる桁上げ

$$\begin{array}{r} 1 \mid 0111101 \quad -67 \\ +) \quad 0 \mid 0100011 \quad +35 \\ \hline \end{array}$$

$$63+127$$

☐ 最上位に入ってくる桁上げ

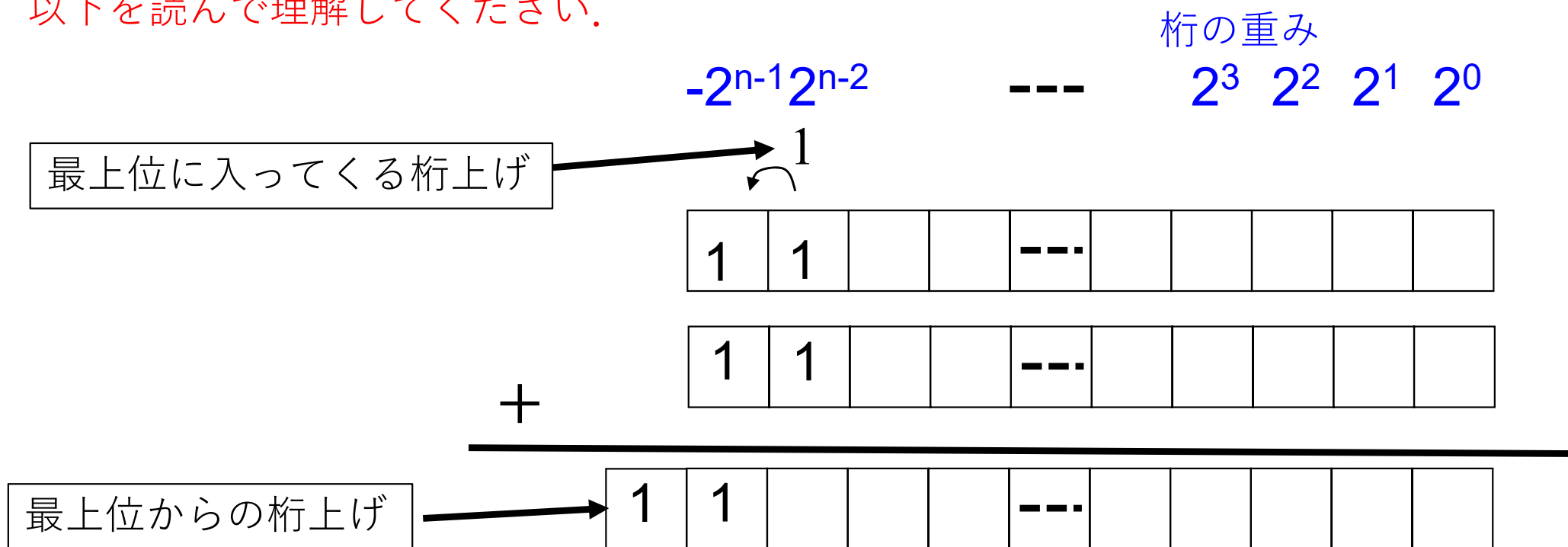
$$\begin{array}{r} 0 \mid 0111111 \quad +63 \\ +) \quad 0 \mid 1111111 \quad +127 \\ \hline \end{array}$$

## 演習問題その②

今までの話をまとめると、

「正＋負，負＋負の場合，最上位ビットからの桁上げが1の場合もあるが，それを無視しても，最上位ビットに入ってくる桁上げも1ならば計算結果は正しい。」ということである．その理由を説明せよ．

以下を読んで理解してください．



最上位に入ってくる桁上げは，本当は  $+2^{n-1}$  を意味する．しかし，上の足し算では，2の補数で考えているので，これを  $-2^{n-1}$  として扱っている．そのため，差し引き結果が  $-2^n$  となるはずである．

一方，最上位からの桁上げは， $-2^n$  のはずであるが，これを無視している．上記の2つのことがちょうど打ち消しあって値が正しいといえる．

## MIPSにおけるオーバーフロー対応

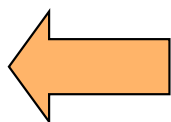
加算 (add)、即値加算 (addi)、減算 (sub)

→ オーバフロー時に例外が発生

符号なし加算 (addu)、符号なし即値加算 (addiu)、  
符号なし減算 (subu)

→ オーバフローが起こっても例外は発生しない

(足し算自体は同じようにするが、オーバーフローのチェックをするかどうかが違う)



オーバーフローを認識するかしないかは、状況に応じるため、  
それに対応して、命令を選択する。

例外(exception) ≡ 割込み(interrupt) (プロセサ外の原因を特にいう)

＝プログラム実行を妨げる予定外の事象

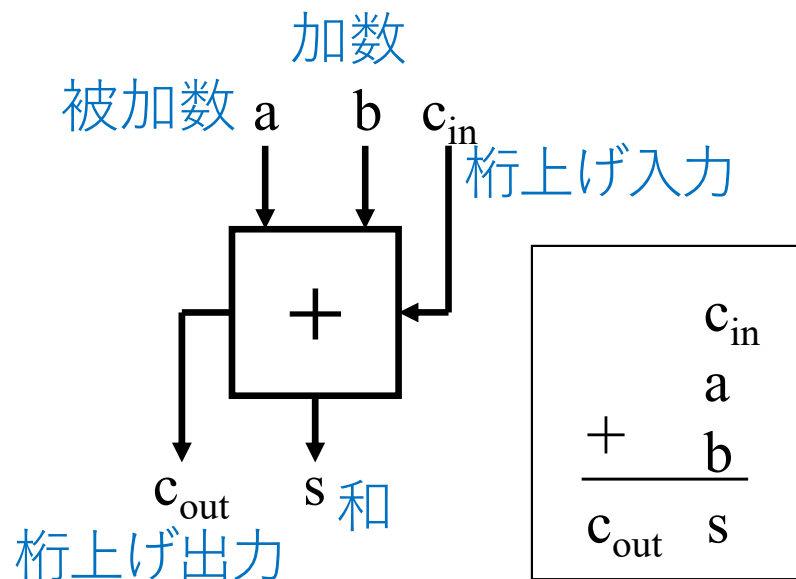
(参考) MIPSでは、例外発生時の命令アドレスを退避可能  
(例外プログラム・カウンタに格納される)



- 2進数の加算と減算（教科書3.2節, 付録(B.2, B.3, B.5)の一部 + $\alpha$ )
  - 実行の様子 **ポイント：最上位からの桁上げは無視してOK**
  - オーバフロー（オーバーフローというのが一般的かもしれないが教科書に合わせます）
  - **ハードウェアの概要**（加算器⇒ALU）
- 乗算の実現方法（教科書3.3節）
  - 基本的な乗算アルゴリズムとそのハードウェア
  - その改良
- 除算の実現方法（教科書3.4節）
  - 基本的な除算アルゴリズムとそのハードウェア
- 浮動小数点形式による実数の演算（教科書3.5節）
  - 仮数、指数、正規化
  - IEEE754浮動小数点規格
  - 演算精度と丸め
  - 浮動小数点演算の実現方法

2022年度は  
試験範囲外

# 全加算器 (1ビット加算器)



- 2進数の加算の1桁分の計算をする回路
- 3つの 1 bit の入力  $a$ ,  $b$ ,  $c_{in}$  の中で値が1である入力の数を 2bit の2進数( $c_{out}$  と  $s$ )で出力する

	0		0		0		1		0		1		1		1		1
	0		0		1		0		1		0		1		1		1
+	0	+	1	+	0	+	0	+	1	+	1	+	0	+	1		
<hr/>		<hr/>		<hr/>		<hr/>		<hr/>		<hr/>		<hr/>		<hr/>		<hr/>	
0	0	0	1	0	1	0	1	1	0	1	0	1	0	1	1	1	1

a	b	$c_{in}$	1の数	$c_{out}$	s
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

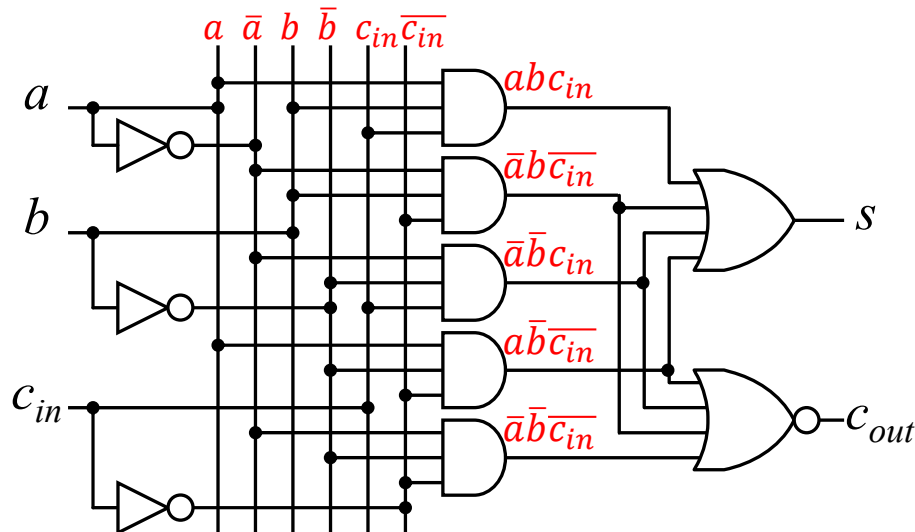
- 左の真理値表を丸暗記する必要は無い (参考程度)。
- 全加算器の名前や機能は理解しておいて欲しい。

# 全加算器（1ビット加算器）の実現（1）

- 全加算器の真理値表やカルノー図から  $s$  や  $c_{out}$  を表す論理式を求めて変形すると、例えば下の回路が得られる

$$s = \bar{a} \bar{b} c_{in} + \bar{a} b \bar{c}_{in} + a \bar{b} \bar{c}_{in} + abc$$

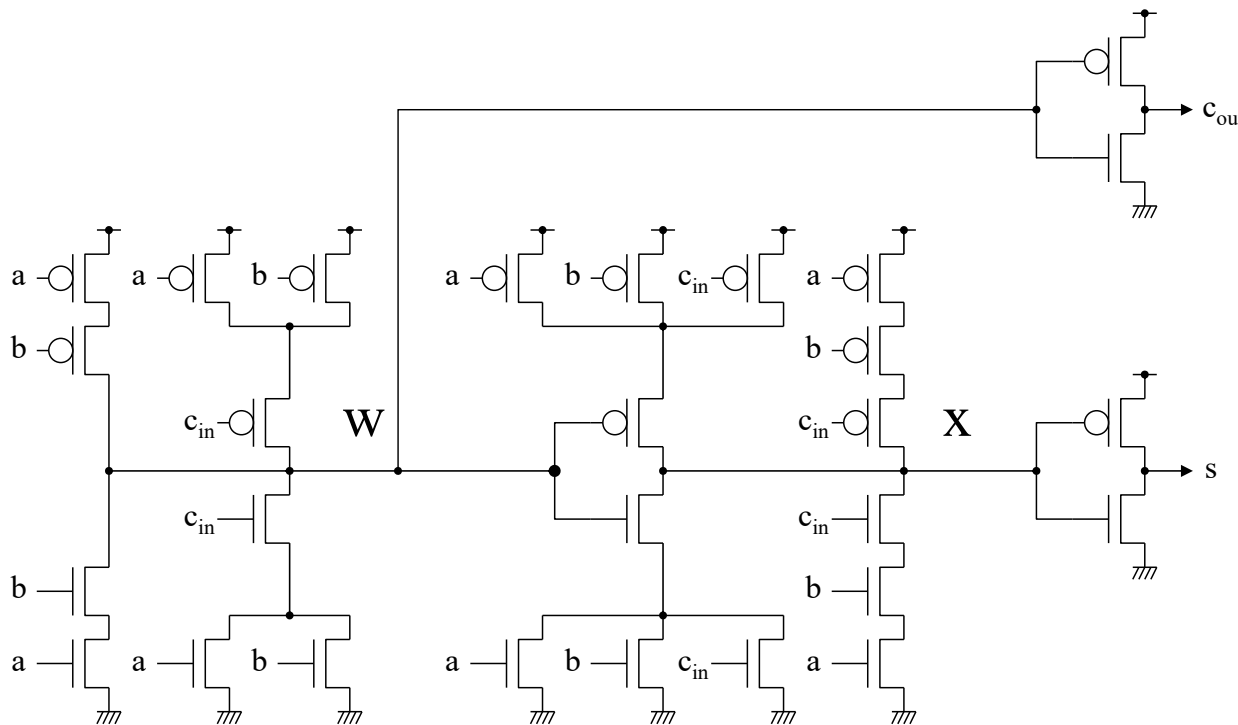
$$c_{out} = \bar{a} \bar{b} c_{in} + \bar{a} b \bar{c}_{in} + a \bar{b} \bar{c}_{in} + \bar{a} \bar{b} \bar{c}_{in}$$



- 論理回路で学んだ知識の範囲でも上のような回路図は導出できる筈。
- しかし、これを元にCMOS論理回路に置き換えると、トランジスタ数は54個程度になり、あまり実用的ではない。

# 全加算器（1ビット加算器）の実現（2）

- 実際に使われている全加算器の回路図例を以下に示す。
- $c_{out}$  を生成する回路と、それを用いて  $s$  を生成する回路から構成されており、巧妙である。



$$w = \overline{ab + (a + b)c_{in}}$$

$$c_{out} = \bar{w}$$

$$= ab + (a + b)c_{in}$$

$$x = \overline{(a + b + c_{in})w + abc_{in}}$$

$$= \overline{ab\bar{c}_{in} + a\bar{b}\bar{c}_{in} + \bar{a}b\bar{c}_{in} + abc_{in}}$$

$$s = \bar{a}\bar{b}\bar{c}_{in} + \bar{a}b\bar{c}_{in} + a\bar{b}\bar{c}_{in} + abc_{in}$$

- 上の図のように、トランジスタ28個で実現でき、実用的である。
- 上の回路図が全加算器の真理値表通りの動作をすることは、電気電子回路で学んだ知識を駆使して根気強く回路図と論理式の対応を追っていけば理解できるかも知れない（興味があれば、是非...）。
- 逆に、真理値表から上の回路図を導出する（発明する）のは、容易ではない。

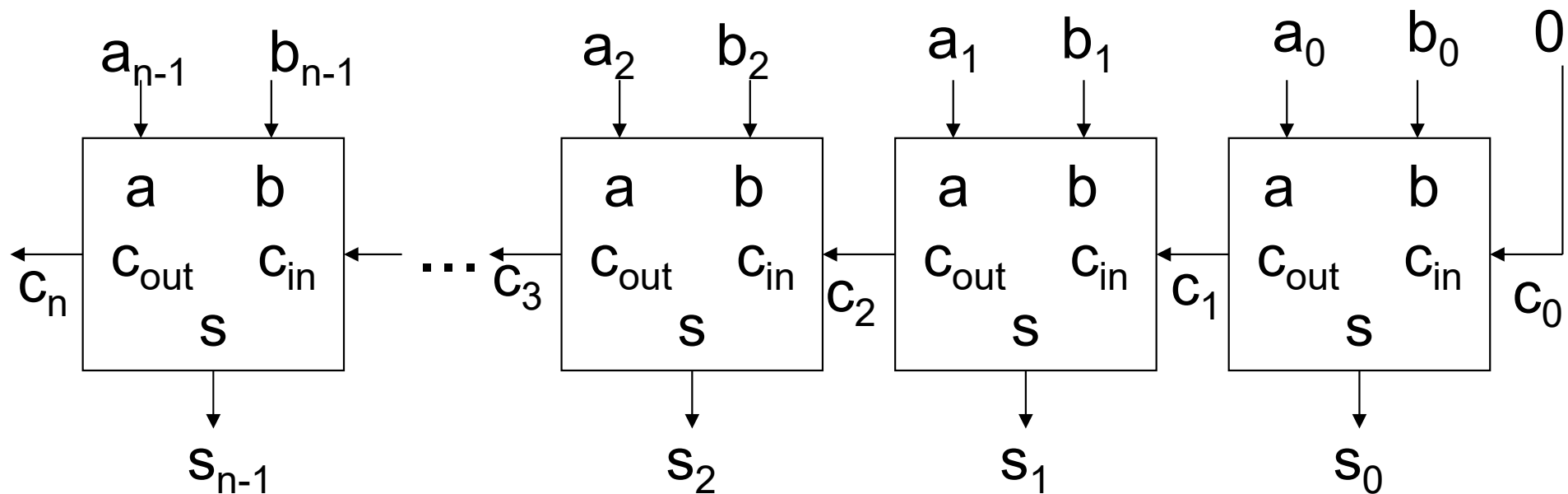
# 算術演算回路の設計

## 1 ビットの演算から多ビットの演算へ

- 1 ビットのモジュールは最適化（例えば前頁の全加算器）
- 多ビット化を考えながら 1 ビット分を設計
- 1 ビットのモジュールを複数用いて多ビットの回路を設計

