# マイクロプロセッサ演習

2004 年度

第 8 回

## 1 ALU

### 1.1 ALU を構成する基本要素

今まで学んだ様に、MIPS CPU は 32 ビットの数値に対して算術演算 (加算・減算) ことが行うことができ、さらに論理演算 (AND、OR、シフト) を行なうことができる (算術演算、論理演算の詳細ついては付録 A 参照)。これをハードウェアとして実現するのが**算術論理演算ユニット** (Arithmetic Logic Unit :ALU) である。

ALU を構成する基本要素は図 1 の 4 つである。以下、ALU の実現方法を見てゆくが、はじめに加算器の実現について取り扱う。

### 1.2 [問題] 加算器

まず 1 ビットの加算 (半加算、全加算) について触れ、それを 32 ビットに拡張することを考えよう。

[**1 ビットの加算 (半加算、全加算)**]

論理回路などの授業で学んだように、加算を行なうには下位ビットから順に 1 ビットずつ加算を行なってゆけば良い (簡単な復習が 付録 A にある)。

その際、各ビットの加算には「和」と上位ビットへの「桁上がり (carry out)」の 2 つの出力があることを思い出そう。さらに、最下位ビット以外のビットの加算には、下位ビットからの「桁上がり (carry in)」も考慮にいれなければならないのであった。

以上を考慮すると、 1 ビットの加算を行なうには、下位ビットからの桁上がりを考慮しない「半加算器」と下位ビットからの桁上がりを考慮する「全加算器」が考えられる (図 2)。

$n$ ビットの加算を行なう際、最下位ビットの加算は半加算器で、それ以外のビットの加算は全加算器で実現できることに注意しよう。

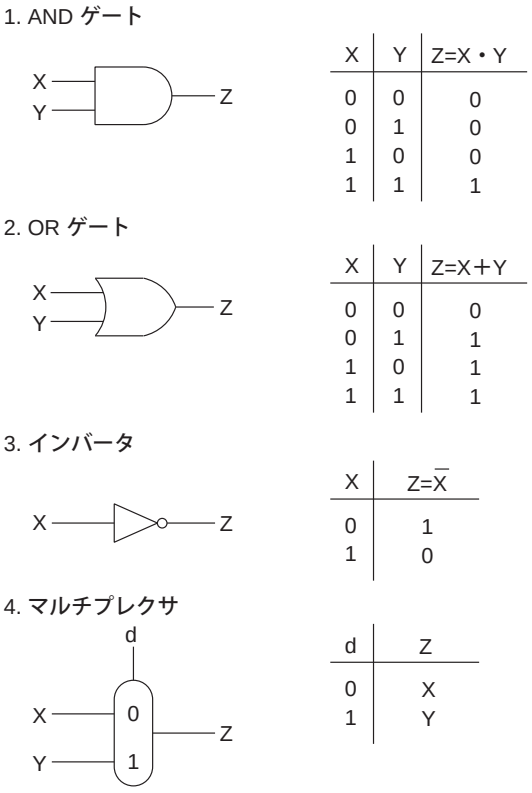


図 1: ALU を構成する 4 つの基本要素とその真理値表。

[**問題**] (論理回路の復習)

1. 半加算器の真理値表を書け。
2. 書いた真理値表をもとに、半加算器の $S$ と $C$ の論理式を書いてみよ。
   (**注意**) 論理式とは、出力 $S$ や $C$ を入力 $X$、$Y$ および記号「$\cdot$ (かつ)」、「$+$ (または)」、「$^-$ (否定)」で表したものである。ここでの「$\cdot$」、「$+$」はブール代数の演算であり、算術演算の記号と混同しないようにしよう。
3. 論理式をもとに、半加算器を図 1 の部品を用いて構成してみよ。