# n ビットの順次桁上げ加算器

- 全加算器を n 個並べたもの
- ripple-carry-adder (RCA) と呼ぶ
- 計算時間 (最大の段数) は n に比例

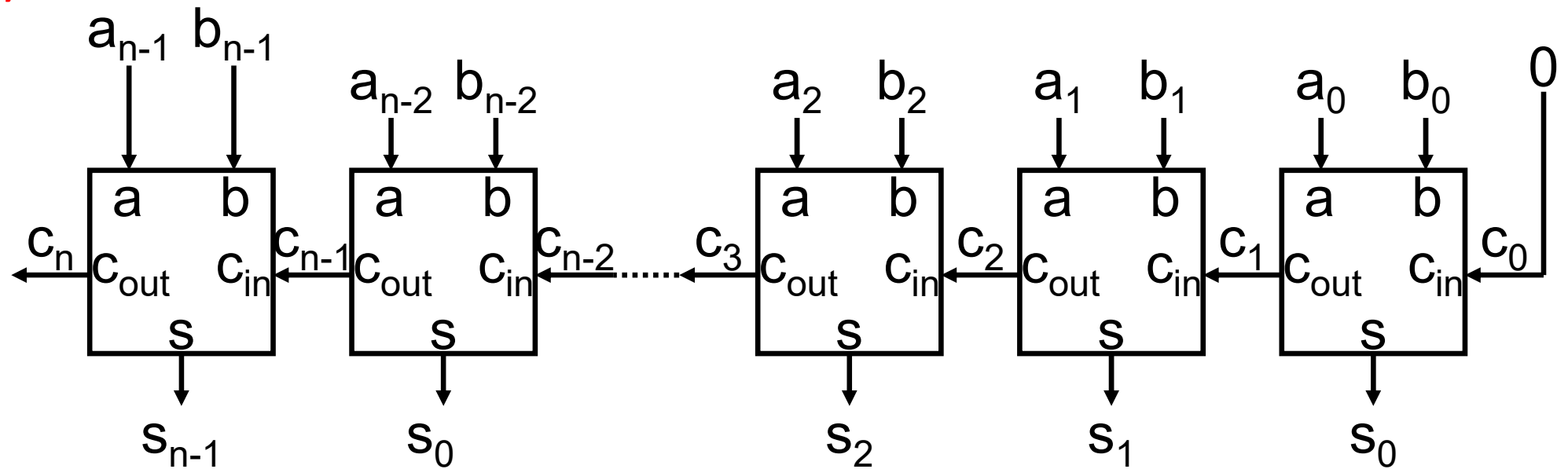


$$(a_{n-1}a_{n-2} \dots a_0) + (b_{n-1}b_{n-2} \dots b_0)$$

# 順次桁上げ加算器のオーバフロー検出回路

- 下の  $n$  bit 順次桁上げ加算器が2の補数表現の加算をする場合に、オーバフロー発生時に 1 となる信号を生成する回路を付け加えよ

## (1) スライドP.10の方法



操作	オペランドA	オペランドB	結果
A+B	$A \geq 0$	$B \geq 0$	$A+B < 0$
A+B	$A < 0$	$B < 0$	$A+B \geq 0$

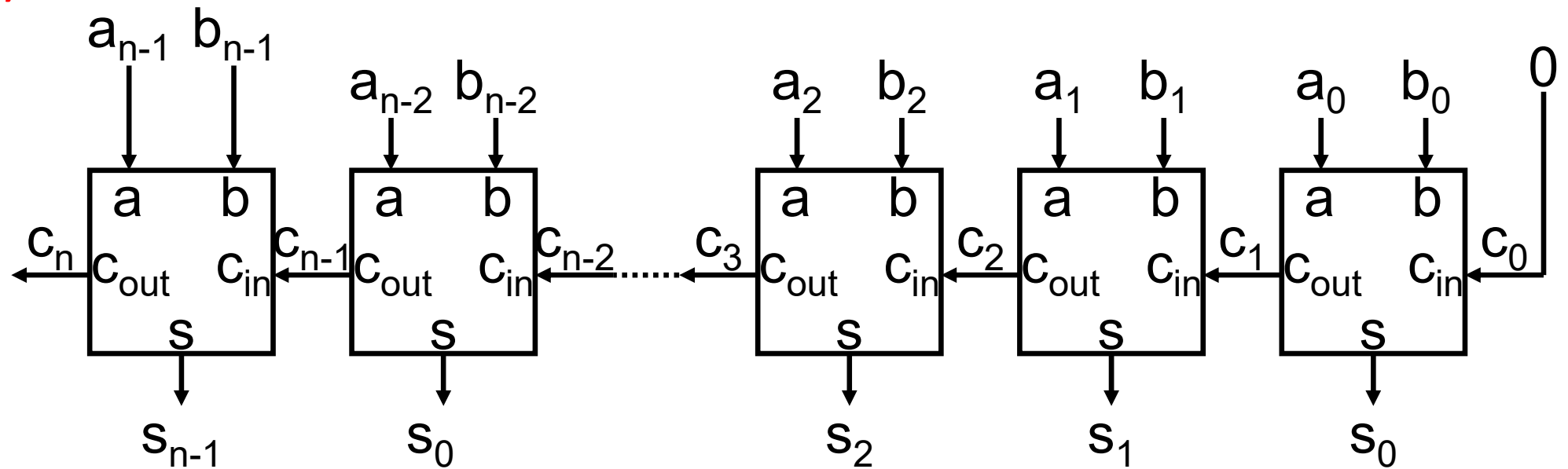
— V

$$V = \overline{a_{n-1}} \overline{b_{n-1}} s_{n-1} + a_{n-1} b_{n-1} \overline{s_{n-1}}$$

# 順次桁上げ加算器のオーバーフロー検出回路

- 下の  $n$  bit 順次桁上げ加算器が2の補数表現の加算をする場合に、オーバーフロー発生時に1となる信号を生成する回路を付け加えよ

## (2) スライドP.13の方法



- 最上位（符号ビット）に入ってくる桁上げと出て行く桁上げが違う場合にオーバーフローと判定すればよい

$\angle V$

$$V = c_{n-1} \oplus c_n \quad (\oplus \text{ は EX-OR})$$



復習ミニクイズ：  $x$  から  $-x$  を作るにはどうすればいいか？

復習

# 加算・減算のハードウェア

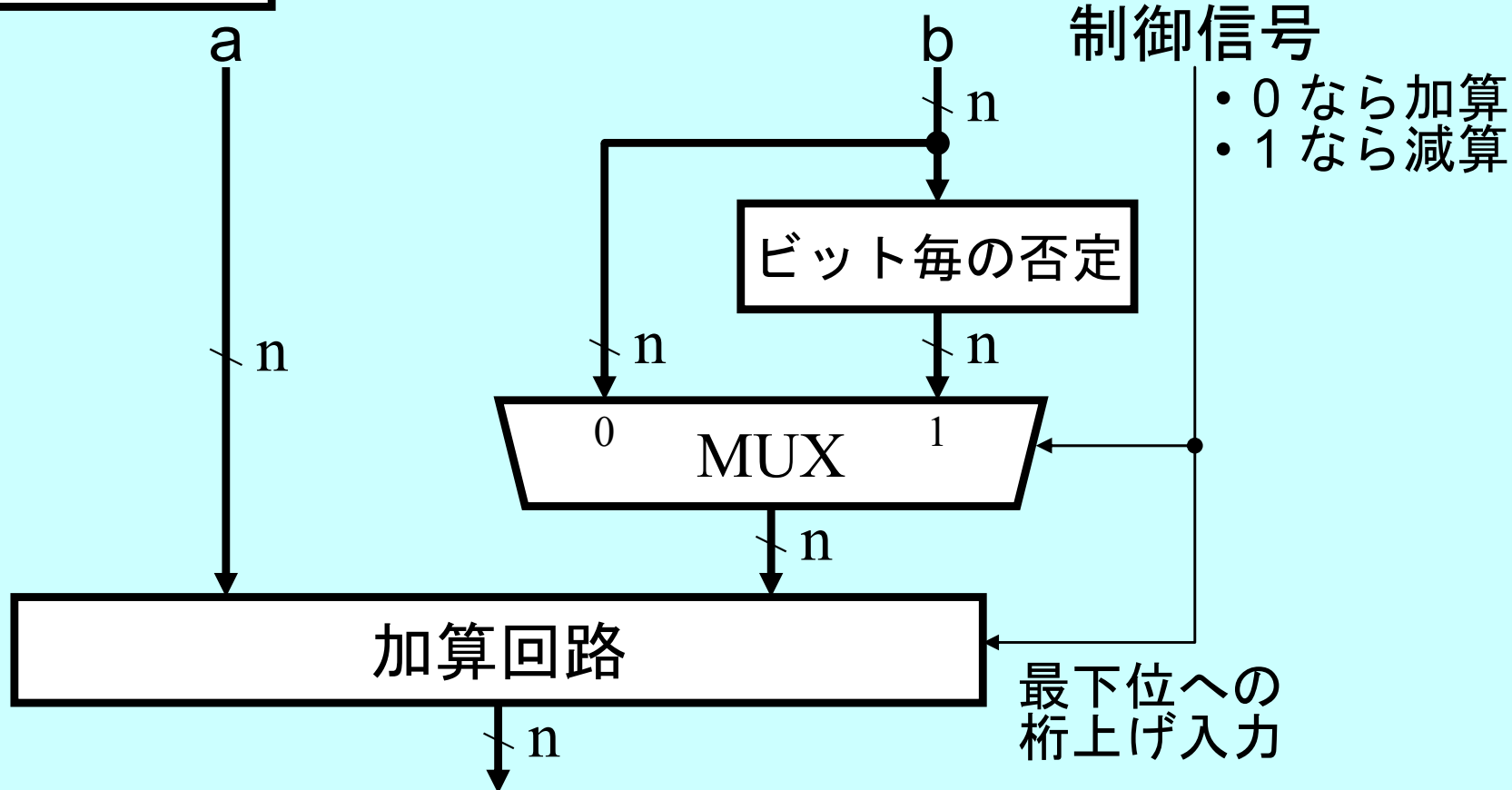
加算と減算のハードウェア  
は共通化可能！

- $a - b = a + (-b)$

$-b$  は  $b$  をビット毎で否定し、最下位に1を足す。

2の補数＝ビット毎の否定を行い、最下位ビットに1を足す。

## 加減算回路



## Lec3の復習テスト

(問い) 1100を右および左に1ビット算術シフト、論理シフトせよ

$1100_2$  ( $12_{10}$ ) を右に1ビット論理シフト 一>

$1100_2$  ( $-4_{10}$ ) を右に1ビット算術シフト 一>

$1100_2$  ( $12_{10}$ ) を左に1ビット論理シフト 一>

$1100_2$  ( $-4_{10}$ ) を左に1ビット算術シフト 一>

# 論理演算 (シフト演算)

## 論理シフト演算

ビットを左または右にずらし、  
空いた部分に0を挿入する操作

3ビット左にシフト

00001101 (13)

⇒ 01101000 (104)

C言語的な記述だと

$13_{10} \ll 3_{10} = 104_{10}$

1ビット右にシフト

00011010 (26)

⇒ 00001101 (13)

C言語的な記述だと

$26_{10} \gg 1_{10} = 13_{10}$

## 算術シフト演算

論理シフトとほぼ同じだが、

- 右シフトは符号ビットと同じ値を挿入
- 1ビット左シフトするごとに2倍、
- 右シフトするごとに1/2

## shift left logical命令

sll \$t2, \$s0, 8

# レジスタ\$t2 = レジスタ\$s0 << 8ビット  
左へ8ビットシフト

## shift right logical命令

srl \$t2, \$s0, 8

# レジスタ\$t2 = レジスタ\$s0 >> 8ビット  
右へ8ビットシフト

\*sraは右に算術シフト

# 論理演算 (AND, OR演算)

## AND演算

(ビットマスク操作とも呼ばれる)

それぞれの桁のビットの  
論理積(AND)をとる

00001101
& 11001001
<div></div>

## and命令

```
and    $t0, $t1, $t2
# レジスタ $t0 = レジスタ $t1 & レジスタ $t2
```

## andi命令 (andの即値命令)

```
andi    $t0, $t1, 100
# レジスタ $t0 = レジスタ $t1 & 100 (即値)
```

## OR演算

それぞれの桁のビットの  
論理和(OR)をとる

00001101
11001001
<div></div>

## or命令

```
or     $t0, $t1, $t2
# レジスタ $t0 = レジスタ $t1 | レジスタ $t2
```

## ori命令 (orの即値命令)

```
ori     $t0, $t1, 100
# レジスタ $t0 = レジスタ $t1 | 100 (即値)
```

# 即値の拡張

即値は16ビット, MIPSの演算は, 32ビット

ポイント！

即値は16ビット → 32ビットの拡張をする

andiやoriは論理演算なので、即値は  拡張

addiやsubiは算術演算なので、即値は  拡張

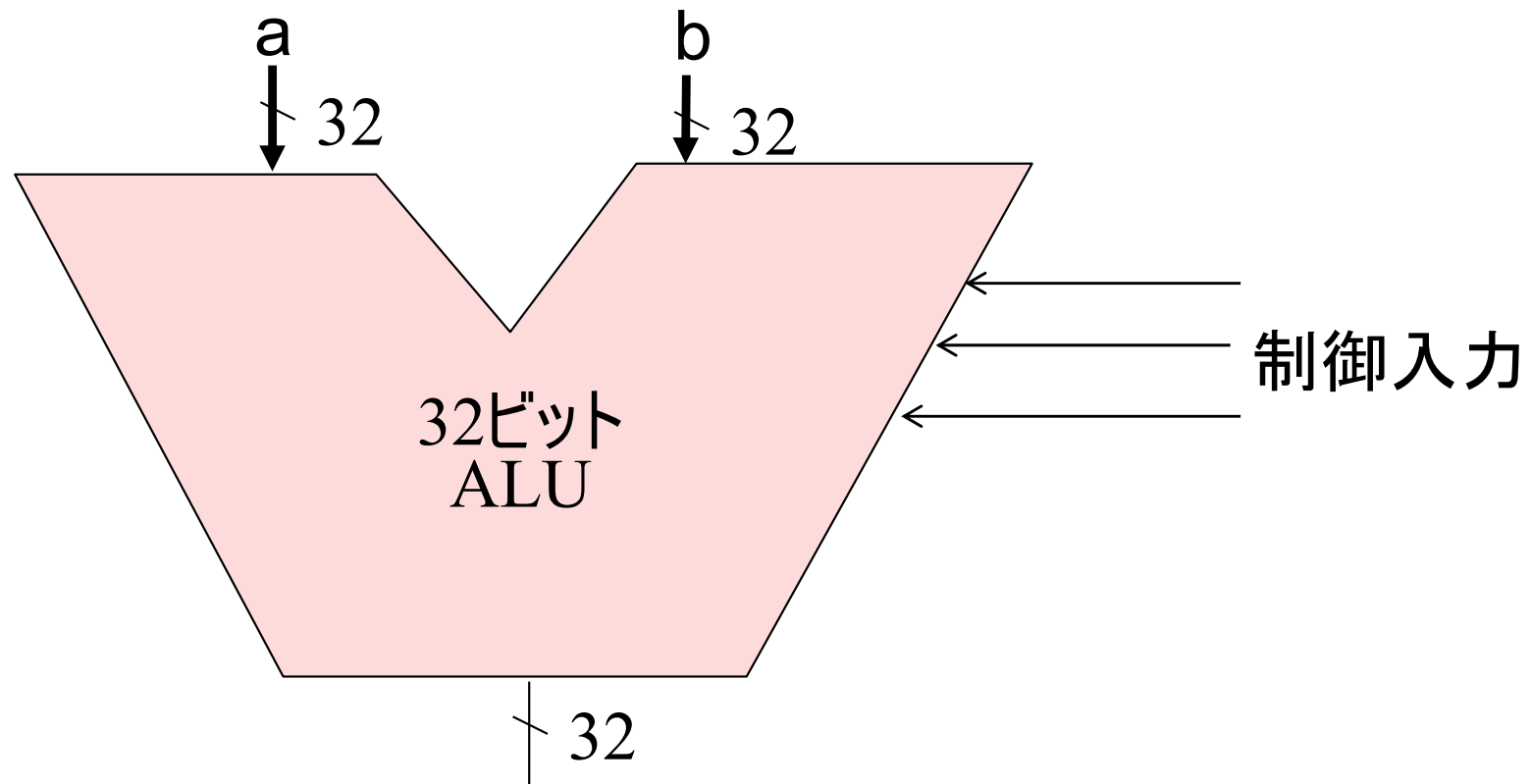
## 命令実行に必要なハードウェアの構成要素

- **ALU**: 算術論理演算ユニット  の略
  - 算術演算と論理演算をする演算器
- **メモリ**: 実行するプログラムやデータを格納
  - 容量の大きい主記憶
  - 容量が小さいが高速なキャッシュメモリ
- **レジスタ**: 高速なプロセッサ内のメモリ
  - MIPSではALUの入出力となる（他のほとんどのアーキテクチャでも）
- **PC: Program Counter** の略
  - 「現在実行中の命令」が格納されているアドレスを保持

# ALU (Arithmetic Logic Unit)

- プロセッサに含まれる演算器で、通常、多数の演算機能を持つ
  - 加算、減算
  - ビット毎の AND, OR, NOT, EXOR
  - 比較命令, ....

今まで見てきた演算は、その実現回路に共通部分が多いので、まとめた方がお得



制御入力により、 $a+b$ ,  $a-b$ ,  $a \text{ (and) } b$ ,  $a \text{ (exor) } b$ , .....



## 演習問題 その③

2つのレジスタ\$t3と\$t4を加算した時に、最上位ビットからのキャリーアウトがあるかどうかを判定するMIPSの命令列を考えよ。具体的には、キャリーアウトがある場合に\$t2=1, そうでない場合に\$t2=0とセットする命令列を答えよ。

ヒント 1 : 2命令で可能である

ヒント 2 : sltu という命令を用いよ

sltu \$t1, \$t2, \$t3

- slt 命令の符号なしバージョン
- 符号を無視した比較を行い、
  - $t2 < t3$  なら  $t1 = 1$  とする
  - そうでないなら  $t1 = 0$  とする

# 演習問題 その③解答

1 命令目でオーバーフローを無視した\$t3と\$t4の加算を行い、結果を\$t2に格納する命令を実行することとしよう。

このとき最上位ビットから桁上がりがあったなら\$t2=1とするような命令を2 命令目で実行すればよい。

一般に、符号を無視した加算  $C=A+B$  ( $A \geq 0, B \geq 0$ ) において、 $C \geq A$  かつ  $C \geq B$  が成り立つ筈である。しかし、加算時に最上位ビットから桁上り（重み  $2^n$ ）が発生した場合、桁上りは結果のレジスタに格納されない。従って、レジスタに格納された演算結果は、 $C' = A+B-2^n$  となる。このとき、 $B < 2^n$  から、 $C' < A$  となる（同様の議論から、 $C' < B$  も成り立つ）。

よって、1 命令目終了時の\$t2の値が\$t3や\$t4より小さくなっていたら、1 命令目の足し算のときに最上位ビットから桁上がりがあったことになる。

以上より、求める命令列は以下の通り（どちらでも可）。

```
addu $t2, $t3, $t4
```

```
sltu $t2, $t2, $t4 # $t2 < $t4 なら $t2=1 (さもなくば0) とする
```

または、

```
addu $t2, $t3, $t4
```

```
sltu $t2, $t2, $t3
```

- 2進数の加算と減算（教科書3.2節, 付録(B.2, B.3, B.5)の一部 + $\alpha$ )
  - 実行の様子 **ポイント：最上位からの桁上げは無視してOK**
  - オーバフロー（オーバーフローというのが一般的かもしれないが教科書に合わせます）
  - ハードウェアの概要（加算器⇒ALU）
- **乗算の実現方法**（教科書3.3節）
  - 基本的な乗算アルゴリズムとそのハードウェア
  - その改良
- 除算の実現方法（教科書3.4節）
  - 基本的な除算アルゴリズムとそのハードウェア
- 浮動小数点形式による実数の演算（教科書3.5節）
  - 仮数、指数、正規化
  - IEEE754浮動小数点規格
  - 演算精度と丸め
  - 浮動小数点演算の実現方法

2022年度は  
試験範囲外

# 乗算

- ※ 以下の説明では簡単のため、符号なし整数同士の乗算を想定する。
- ※ 従って、アルゴリズム中のシフト演算は全て論理シフトとする。

- 基本的な流れは10進数の場合と同じ

0010	被乗数
× 0011	乗数
-----	
0010	部分積0
0010	部分積1
0000	部分積2
0000	部分積3
-----	
0000110	積

## $n$ ビット乗算の流れ

### 1. 部分積 $i$ ( $0 \leq i < n$ ) を生成する

- 乗数の下位から $i$ ビット目が1の場合  
⇒ 部分積 $i$ は、被乗数を $i$ 桁左シフトしたもの
- 乗数の下位から $i$ ビット目が0の場合  
⇒ 部分積 $i$ は、0

### 2. 部分積の総和（積）を求める

被乗数が $n$ ビット、乗数が $m$ ビットのとき、積は   ビット  
⇒ 乗数、被乗数より多くのビット数が必要（オーバーフローに注意）

- 色々な計算法（アルゴリズム）がある

- 被乗数レジスタを左シフトしてから積レジスタに足す
- 被乗数レジスタを積レジスタの上位桁に足してから積レジスタを右シフトする
- 更に、積レジスタの空きビットを活用して乗数レジスタを削除

# 第1の乗算アルゴリズム：実行の様子

教科書p180

例：4ビットを用いて、 $2_{10} \times 3_{10}$  ( $0010_2 \times 0011_2$ ) を計算

		4ビット	8ビット	8ビット
	ステップ	乗数	被乗数	積
0	初期値	0011	<u>0000 0010</u>	0000 0000
1	1a: 1 → 積 = 積 + 被乗数	0011	0000 0010	0000 0010
	2: 被乗数を左へシフト	0011	<u>0000 0100</u>	0000 0010
	3: 乗数を右へシフト	0001	0000 0100	0000 0010
2	1a: 1 → 積 = 積 + 被乗数	0001	0000 0100	0000 0110
	2: 被乗数を左へシフト	0001	<u>0000 1000</u>	0000 0110
	3: 乗数を右へシフト	0000	0000 1000	0000 0110
3	1: 0 → 演算なし	0000	0000 1000	0000 0110
	2: 被乗数を左へシフト	0000	<u>0001 0000</u>	0000 0110
	3: 乗数を右へシフト	0000	0001 0000	0000 0110
4	1: 0 → 演算なし	0000	0001 0000	0000 0110
	2: 被乗数を左へシフト	0000	0010 0000	0000 0110
	3: 乗数を右へシフト	0000	0010 0000	0000 0110

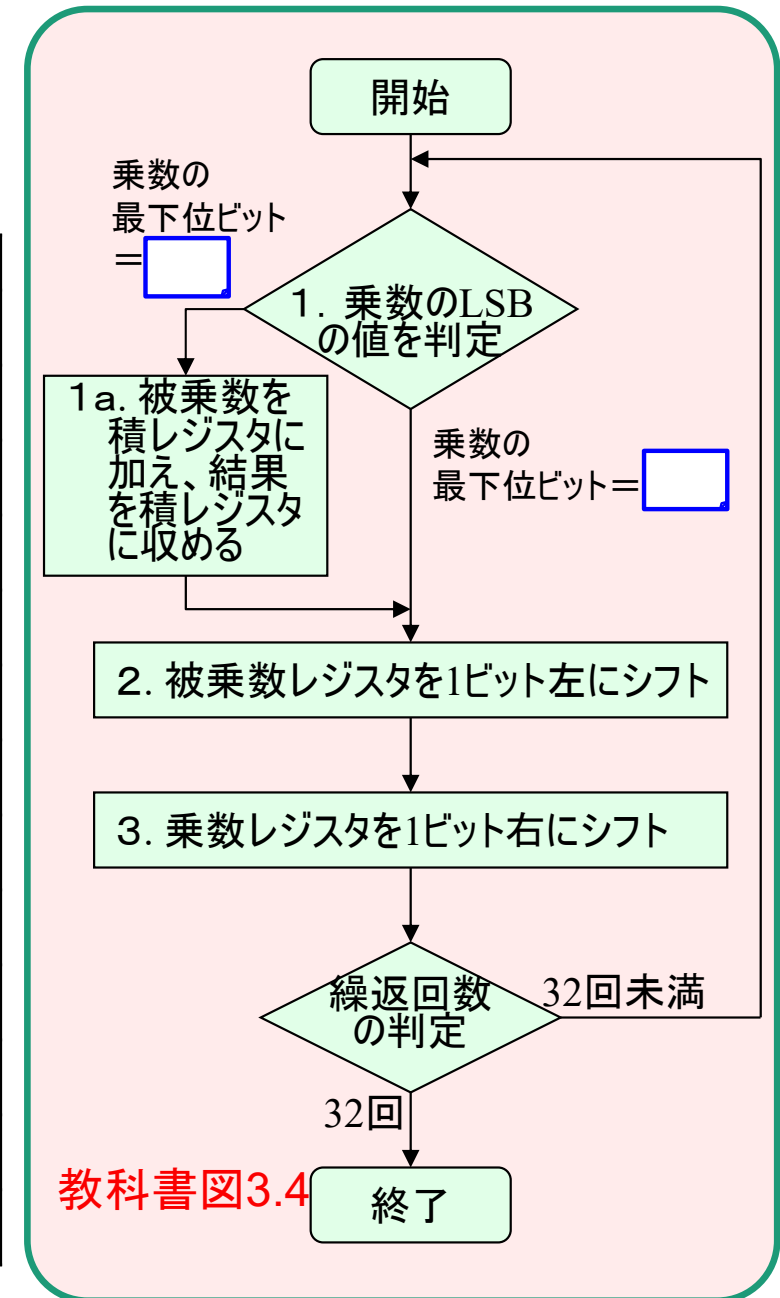
0010 被乗数  
× 0011 乗数  
-----  
00000010 部分積  
00000100 (被乗数レジスタ)  
00001000  
00010000  
-----  
00000110 積

計算結果は  $00000110_2 = 6_{10}$

# 第1の乗算アルゴリズム：フローチャート

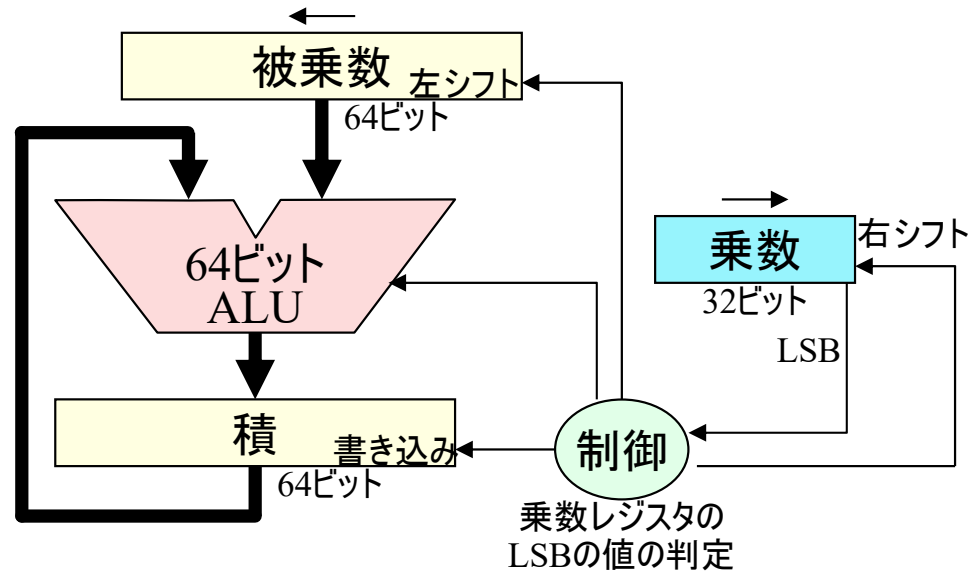
例：4ビットを用いて、 $2_{10} \times 3_{10}$  ( $0010_2 \times 0011_2$ ) を計算

		4ビット	8ビット	8ビット
	ステップ	乗数	被乗数	積
0	初期値	0011	<u>0000 0010</u>	0000 0000
1	1a: 1 → 積 = 積 + 被乗数	0011	0000 0010	0000 0010
	2: 被乗数を左へシフト	0011	<u>0000 0100</u>	0000 0010
	3: 乗数を右へシフト	0001	0000 0100	0000 0010
2	1a: 1 → 積 = 積 + 被乗数	0001	0000 0100	0000 0110
	2: 被乗数を左へシフト	0001	<u>0000 1000</u>	0000 0110
	3: 乗数を右へシフト	0000	0000 1000	0000 0110
3	1: 0 → 演算なし	0000	0000 1000	0000 0110
	2: 被乗数を左へシフト	0000	<u>0001 0000</u>	0000 0110
	3: 乗数を右へシフト	0000	0001 0000	0000 0110
4	1: 0 → 演算なし	0000	0001 0000	0000 0110
	2: 被乗数を左へシフト	0000	0010 0000	0000 0110
	3: 乗数を右へシフト	0000	0010 0000	0000 0110