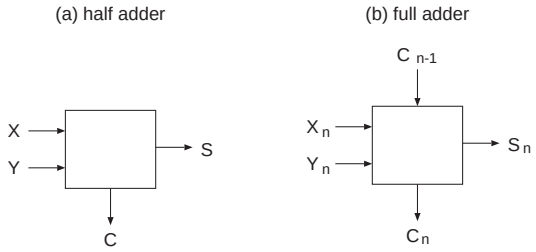


図 2: 半加算器と全加算器の模式図。(a) 半加算器。$X$,$Y$: 入力、$S$: 和、$C$: 上位ビットへの桁上がり。(b) 全加算器。$X_n$, $Y_n$: 入力、$C_{n-1}$: 下位ビットからの桁上がり、$S_n$: 和、$C_n$: 上位ビットへの桁上がり。

全加算器は半加算器 2 つと OR ゲートを用いて構成できる。余力があれば考えてみてほしい。

[$n$ **ビットの加算器への拡張**]

前節で触れたように、$n$ ビットの加算器は 1 ビットの加算器 (半加算器や全加算器) を組み合わせることで構成できる。

[**問題**] 図 2 の半加算器と全加算器の模式図を用いて、$n$ ビットの加算器を実現してみよ。(ここで構成した $n$ ビット加算器は ripple carry adder と呼ばれる)

## 1.3 [問題] ALU の構成

以上の準備のもと、ALU を構成をする。

加算器のときと同様に「1 ビットの ALU」を構成し、それを組み合わせることで「$n$ ビットの ALU」が実現できる。

[**AND、OR、加算を行なう ALU**]

まず、AND、OR、加算を行なう 1 ビットの ALU は図 3 のように構成される。

操作入力 $d$ の値によって、出力 $Z_n$ の値が AND、OR、加算のいずれかに選択される。全加算器のためにキャリーイン $C_{n-1}$ およびキャリーアウト $C_n$ が存在することにも注意しよう。

この 1 ビット ALU を 32 個組み合わせれば、32 ビット ALU が構成される (図 4)。

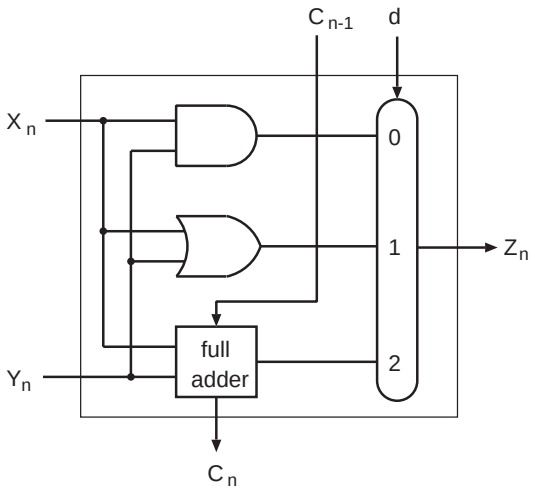


図 3: AND、OR、(全) 加算を含む 1 ビット ALU。入力 $X_n$、$Y_n$、出力 $Z_n$ の他に、全加算器のためのキャリーイン $C_{n-1}$ および キャリーアウト $C_n$、結果選択のための操作入力 $d$ を含む。

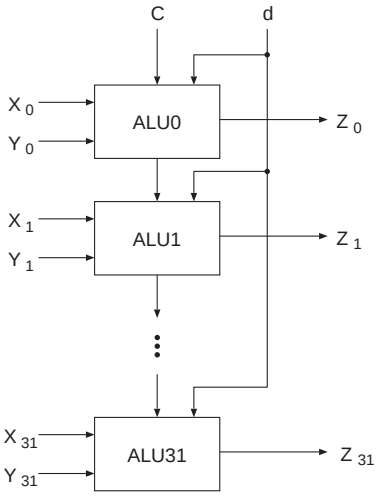


図 4: 32 ビット ALU。

加算器の時には、最下位ビット (図 4 では $X_0$ と $Y_0$) の計算にはキャリーインを考慮する必要はなかったが、ここではキャリーイン $C$ が存在する。それは減算を行なう ALU を構成したいためである。それを次節でみよう。