# 第1の乗算アルゴリズム：ハードウェア

教科書図3.3



初期状態では

積レジスタ =

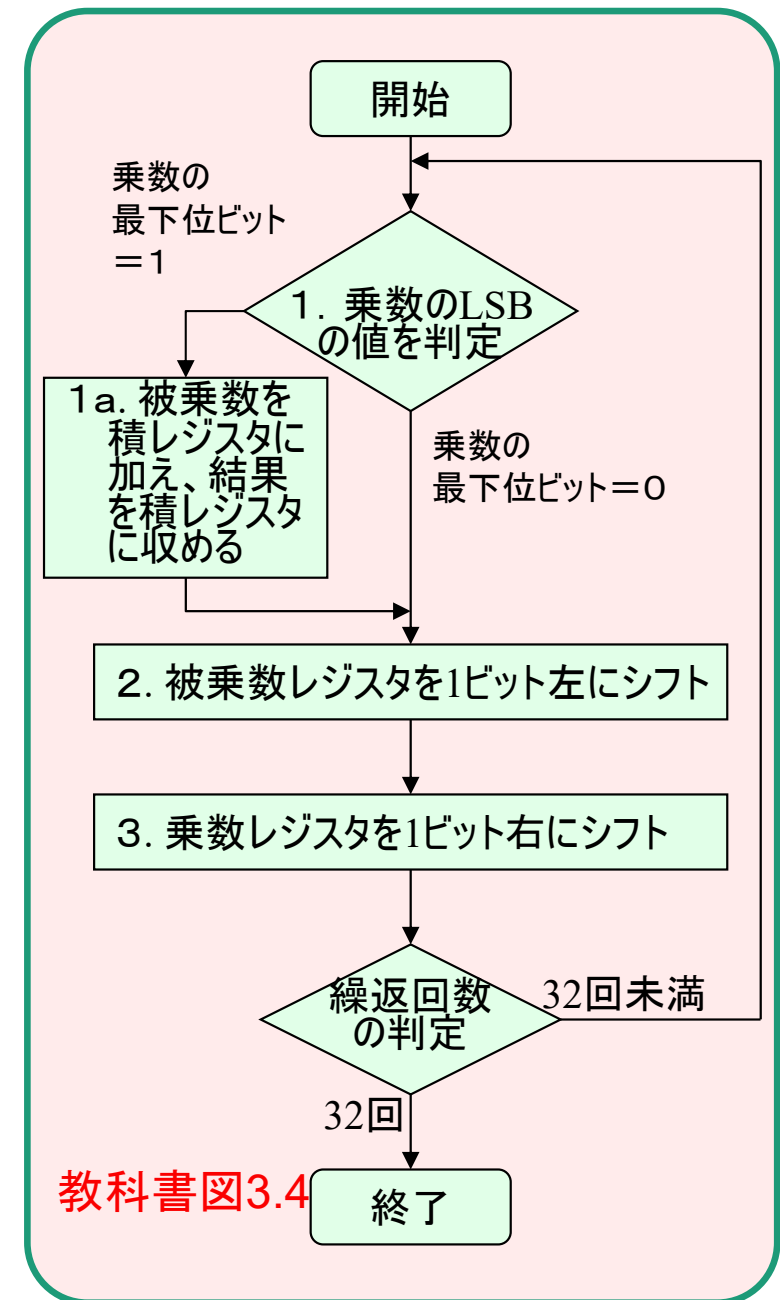
被乗数レジスタ

右半分 = 32ビットの被乗数

左半分 =

Point:

常に部分積を64ビットと思って足していく  
→ (わかりやすいが) ちょっと無駄かも...



教科書図3.4

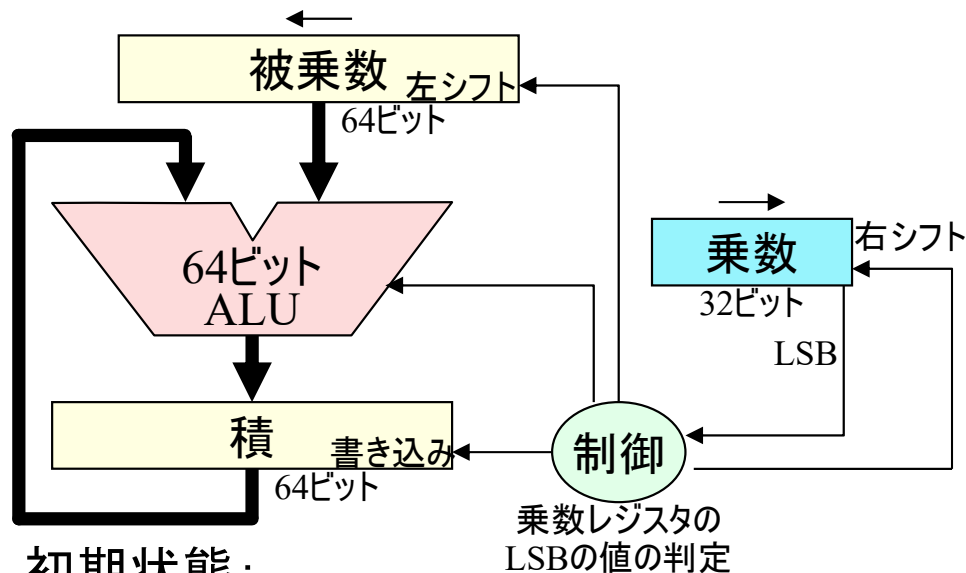
※ フローチャート（右上図）の箱は**操作**を表し、矢印は操作間の順序関係を表す。  
※ 回路図（左上図）の箱は**演算器等**を表し、矢印はデータや制御信号が伝わる方向を表す。

## 第2の乗算アルゴリズム：ALUを32ビットに

このバージョンは教科書第5版では割愛されているが（最終バージョンを理解するために）これをまず理解してください

- 第1バージョンでは、被乗数レジスタの半分が常に0  
⇒ 64ビットALUが無駄（回路面積の無駄、演算の遅延時間も無駄に長い）

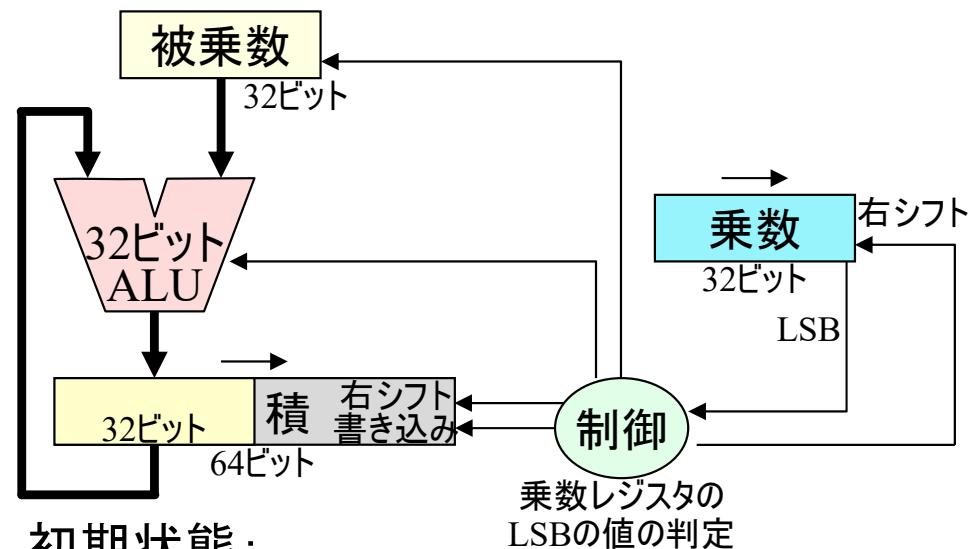
乗算ハードウェアの第1のバージョン



初期状態:

- 積レジスタ=0
- 被乗数レジスタ
  - 右半分=被乗数(32ビット)
  - 左半分=0

乗算ハードウェアの第2のバージョン



初期状態:

- 積レジスタ=0

Point:

- 常に積レジスタの上位32ビットに被乗数を足す
- 被乗数を左シフトする代わりに、積レジスタを右シフトする



## 第2の乗算アルゴリズム：実行の様子


Point:

- 常に積レジスタの上位32ビットに被乗数を足す
- 被乗数を左シフトする代わりに、積レジスタを右シフトする

例：4ビットを用いて、 $2_{10} \times 3_{10}$  ( $0010_2 \times 0011_2$ ) を計算

サイクル	ステップ	乗数	被乗数	積
0	初期値	0011	0010	0000 0000
1	1a:1→積=積+被乗数	0011	0010	0010 0000
	2:積を右へシフト	0011	0010	0001 0000
	3: 乗数を右へシフト	0001	0010	0001 0000
2	1a:1→積=積+被乗数	0001	0010	0011 0000
	2:積を右へシフト	0001	0010	0001 1000
	3: 乗数を右へシフト	0000	0010	0001 1000
3	1a:0→演算なし	0000	0010	0001 1000
	2:積を右へシフト	0000	0010	0000 1100
	3: 乗数を右へシフト	0000	0010	0000 1100
4	1a:0→演算なし	0000	0010	0000 1100
	2:積を右へシフト	0000	0010	0000 0110
	3: 乗数を右へシフト	0000	0010	0000 0110

計算結果は  $00000110_2 = 6_{10}$

00000010を足す  
  
 00100000を足して  
 右に4ビットシフト

ポイント①: 1回右シフトすると何倍になるか?   倍

ポイント②: 以下の式を展開して整理し、第2の乗算アルゴリズムが求める値が第1の乗算アルゴリズムが求める値に一致することを確認せよ

- $S_i$ は、乗数の下から*i*ビット目が1なら被乗数、乗数の下から*i*ビット目が0なら0

積レジスタの初期値

積レジスタの上位4ビットに $S_0$ を足す

積レジスタを右シフト

積レジスタの上位4ビットに $S_1$ を足す

積レジスタを右シフト

積レジスタの上位4ビットに $S_2$ を足す

積レジスタを右シフト

積レジスタの上位4ビットに $S_3$ を足す

積レジスタを右シフト

$$\left( \left( \left( \left( 0 + 2^4 S_0 \right) \times \frac{1}{2} + 2^4 S_1 \right) \times \frac{1}{2} + 2^4 S_2 \right) \times \frac{1}{2} + 2^4 S_3 \right) \times \frac{1}{2}$$

=

(=第1の乗算アルゴリズムで求める値) 41

## 確認クイズ

- 1サイクル目で足された数は、その後合計何回シフトされるか？
- 2サイクル目で足された数は、その後合計何回シフトされるか？

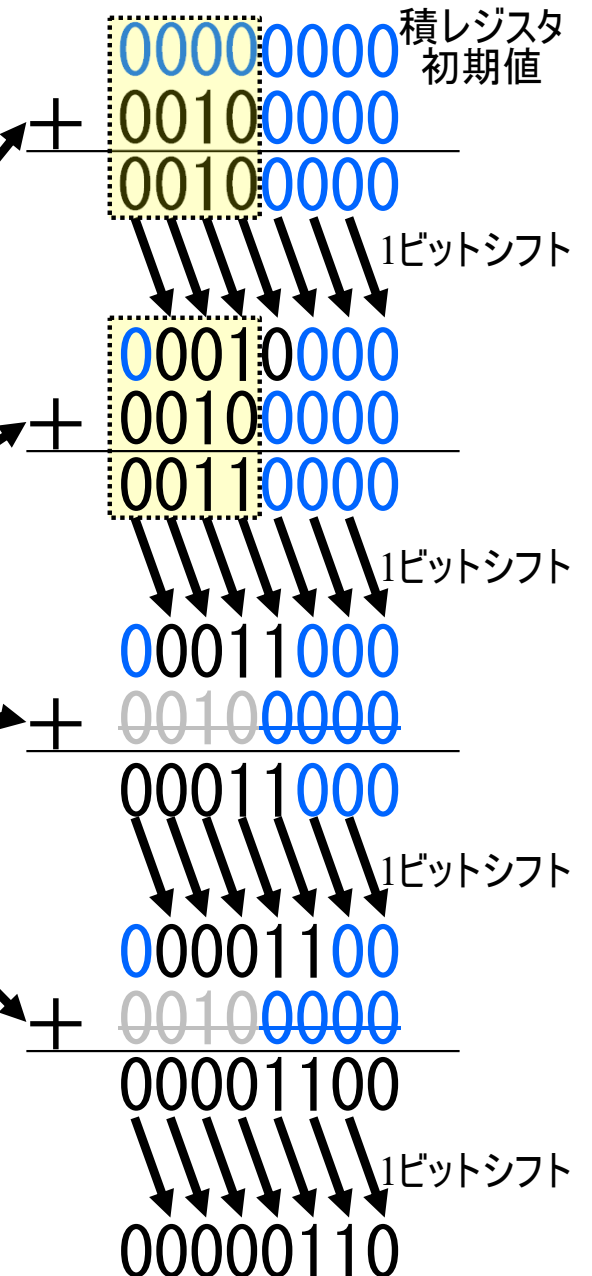
難関

$$\begin{array}{r}
 0010 \text{ 被乗数} \\
 \times 0011 \text{ 乗数} \\
 \hline
 00000010 \text{ 部分積} \\
 00000100 \\
 \hline
 00001000 \\
 00010000 \\
 \hline
 00000110 \text{ 積}
 \end{array}$$

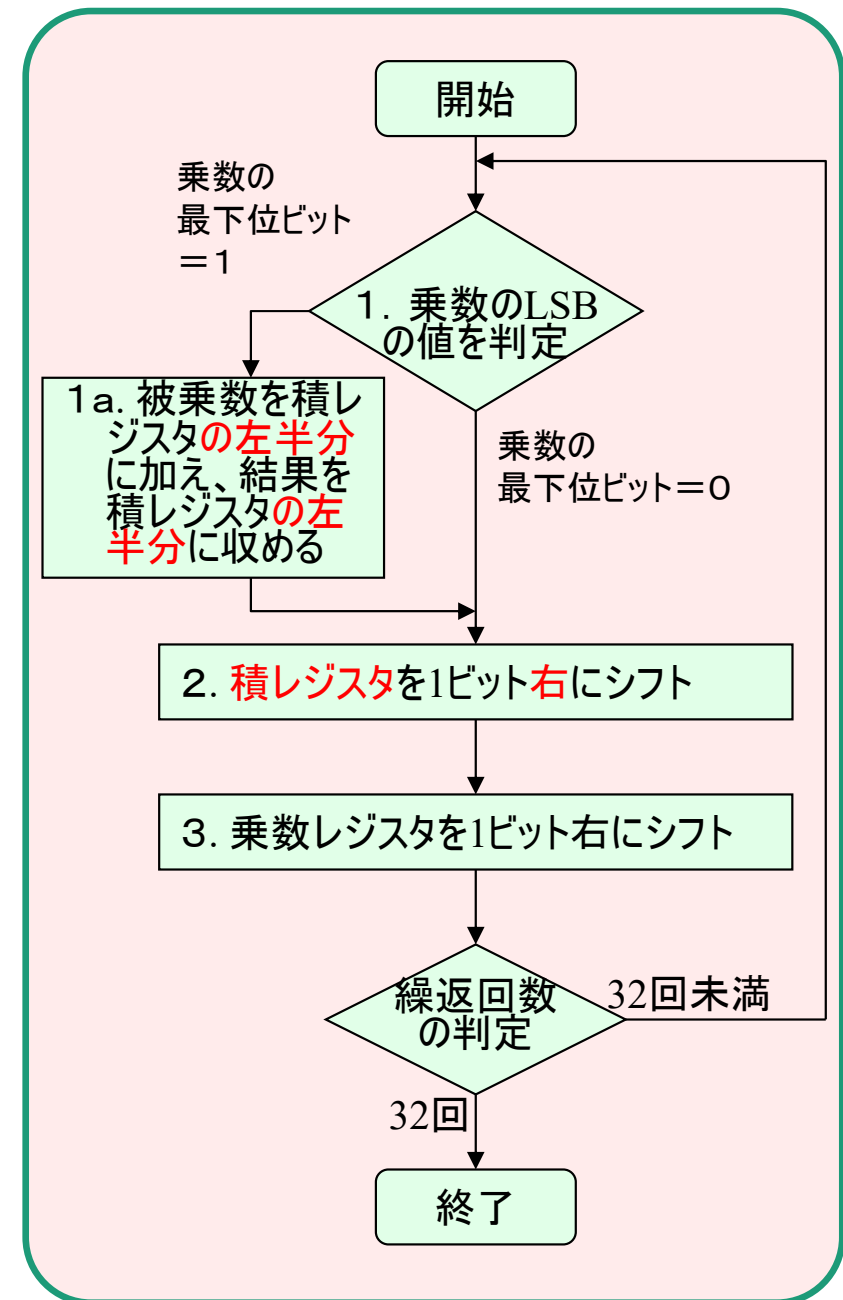
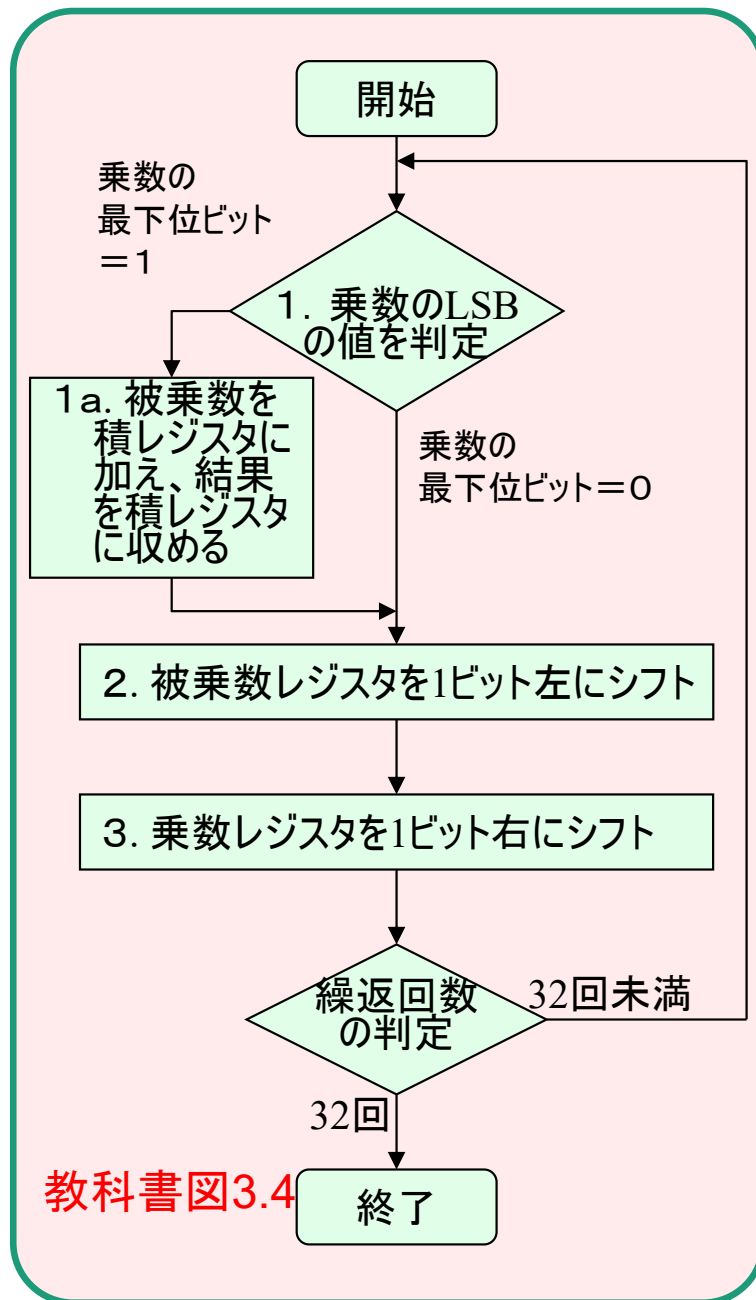
第1のバージョンの考え方  
8ビットの足し算でやる

$$\begin{array}{r}
 0010 \text{ 被乗数} \\
 \times 0011 \text{ 乗数} \\
 \hline
 00100000 \\
 00100000 \\
 \hline
 00100000 \\
 00100000 \\
 \hline
 00000110 \text{ 積}
 \end{array}$$

第2のバージョンの考え方  
4ビットのみの足し算でやる



# 第1と第2のバージョンの比較



でも、第2の乗算アルゴリズムにもまだ無駄がある

4ビットを用いて、 $2_{10} \times 3_{10}$  ( $0010_2 \times 0011_2$ ) を計算

初期 サイクル	ステップ	乗数	被乗数	積
0	初期値	0011	0010	0000 0000
1	1a:1→積=積+被乗数	0011	0010	0010 0000
	2:積を右へシフト	0011	0010	0001 0000
	3: 乗数を右へシフト	0001	0010	0001 0000
2	1a:1→積=積+被乗数	0001	0010	0011 0000
	2:積を右へシフト	0001	0010	0001 1000
	3: 乗数を右へシフト	0000	0010	0001 1000
3	1a:0→演算なし	0000	0010	0001 1000
	2:積を右へシフト	0000	0010	0000 1100
	3: 乗数を右へシフト	0000	0010	0000 1100
4	1a:0→演算なし	0000	0010	0000 1100
	2:積を右へシフト	0000	0010	0000 0110
	3: 乗数を右へシフト	0000	0010	0000 0110

- 積レジスタの一部のビットは使用していない！（上の表の灰色）
- その部分が、ちょうど「まだ情報として必要な乗数のビット」数（上の水色）と一致！

⇒積レジスタの使用していない部分に、乗数の必要なビットを格納すれば、乗数のレジスタをなくせる！

# 演習問題 その④

教科書p180

- 前ページの考え方をういて、4ビットを用いて、 $2_{10} \times 3_{10}$  ( $0010_2 \times 0011_2$ ) を計算する様子を示した以下の表の空欄をせよ。なお、用いるハードウェアは、4ビットのALU、4ビットの被乗数を入れるレジスタ、積(途中も含めて)を入れる8ビットのレジスタである。
- この乗算を32ビットで実現するハードウェアのブロック図を示せ。

サイクル	ステップ	被乗数	積
0	初期値	0010	0000 <u>0011</u>
1	1a:1→積=積+被乗数	0010	0010 <u>0011</u>
	2:積を右へシフト	0010	0001 <u>0001</u>
2	1a:1→積=積+被乗数	0010	
	2:積を右へシフト	0010	
3	1a:0→演算なし	0010	0001 10 <u>00</u>
	2:積を右へシフト	0010	0000 110 <u>0</u>
4	1a:0→演算なし	0010	0000 110 <u>0</u>
	2:積を右へシフト	0010	0000 0110

$$\begin{array}{r}
 0010 \text{ 被乗数} \\
 \times 0011 \text{ 乗数} \\
 \hline
 0010 \rightarrow 00100000 \\
 0010 \rightarrow 00110000 \\
 \hline
 0010 \\
 0010 \\
 \hline
 00000110
 \end{array}$$

第2バージョンの時の積レジスタの様子:  
 下線部が未使用だった!

計算結果は  $00000110_2 = 6_{10}$

# 最終バージョンの乗算アルゴリズムのハードウェア

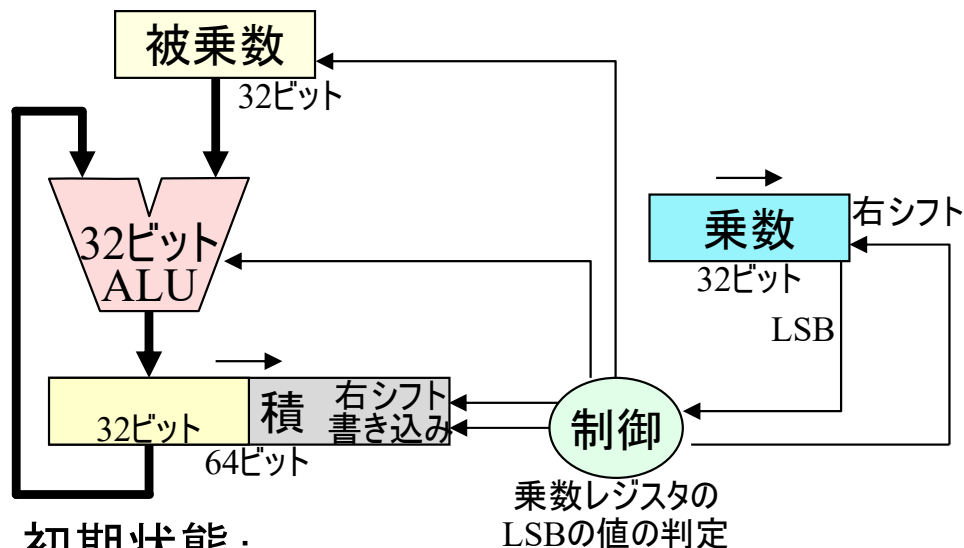
- 第2バージョンでは、積レジスタに無駄なスペース（実は乗数レジスタのスペースに一致）
  - 積レジスタと乗数レジスタの結合

## 演習④のブロック図の解答

### 乗算ハードウェアの最終バージョン

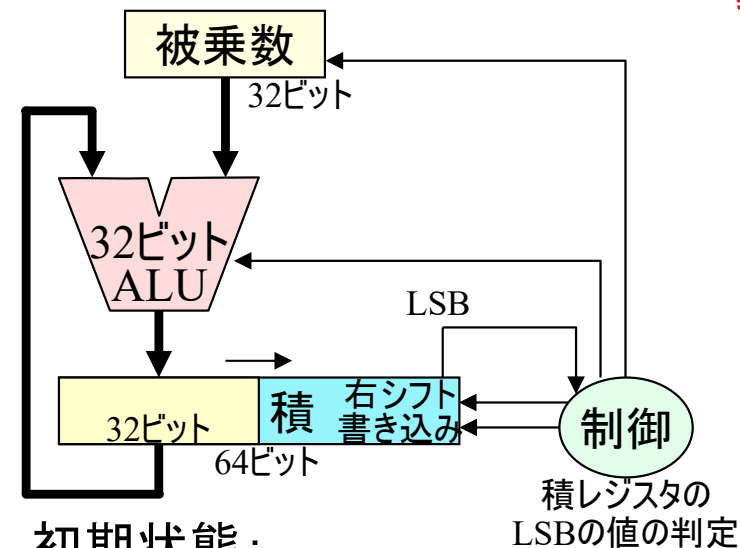
教科書図3.5

### 乗算ハードウェアの第2のバージョン



初期状態:

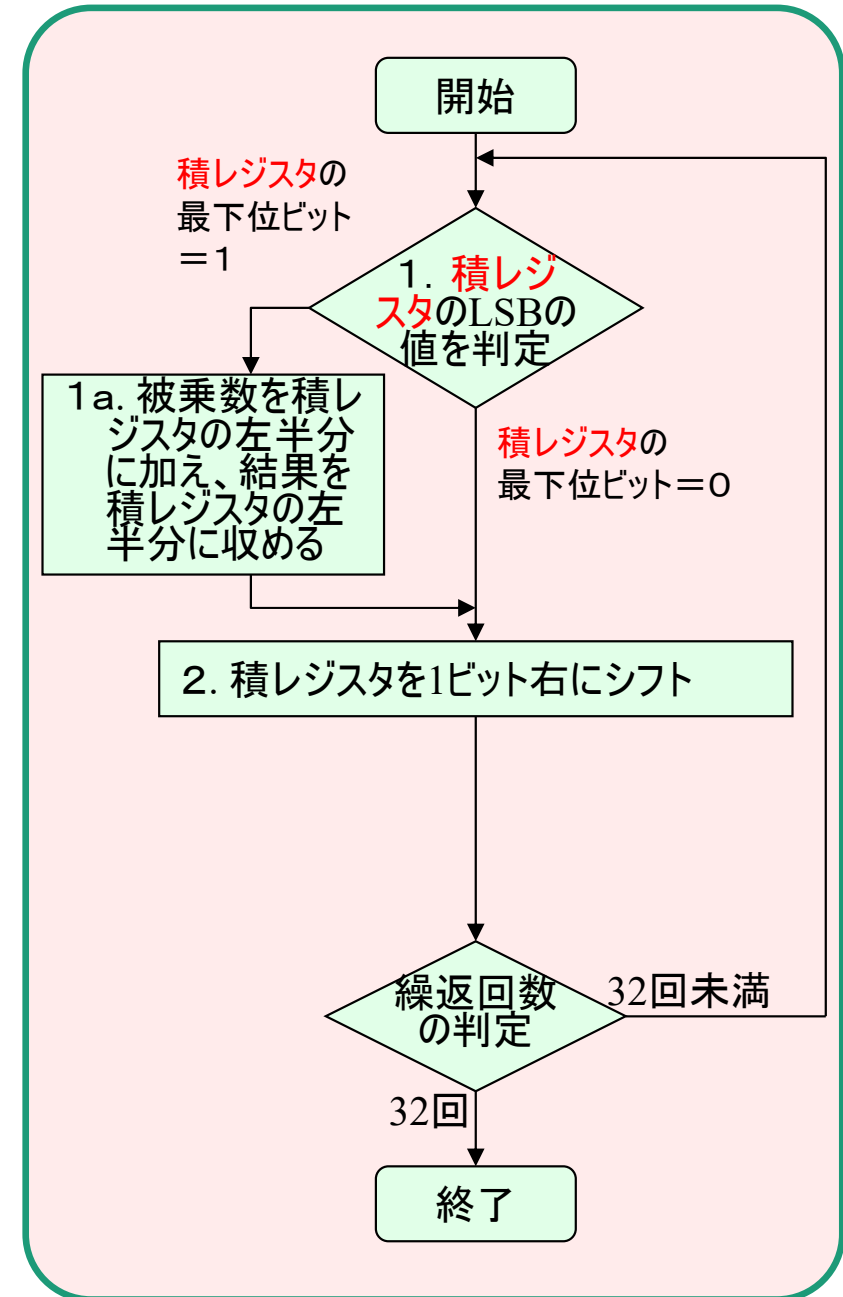
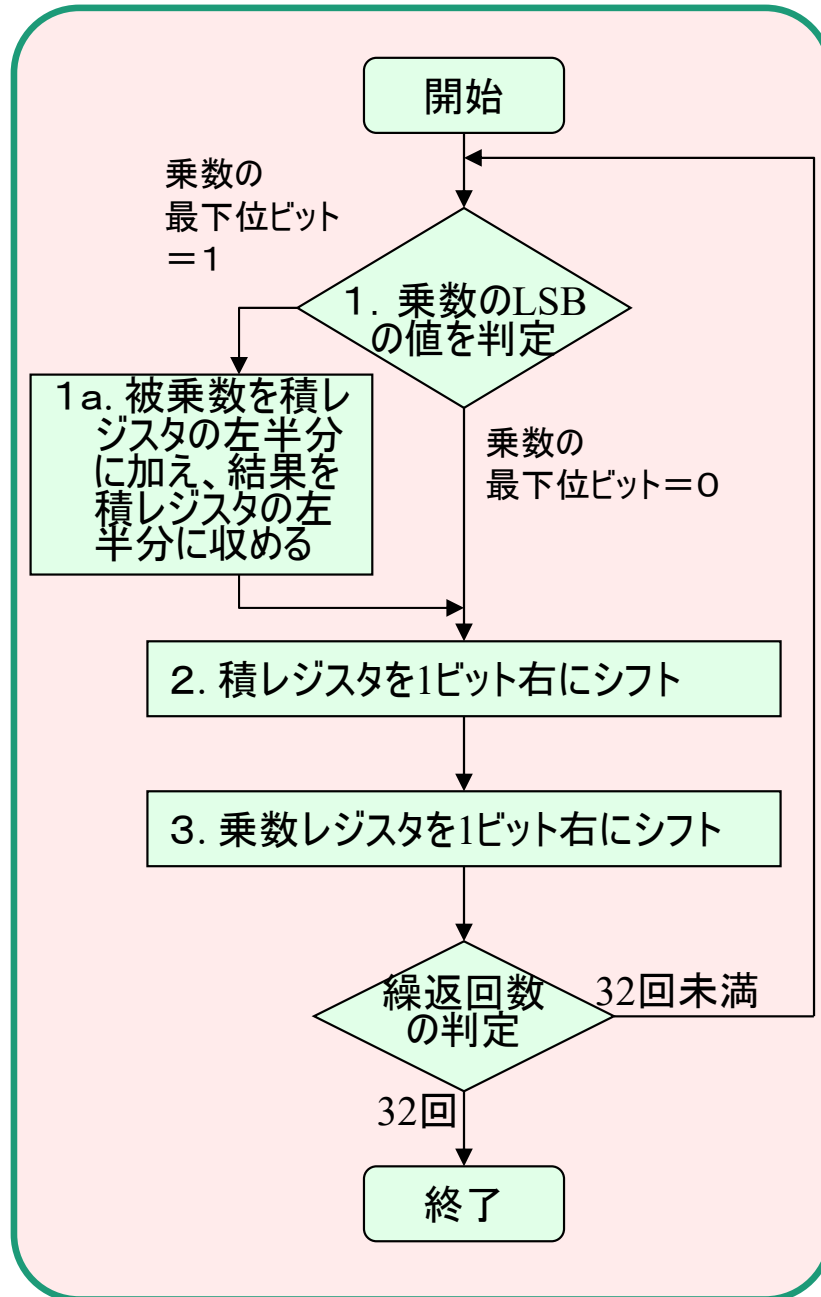
- 積レジスタ=0



初期状態:

- 積レジスタの左半分
- 積レジスタの右半分

## 第2と最終バージョンの比較





Good な質問:部分積の足し算で桁上げが起こる場合はどうするの？

補足説明

## 第2の乗算アルゴリズム：実行の様子

再掲

Point:

- 常に積レジスタの上位32ビットに被乗数を足す
- 被乗数を左シフトする代わりに、積レジスタを右シフトする

例：4ビットを用いて、 $2_{10} \times 3_{10}$  ( $0010_2 \times 0011_2$ ) を計算

サイクル	ステップ	乗数	被乗数	積
0	初期値	0011	0010	0000 0000
1	1a:1→積=積+被乗数	0011	0010	0010 0000
	2:積を右へシフト	0011	0010	0001 0000
	3: 乗数を右へシフト	0001	0010	0001 0000
2	1a:1→積=積+被乗数	0001	0010	0011 0000
	2:積を右へシフト	0001	0010	0001 1000
	3: 乗数を右へシフト	0000	0010	0001 1000
3	1a:0→演算なし	0000	0010	0001 1000
	2:積を右へシフト	0000	0010	0000 1100
	3: 乗数を右へシフト	0000	0010	0000 1100
4	1a:0→演算なし	0000	0010	0000 1100
	2:積を右へシフト	0000	0010	0000 0110
	3: 乗数を右へシフト	0000	0010	0000 0110

計算結果は  $00000110_2 = 6_{10}$

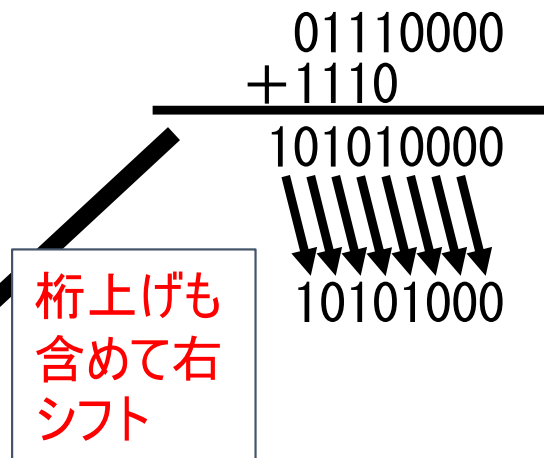
00000010を足す  
↓  
00100000を足して  
右に4ビットシフト

## 第2の乗算アルゴリズム：実行の様子

教科書には書かれていないが、  
ステップ 1a で得られた桁上げを捨てずに持っておき、  
ステップ 2 で、桁上げも含めて右シフトすればOK

例：4ビットを用いて、 $14_{10} \times 3_{10} (1110_2 \times 0011_2)$  を計算

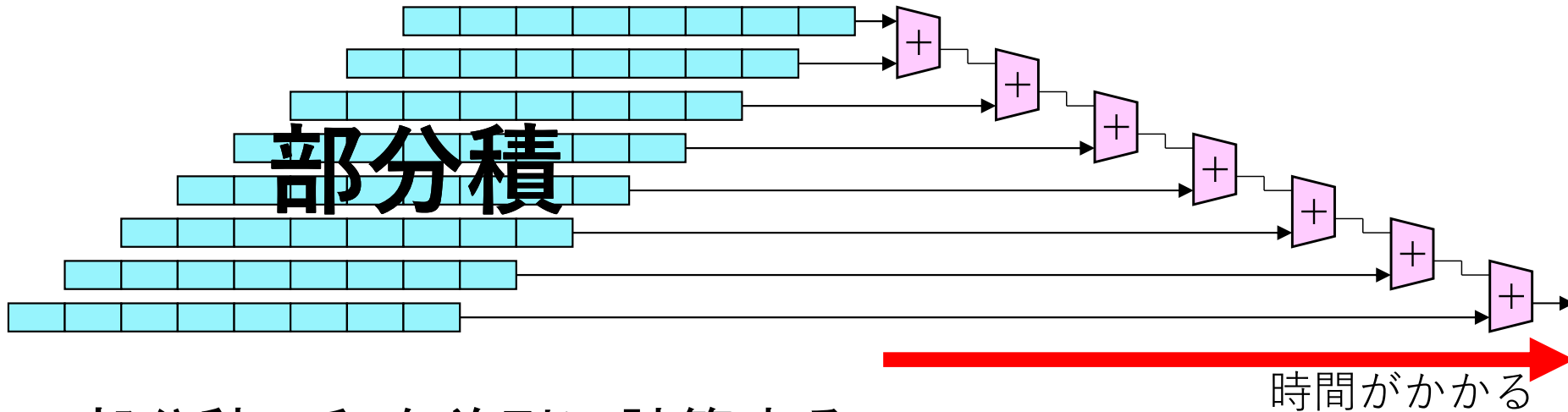
サイクル	ステップ	乗数	被乗数	積
0	初期値	0011	1110	0000 0000
1	1a:1→積=積+被乗数	0011	1110	①1110 0000
	2:積を右へシフト	0011	1110	0111 0000
	3: 乗数を右へシフト	0001	1110	0111 0000
2	1a:1→積=積+被乗数	0001	1110	①0101 0000
	2:積を右へシフト	0001	1110	1010 1000
	3: 乗数を右へシフト	0000	1110	1010 1000
3	1a:0→演算なし	0000	1110	1010 1000
	2:積を右へシフト	0000	1110	0101 0100
	3: 乗数を右へシフト	0000	1110	0101 0100
4	1a:0→演算なし	0000	1110	0101 0100
	2:積を右へシフト	0000	1110	0010 1010
	3: 乗数を右へシフト	0000	1110	0010 1010



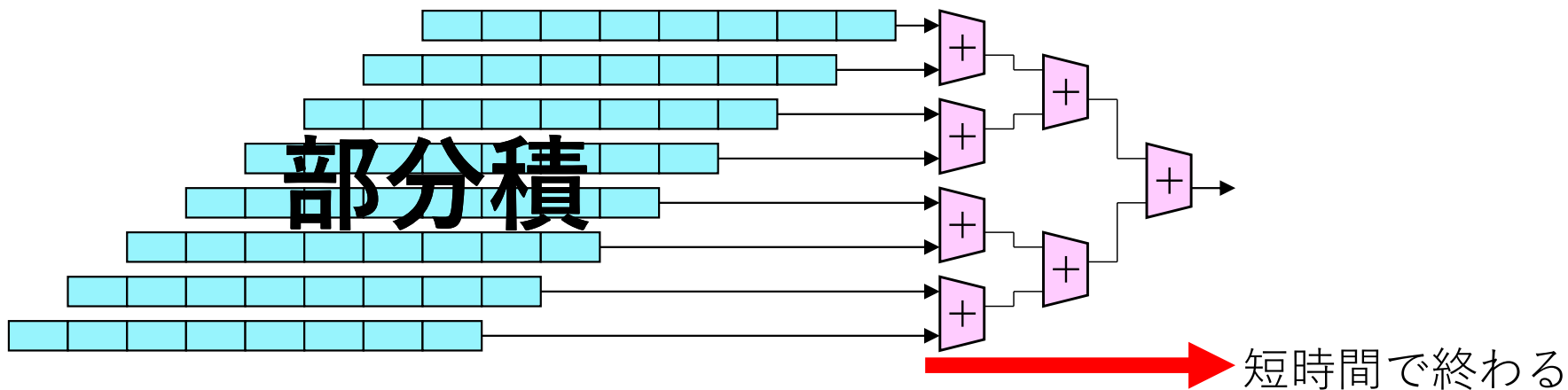
計算結果は  $00101010_2 = 42_{10}$

# 乗算の高速化

- 部分積の和を逐次的に計算する
  - 1個の加算器を使い回せば、回路規模は節約できる
  - $n$  bit 乗算に加算  $n$  回分の計算時間が必要（性能は悪い）



- 部分積の和を並列に計算する
  - 多数の加算器を並列動作させれば、短時間で計算が終わる（高性能）
  - 多数の加算器を実装するので、回路規模が大きくなる



32ビット×32ビット=□ビットの乗算結果(積)を保持するために、  
2つで1組の特別な32ビットレジスタ(Hi, Lo)がある。

## MIPS命令

`mult` : 符号あり乗算(multiply)の命令

`multu` : 符号なし乗算(multiply unsigned)の命令

## 乗算結果をレジスタHi, Loに取り出す命令

`mflo $1: move from lo` 命令

`mghi $1: move from hi` 命令

もし、常に32ビットで計算している場合、乗算の結果としても32ビットを期待するなら、□命令  
を用いれば計算結果を得られる。ただし、オーバフローには注意 (演習⑤参照)

# 演習問題 その⑤

教科書p182

32ビット×32ビットの乗算をして、結果が32ビットであるかどうかの判定はどうしたらいいか？ 符号付と符号なしの両方の場合を考えよ。

オーバフローが発生したら、OverFlowに飛ぶコードを、符号なしと符号あり、の2通りで以下に示す。自力で考えてください。ただし、少し難しいので、2ページ後にヒントスライドを用意しました。

## 符号なしの場合の解答

```
multu $s2, $s3
mfhi  $t0
bne  $t0, $zero, OverFlow
----- #オーバフローしなかった時の後続命令
```

OverFlow: ----- #オーバフローの処理ルーチン

## 演習問題 その⑤の回答の続き

簡単のため、

- \$t2に100...000(32ビット)

- \$t3に111...111(32ビット)

がすでに入っているとする。

これを利用して以下のコードで次ページの条件をチェックできる

```
mult    $s0, $s1
mflo    $t0
mfhi    $t1
sltu    $t4, $t0, $t2
bne     $t4, $zero, Zero
bne    $t1, $t3, OverFlow
j Cont
Zero: bne    $t1, $zero, OverFlow
Cont: -----    #オーバフローしなかった時の後続命令
      -----
Overflow:
```

どの部分が穴うめでも  
ちゃんと埋めれるように！

## 自習用の解説

結果の上位32ビット（Hi）と下位32ビット（Lo）がどういう時にオーバーフローが起こるのかを整理して考えましょう。

### 符号なしの場合

NG : オーバフロー

OK	000...000	101...010
NG	000...001	101...010

つまり、オーバーフローでないのは、

Hi がall 0 の時

### 符号ありの場合

OK	000...000	001...010
OK	111...111	100...000
NG	000...001	001...010
NG	101...001	101...010
NG	000...000	101...010
NG	111...111	000...000

つまり、オーバーフローの条件は、  
Loの符号ビット=1なら Hi ≠ all 1

Loの符号ビット=0なら Hi ≠ all 0



- 2進数の加算と減算（教科書3.2節, 付録(B.2, B.3, B.5)の一部 + $\alpha$ )
  - 実行の様子 **ポイント：最上位からの桁上げは無視してOK**
  - オーバフロー（オーバーフローというのが一般的かもしれないが教科書に合わせます）
  - ハードウェアの概要（加算器⇒ALU）
- 乗算の実現方法（教科書3.3節）
  - 基本的な乗算アルゴリズムとそのハードウェア
  - その改良
- 除算の実現方法（教科書3.4節）
  - 基本的な除算アルゴリズムとそのハードウェア
- 浮動小数点形式による実数の演算（教科書3.5節）
  - 仮数、指数、正規化
  - IEEE754浮動小数点規格
  - 演算精度と丸め
  - 浮動小数点演算の実現方法

2022年度は  
試験範囲外

# 整数型と固定小数点形式

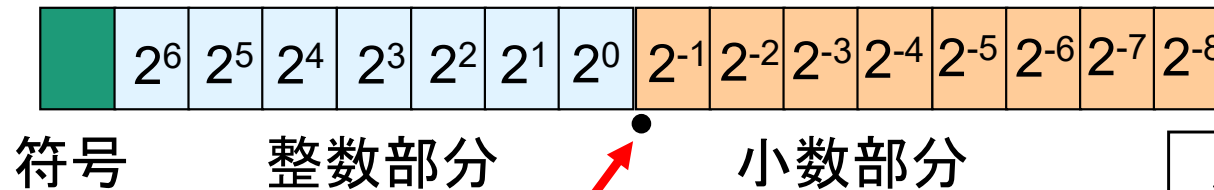
- 整数型では整数しか表せない
- 例えば「5.875」という実数を整数型で表すことはできないか？
  - A君の答え：「5.875」という値を1000倍した「5875」ならば、整数型で表現できます
- 計算機の中は数値を2進数で表しているので、10倍、100倍、1000倍という操作をするためには乗算という重い計算が必要
  - 従って、上記のように1000倍にスケールリングして実数を整数で近似するのは賢明ではない
  - 2倍、4倍、8倍という操作なら、シフト演算という軽い計算で実現できる
- $5.875_{10} = 2^2 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3}$ を2進数に変換すると  $101.111_2$  となるので、例えばこれを8倍にスケールリングした（つまり3ビット左シフトした） $101111_2$ を整数型で表現することが考えられる
- **2進数の小数を予め決められた（固定された）桁数シフトして整数型で表す方法**（言い方を変えると、**小数点以下のビット数を固定した2進小数で表す方法**）を**固定小数点形式**という

# 固定小数点形式

実数を $2^8=256$ 倍して整数で近似することに相当

## 固定小数点形式(2進数の場合)

- 例(符号1ビット、整数部分7ビット、小数部分8ビットの場合)



ここに小数点があると思えばよい

小数点の位置が固定

例) 44.125

(7ビット整数、8ビット小数)

00101100 00100000

符号 44

0.125

固定小数点形式の長所:

1. 整数演算用の演算器や命令がそのまま流用できる
2. 多くの場合、浮動小数点形式より演算が高速、低消費電力

固定小数点形式の短所

1. 浮動小数点形式に比べ、表現可能な値の範囲が狭い
2. 表現できる値の刻み幅が固定的で、絶対値の小さい値ほど相対誤差が大きい

ミニクイズ 上の例で

1. 表現可能な数の絶対値の最小は(0以外)?
2. 表現可能な数の絶対値の最大は?


# 科学記数法と浮動小数点形式

## 科学記数法(10進数の場合)

$$\underbrace{2.4888}_{\text{仮数部分}} \times \underbrace{10^{58}}_{\text{指数部分}}$$

化: 小数点の左側が1桁でかつ0でないようにする

ミニクイズ 以下の数を正規化せよ

1.  $0.2435 \times 10^{21} =$

2.  $24.35 \times 10^{21} =$

$$2.435 \times 10^0 = 2.435$$

$$2.435 \times 10^1 = 24.35$$

$$2.435 \times 10^2 = 243.5$$

$$2.435 \times 10^3 = 2435.$$

指数部の値により、小数点の位置が移動する

## 浮動小数点形式(2進数の場合)

$$X = m \times 2^e$$

部:

- 所定のビット数の2進数で表す
- 正規化する

部:

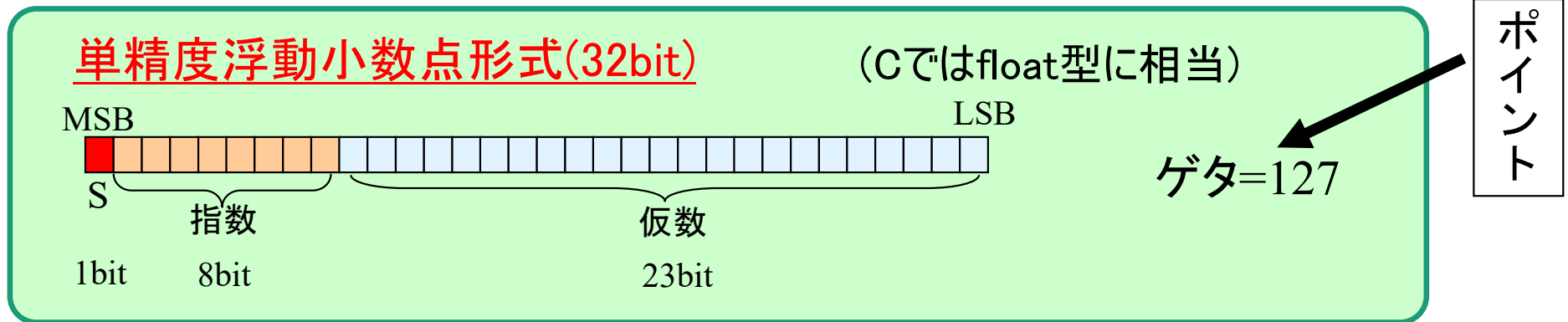
- 所定のビット数の2進数で表す

固定小数点に比べて、

1. 表現可能な値の範囲が広い
2. 表現できる数の刻み幅が変化する(表現する値の絶対値によらず、相対精度がほぼ一定)

# 浮動小数点形式の数値表現：単精度

IEEE754による表現:  $(-1)^S \times (1 + \text{仮数}) \times 2^{(\text{指数} - \text{ゲタ})}$



符号: Sは表現する数値の符号(1なら負)

仮数: 表現する数値を正規化すると仮数の先頭(整数部分)は必ず1になるので、それを省いて小数点以下23桁を格納する(省略された最上位桁を「隠しビット」、このように省略することを「ケチ表現」と言う)

指数: 2の補数表現のままでは,  $-1 = 11111111$ ,  $1 = 00000001$  であり, 最上位ビットで, 指数の大小比較が出来ない.  $\Rightarrow$  ゲタを履かせる.  
最も小さな負の指数を  $0000 \dots 0_2$  で表し,  
最も大きな正の指数を  $1111 \dots 1_2$  で表す.

# IEEE754形式の補足説明 1/2

**符号** 負なら1、正なら0で表す

**仮数** 正規化された2進数の浮動小数点表現では必ず1が最初にくるので、IEEE754形式では省略する

例  $\underbrace{1.01011}_{\text{仮数部分}} \times \underbrace{2^{10}}_{\text{指数部分}} \Rightarrow$  仮数部分には、01011を格納する

- 1.0 という仮数を表現する時は、仮数部分=0000...0
- では、0はどう表現するの？ → 例外として扱う（次ページ）

**指数** 指数の部分は2の補数でなく、以下のようなゲタばき表記を使用  
⇒通常の整数と同じルールで2つの数の比較が可  
（左のビットほど優先して単純に大小判定が可能）

00000000	→ 一番小さな指数	=	$0 - 127 = -127$
00000001	→	=	$1 - 127 = -126$
....			
11111110	→	=	$254 - 127 = +127$
11111111	→ 一番大きな指数	=	$255 - 127 = +128$

大小関係のルールは  
符号なしの8ビットの  
2進数と同様となる

※ 最小、最大の指数は特別用途に使うので、実際には使用しない（次ページ）72

# IEEE754形式の補足説明 2/2

IEEE754による表現:  $(-1)^S \times (1 + \text{仮数}) \times 2^{(\text{指数}-\text{ゲタ})}$

ただし、指数が0, および最大値の時は特別とする以下のルールを適用

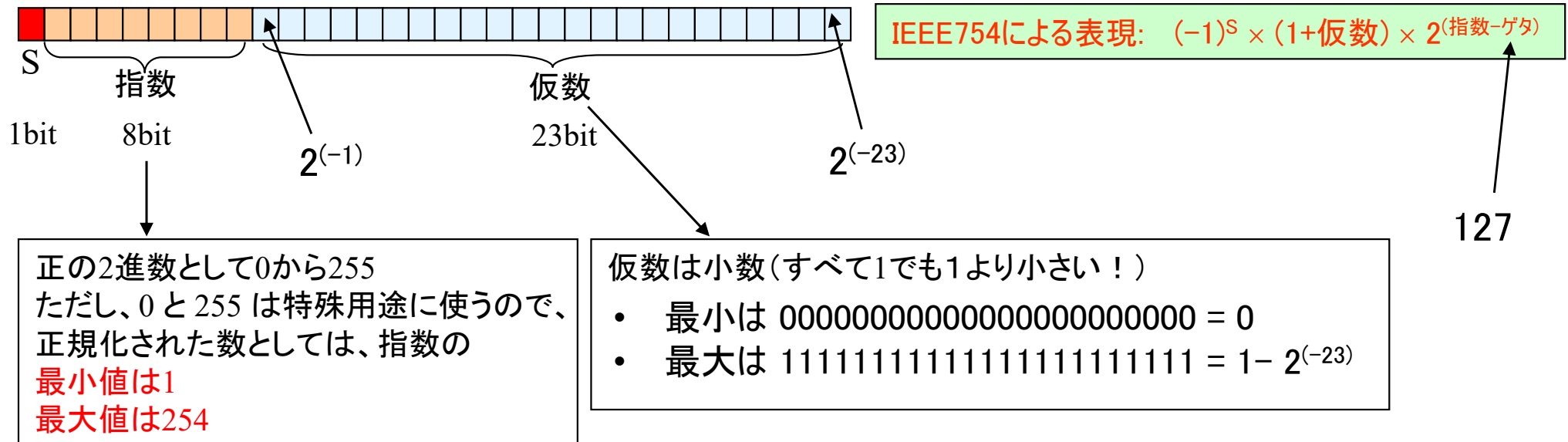
単精度		倍精度		内容	教科書図3.13
指数	仮数	指数	仮数		
0	0	0	0	$\pm 0$	※1: ゼロ
0	$\neq 0$	0	$\neq 0$	$\pm$ 非正規化数	※2: 絶対値が非常に小さい数
1~254	任意	1~2046	任意	$\pm$ 浮動小数点数	※3: 普通の数
255	0	2047	0	$\pm$ 無限大	※4: オーバーフロー、ゼロによる除算など
255	$\neq 0$	2047	$\neq 0$	NaN (非数)	※5: $\sqrt{-1}$ など

※1: ゼロは2種類存在する(+0 (S=0) と -0 (S=1))

※2: 非正規化数の時:  $(-1)^S \times (0 + \text{仮数}) \times 2^{(-126)}$

- 例えば  $(1.0101) \times 2^{(-127)}$  を単精度浮動小数点形式で表現する場合、ゲタ(127)を履かせても指数部は 0 となってしまう、浮動小数点数(※3)として表すことはできない
- この場合、 $(0.10101) \times 2^{(-126)}$  というように指数部を -126 にして非正規化数(※2)で表す

## 自己確認クイズ：正規化された正の浮動小数点数の最小値(0を除く)・最大値は？



最小の数は、最小の仮数と最小の指数の時で

最大の数は、最大の仮数と最大の指数の時で



## 自己確認クイズ: 非正規化数の場合はどうか？

・非正規化数の時:  $(-1)^S \times (0 + \text{仮数}) \times 2^{(-126)}$

(-126) という数にどうして  
なっているかわかりますね！

仮数は(表す数として0を考えないとする)

- 最小は  $000000000000000000000001 = 2^{(-23)}$
- 最大は  $111111111111111111111111 = 1 - 2^{(-23)}$



表現できる絶対値の

◆ 最小値は、

◆ 最大値は、

この間を、 $2^{(-149)}$ の固定の刻み幅で数表現

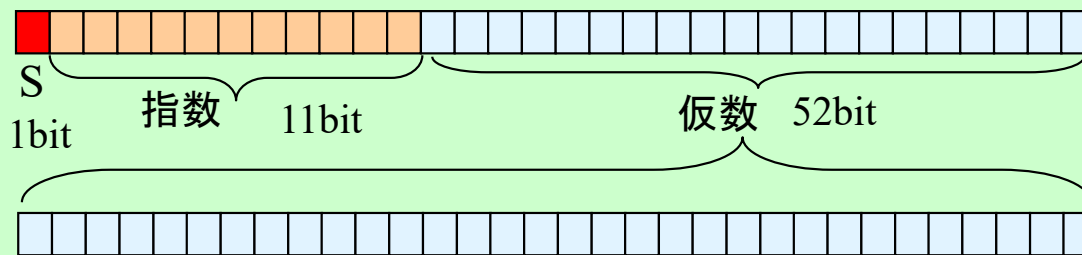
- ◆  $2^{(-149)}$ の刻み幅で最大値より1つ上の数は、 $2^{(-126)}$ となる。これが正規化された数の最小値なので、正規化された数で表せる数と、非正規化数で表せる数が切れ目なく続くことになる。逆にいうとそのように、この数(-126)が決められている。

# 浮動小数点形式の数値表現：倍精度

IEEE754による表現:  $(-1)^S \times (1 + \text{仮数}) \times 2^{(\text{指数} - \text{ゲタ})}$

倍精度浮動小数点形式(64bit)

(Cでは  型に相当)



ゲタ=1023

ゲタはIEEE754規格で定義されているが、「指数部のビット数を  $k$  とするとゲタは  $2^{k-1}-1$ 」と覚えるとよい

(半精度:  $k=5$ , ゲタ=15, 単精度:  $k=8$ , ゲタ=127, 倍精度:  $k=11$ , ゲタ=1023, 4倍精度:  $k=15$ , ゲタ=16383)

00000000000 → 一番小さな指数 =  $0 - 1023 = -1023$

00000000001 →  $= 1 - 1023 = -1022$

....

11111111110 →  $= 2046 - 1023 = 1023$

11111111111 → 一番大きな指数 =  $2047 - 1023 = 1024$

※ 最小、最大の指数は特別用途に使うので、実際には使用しない

# 演習問題 その⑦：各自やってください

問題①  $-0.75$ を単精度、倍精度(IEEE754)で表現せよ。この形式では、左から符号、指数、仮数となっており、仮数は単精度で23ビット、倍精度で52ビットとなっている。

答)  $-0.75 = -(1 \times 0.5 + 1 \times 0.25) = -1.1_2 \times 2^{-1}$  となるので、

単精度の場合 :  $S=1$ , 仮数 $=0.1_2$ , 指数 $=127-1=126$  より、

10111111 01000000 00000000 00000000

倍精度の場合 :  $S=1$ , 仮数 $=0.1_2$ , 指数 $=1023-1=1022$  より

10111111 11101000 00000000 00000000 00000000 00000000 00000000 00000000

問題② 単精度(IEEE754)で表現された

11000000 10100000 00000000 00000000 を10進数で表せ。

答) 単精度(IEEE754)として解釈すると :  $S=1$ , 仮数 $=0.01_2=0.25$ , 指数 $=129$

$$\begin{aligned} (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} &= -1.25 \times 4 \\ &= -5.0 \end{aligned}$$

\* このスライドでは、見やすさのため、8ビットごとに区切っている。また、指数の部分に下線を付けている。

# 演習問題 その⑧

(1) IEEE754単精度浮動小数点形式で

1. 表せる絶対値が最大の数は？

– P.74 より  $\pm(2 - 2^{-23}) \times 2^{127}$  (参考: およそ  $3.40 \times 10^{38}$  である)

2. 表せる絶対値が最小の数は？

– 正規化数の範囲では、P.74 より、 $\pm 2^{-126}$  である (参考: およそ  $1.18 \times 10^{-38}$  である)

– 非正規化数を含めると、P.75 より、 $\pm 2^{-149}$  である (参考: およそ  $1.40 \times 10^{-45}$  である)

(2) IEEE754倍精度浮動小数点形式で

1. 表せる絶対値が最大の数は？

P.74と同様にして、 $\pm(2 - 2^{-52}) \times 2^{1023}$  である

2. 表せる絶対値が最小の数は？

正規化数の範囲では、P.74 と同様にして、 $\pm 2^{-1022}$  である

非正規化数を含めると、P.75 と同様にして、 $\pm 2^{-1074}$  である

(3) 整数部分が8ビット、小数部分が23ビットの固定小数点形式を考えた場合  
の上記の値を求め、(1) と比較せよ

# 演習問題 その⑧(3) の解説

(3) 符号1ビット、整数8ビット、小数23ビットの固定小数点形式の場合はどうか？

P.69 と同様に考えよう。

- 絶対値が最小となるのは以下の2つの場合
  - 0 00000000 000000000000000000000001 ( $+2^{-23}$ )
  - 1 11111111 111111111111111111111111 ( $-2^{-23}$ )
- 絶対値が最大となるのは以下の場合
  - 1 00000000 000000000000000000000000 ( $-2^8$ )

これに対し、単精度浮動小数点形式の場合は(1)より、

- 絶対値最小の値は正規化数でも  $\pm 2^{-126}$ 、非正規化数も使えば  $\pm 2^{-149}$
- 絶対値最大の値はおよそ  $\pm 2^{128}$

以上から、同じビット数でも固定小数点形式に比べて、浮動小数点形式の方が表現できる

- 最大の数が桁違いに大きい
  - 最小の数が桁違いに小さい
- と言える

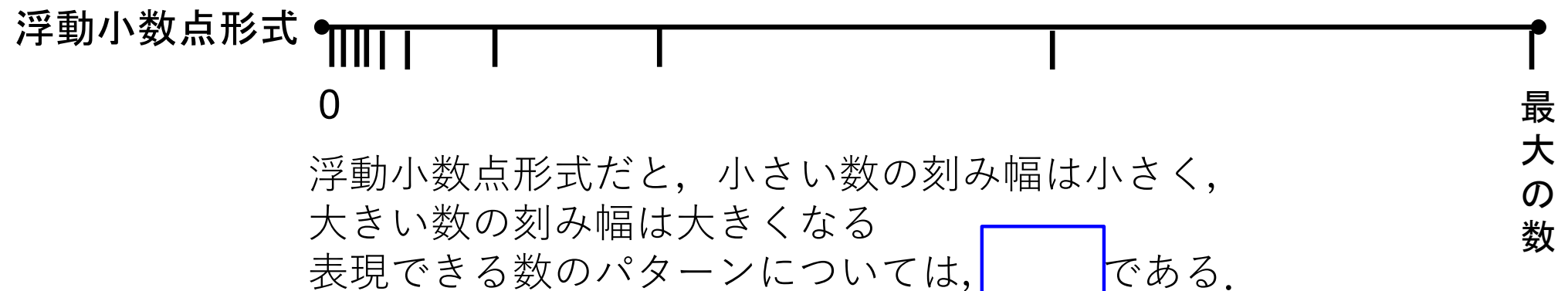
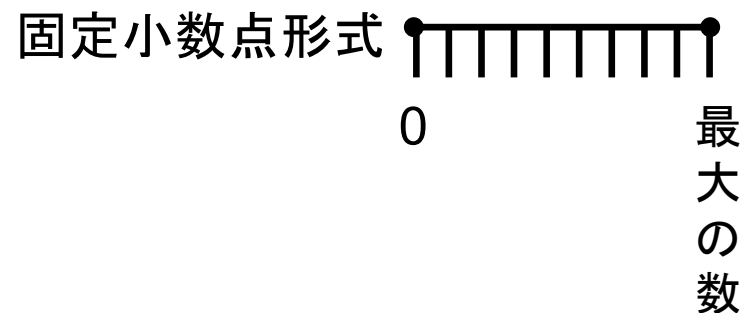
## 自己確認クイズ： 固定小数点と浮動小数点形式の比較

表現できる数のパターンは、

(a) 固定が圧倒的に多い、(b) 浮動が圧倒的に多い、(c) 大差はない、  
どれか？

また、表現できる隣接する2つの数の刻み幅を2つの形式で比較せよ。

比較のイメージ図（正のみ）



# 浮動小数点形式のオーバーフローとアンダーフロー

Overflow: 「表すべき値」 $>$ 「表現できる絶対値最大の正の数」または  
「表すべき値」 $<$ 「表現できる絶対値最大の負の数」のとき  
 $\Rightarrow +\infty$  あるいは  $-\infty$  で表されてしまう

Underflow: 「表すべき値の絶対値」 $<$ 「0以外で表現できる数の絶対値の  
最小値」のとき  
 $\Rightarrow +0$  あるいは  $-0$  で表されてしまう

## 演算の正確性：演習問題その⑨

IEEE754の倍精度で1と2の間に表現できる数の個数は？



(解説)

<input type="text"/>	= 1	仮数が <input type="text"/> の時
⋮		
<input type="text"/>	< 2	仮数が <input type="text"/> の時

\_\_\_\_\_

# 丸め

## 丸め

前ページで見たように、原理的に表現できる数の個数は決まっている。計算結果が表現できない数になれば、もっとも近い」表現できる数にする。

### 【余談】

例1:  $1 \div 11$  を10進数と2進数で表せ

- 10進数: 0.09090909... (割り切れず循環小数になる)
- 2進数: 0.000101110100010111010001011101... (割り切れず循環小数になる)

例2:  $1 \div 5$  を10進数と2進数で表せ

- 10進数: 0.2 (割り切れる)
- 2進数: 0.001100110011... (割り切れず循環小数になる)

浮動小数点の加算や乗算は後で見るように内部では、複数の演算を行う。その内部での演算の中間的な演算結果は、実際の精度よりも2桁余分に保持しておき、最終結果は、使えるビットに適合するように丸めるのがIEEEの方式である。

この時の、2桁を上位から、ガード桁、丸め桁と呼ぶ

\* 次ページの問題を参照

# 演習問題 その⑩

教科書p211(少し変更)

仮数を3桁として、 $1.23_{10} \times 10^1 + 2.33_{10} \times 10^2 + 1.21_{10} \times 10^0$  の計算せよ。  
中間結果として2桁余分に保持できる場合とできない場合の両方を考えよ。


(答え)

指数をそろえる時に、

途中は有効桁5桁の場合

$$\begin{array}{r} 0.1230_{10} \times 10^2 \\ 2.3300_{10} \times 10^2 \\ + 0.0121_{10} \times 10^2 \\ \hline 2.4651_{10} \times 10^2 \end{array}$$

最終的には有効  
桁3桁に丸める


$$2.47_{10} \times 10^2$$

途中も有効桁3桁の場合

$$\begin{array}{r} 0.1230_{10} \times 10^2 \\ 2.3300_{10} \times 10^2 \\ + 0.0121_{10} \times 10^2 \\ \hline 2.4651_{10} \times 10^2 \end{array}$$

$$2.46_{10} \times 10^2$$

\* これは、本当に正しい答えを  
3桁に丸めた結果とは異なる

\* これは、ガード桁、丸め桁について重要性を理解するためだけの問題である。  
また、簡単のために10進数で議論している。

# 演習問題 その⑪:演算の順序の重要性

教科書3.9節

単精度の以下の数値の計算結果はどうなるか？

$$x = -1.5_{10} \times 10^{38} \quad y = 1.5_{10} \times 10^{38} \quad z = 1$$

(1)  $x + (y + z)$



(2)  $(x + y) + z$



- $(y + z)$  は有効桁23ビットでは  $y$  となってしまうのが問題
- これはどうしようもない問題だが、上のように演算順序の工夫でうまく克服できる場合もある。

浮動小数点加算には、結合法則が成り立たない！！

- 2進数の加算と減算（教科書3.2節, 付録(B.2, B.3, B.5)の一部 + $\alpha$ )
  - 実行の様子 **ポイント：最上位からの桁上げは無視してOK**
  - オーバフロー（オーバーフローというのが一般的かもしれないが教科書に合わせます）
  - ハードウェアの概要（加算器⇒ALU）
- 乗算の実現方法（教科書3.3節）
  - 基本的な乗算アルゴリズムとそのハードウェア
  - その改良
- 除算の実現方法（教科書3.4節）
  - 基本的な除算アルゴリズムとそのハードウェア
- 浮動小数点形式による実数の演算（教科書3.5節）
  - 仮数、指数、正規化
  - IEEE754浮動小数点規格
  - 演算精度と丸め
  - **浮動小数点演算の実現方法**

2022年度は  
試験範囲外

例)  $9.999_{10} \times 10^1 + 1.610_{10} \times 10^{-1}$  (仮数を4桁、指数を2桁と仮定した10進の例)

## ステップ1) 小数点の位置合わせ

指数が大きい方に合わせる  
• 仮数の右シフト + 丸め

$$\begin{aligned} 9.999_{10} \times 10^1 &\Rightarrow 9.999_{10} \times 10^1 \\ 1.610_{10} \times 10^{-1} &\Rightarrow 0.01610_{10} \times 10^1 \end{aligned}$$

## ステップ2) 仮数の加算

$$9.999_{10} \times 10^1 + 0.01610_{10} \times 10^1 = 10.01510 \times 10^1$$

## ステップ3) 値の正規化 • シフトして指数の調整 (+オーバーフロー、アンダフローのチェック)

$$10.01510 \times 10^1 = 1.001510 \times 10^2$$

## ステップ4) 仮数の有効桁あわせ

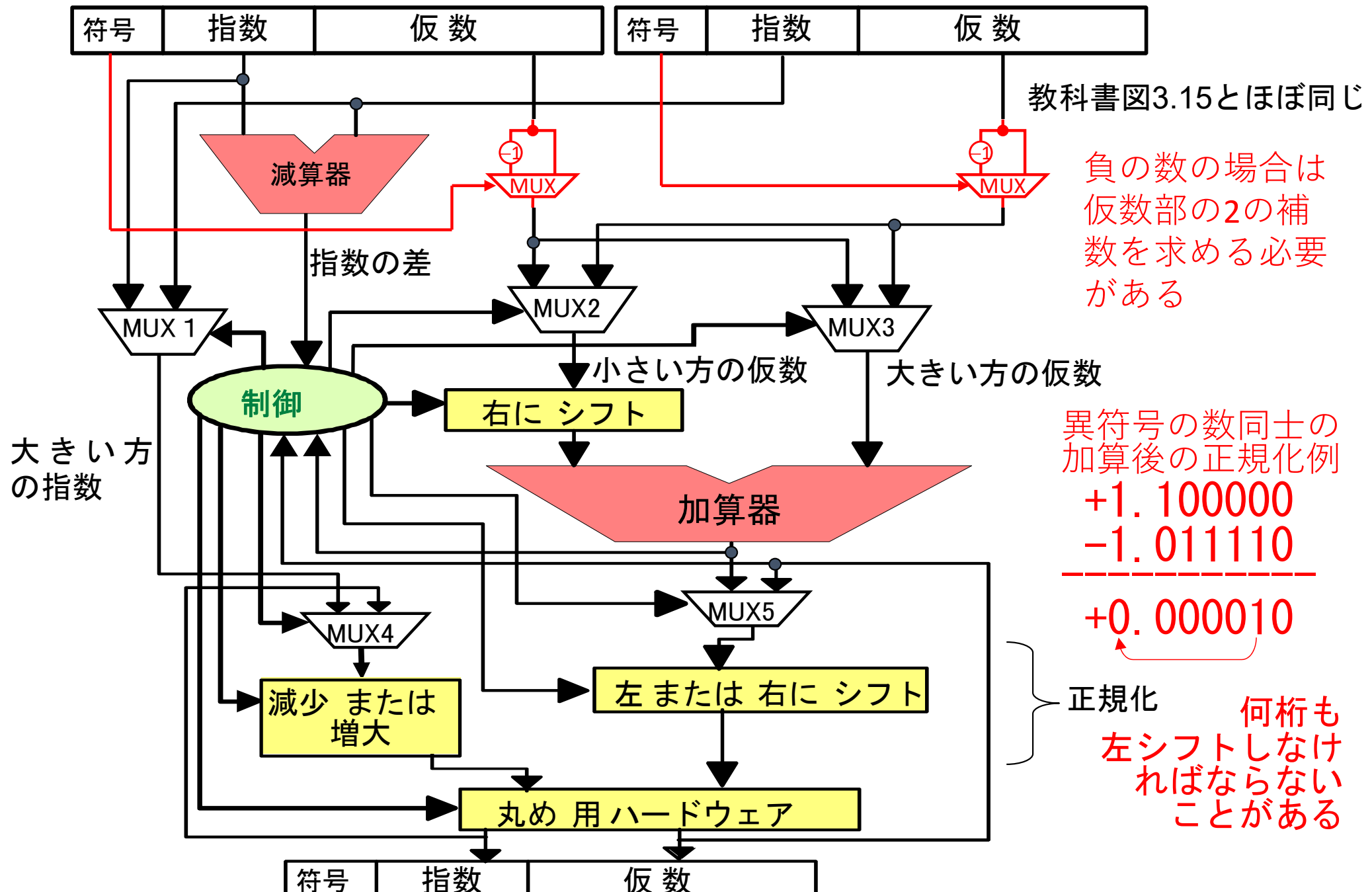
丸め = 四捨五入

$$1.001510 \times 10^2 \Rightarrow 1.002 \times 10^2$$

(丸め)

丸めで繰り上がりが生じた場合はステップ3へ e.g.,  $9.9995 \times 10^1 \Rightarrow 10.000 \times 10^1$

英語の問題 : e.g. の意味は



# 浮動小数点乗算

教科書p230～

- 加算と同様、人の計算と同じように行う:
  1. 指数の計算 (ゲタに注意)
  2. 仮数の計算
  3. 正規化
  4. 丸め (加算と同様、再度正規化を必要とする場合あり)
  5. 符号の決定

$-9.999_{10} \times 10^2 \times -1.610_{10} \times 10^3$  (仮数を4桁、指数を2桁と仮定した10進の例)

$\Rightarrow 9.999_{10} \times 1.610_{10} \times 10^{2+3}$

$\Rightarrow 16.09839 \times 10^5$  (正規化、丸め)  $\Rightarrow 1.610 \times 10^6$  (最後に符号の決定)



乗算で、前ページの10進数の例とIEEE754形式の場合で違う点は？  
(ゲタの扱い方が違う)

$0.5_{10} \times -0.4375_{10}$  を2進数で計算せよ.

2進数では、 $(1.000_2 \times 2^{-1}) \times (1.110_2 \times 2^{-2})$  (簡単のため、仮数4ビットとする.)

1. 指数の計算 (ゲタに注意)

ゲタを考慮しないと、 $(-1)+(-2)=-3$  だが、ゲタを考慮すると  
 $(-1 + 127) + (-2 + 127) - 127 = -3 + 127 = 124$

2. 仮数の計算

$1.000_2 \times 1.110_2 = 1.110000_2$   
4ビットで表すと  $1.110_2 \times 2^{-3}$

3. 正規化

正規化されているかチェックし、オーバフロー、  
アンダフローが発生しているかチェック

4. 丸め (加算と同様、再度正規化を必要とする場合あり) この例では不要

5. 符号の決定 2つの数の符号が違うので、結果の符号は負とする.

(答え)  $-1.110_2 \times 2^{-3}$

# Lec. 8 の要チェック用語集

全加算器

RCA

ALU

オーバフロー

例外

部分積

引き戻し法

浮動小数点形式

固定小数点形式

仮数

指数

正規化

単精度

倍精度

丸め

意味とか何の略かぐらいはチェックすべし

“ハードウェアアルゴリズム”を理解しよう！

## IEEE 754 Format (単精度と倍精度)



余力がある人は

もっと高速なハードウェアアルゴリズムがあります！

例えば、以下を自分で頑張って勉強してみてください。がんばって理解できればLevel Up!

1. 桁上げ先見加算器(付録C-6)
2. 桁上げ保存加算器による乗算(付録Cの演習C.31)

## 演習問題 その⑬

浮動小数点数  $20_{10}$  のIEEE754規格の2進数表現を単精度と倍精度で示せ。

## 演習問題 その⑬ 略解

$20_{10} = 10100_2 = 1.0100_2 \times 2^4$  なので、

4ビット右にシフトして正規化するので、 $\times 2^4$ する。

- 符号ビットは0
- 仮数の最上位は、.010
- 単精度の指数部は、 $= 4 + 127 = 131$
- 倍精度の指数部は、 $= 4 + 1023 = 1027$

以上より、

単精度

01000001101000000000000000000000

赤字が指数

倍精度

01000000001101000000000000000000  
00000000000000000000000000000000

## 演習問題 その⑭

IEEE754単精度表現の下記の浮動小数点がある。

x: 0101 1111 1011 1110 0100 0000 0000 0000

y: 0011 1111 1111 1000 0000 0000 0000 0000

z: 1101 1111 1011 1110 0100 0000 0000 0000

- ①  $x+y$  の結果を示せ
- ② 上記の結果に $z$ を足すとどうなるか？
- ③  $x+y+z$  の正しい結果を得るためには、どのように加算を行えばいいか？

# 演習問題 その⑭ 略解

x: 0101 1111 1011 1110 0100 0000 0000 0000

y: 0011 1111 1111 1000 0000 0000 0000 0000

①まず指数(上の赤字)の差を計算

10111111

-01111111

01000000 --> 64

次に講義でしたとおりの手順で、yの仮数を $1/2^{64}$ （つまり64ビットシフト）して指数をそろえてから仮数同士を足し算。しかし、この有効桁数では、yの仮数が小さすぎて0になってしまう。

1. 011 1110 0100 0000 0000 0000 (xの仮数:最上位に暗黙で足す1を含む)

+0.000 0000 0000 0000 0000 0000 (yの仮数: 暗黙の1を含んだものを $1/2^{64}$ 倍 (64ビット右シフト))

1. 011 1110 0100 0000 0000 0000 すると0になる)

この場合は丸めなどは必要ないので、x+yの結果を浮動小数点形式に戻すと

0101 1111 1011 1110 0100 0000 0000 0000 でこれはxと同じ

②zは-xなので、aの結果にzを足した結果は明らかに0となる。(手順省略)

③z=-xなので、x+y+zをすれば直感的にはyとなって欲しいが、上で見たように0になってしまう。

\* ちなみに、x+z+yと足す順序を変えればちゃんと正しい答えが得られる

ポイント