[**減算のための ALU の拡張**]

32 ビットの符号付き数 $X = X_{31}X_{30}\cdots X_1X_0$ と $Y = Y_{31}Y_{30}\cdots Y_1Y_0$ の減算 $X-Y$ は、$X+(-Y)$ とすることで加算にすることができた。ただし、$(-Y)$ は $Y$ の符号を反転することで得られる。符号反転は「ビット反転をして 1 を加える」ことで実現される (付録 A.2 参照)。

$Y_n$ のビット反転を行なえるようにした 1 ビット ALU は図 5 である。

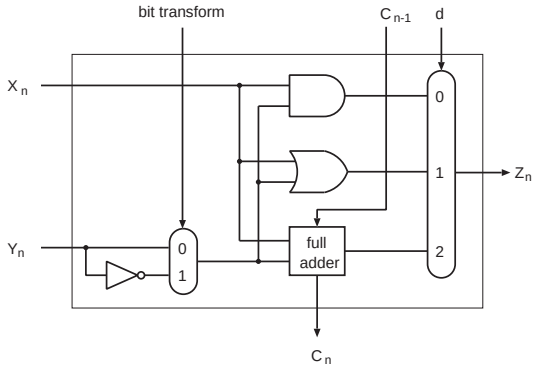


図 5: $Y_n$ のビット反転を行なえるようにした 1 ビット ALU。ビット反転を行なうかどうかを決める操作入力も加わっている。

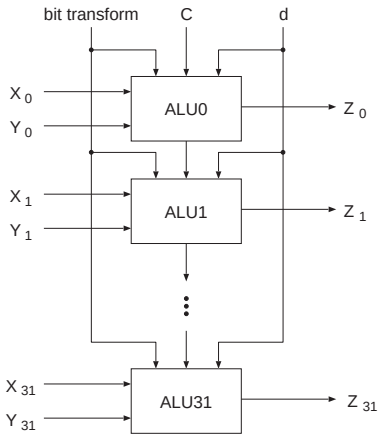


図 6: ビット反転を考慮した 32 ビットの ALU。

また、この ALU を 32 個繋げたのが図 6 である。

[**問題**]

1. 図 6 の 32 ビット ALU において、ビット反転入力 (bit transform) が 0、1 のとき、それぞれ加算、減算を行なわせるようにしたい。そのためには、最下位ビットへのキャリーイン ($C$) に何を入力したらよいか。「加算のとき $C = \cdots$、減算のとき $C = \cdots$」のように答えよ。また、何故それで減算が実現できるか。

### 1.4 ALU: おわりに

以上において、AND、OR、加算、減算を行なえる 32 ビット ALU の構成を見た。

実際の MIPS CPU では、以下のように ALU にさらに拡張が加えられている。

- 大小判定 slt 命令の追加。
- 最上位ビットの ALU におけるオーバーフロー検出機能の追加。
- 結果が 0 であるかどうかの判定機能追加。
- 高速桁上げ機能。(上で見た 32 ビット ALU は、いずれも桁上がりを 32 ビット順々に処理しなければならないため、時間がかかる。そこで、桁上がりを先に計算する**桁上がり先見加算器**が実際には用いられている)

これらは演習では取り扱わないので、授業の資料や教科書を参照して欲しい。

## 2 乗算と除算

### 2.1 [問題] 符号なし数の乗算

簡単のために符号なし数の乗算を考えよう。$0010\times0011$ の計算を行なうには、10 進数のときと同様に筆算で行なうことができる (図 7)。

一般に $n$ ビットの 2 進数と $m$ ビットの 2 進数を掛け合わせると、$n+m$ ビットになる (図 7 では 4 ビット $\times$ 4 ビットが 7 ビットになっているが、最上位ビットから桁上がりが生ずることも考えられ、その場合は 8 ビットになるのである)。

以上の筆算をアルゴリズムとしてまとめると図 8 のようになる (教科書、授業での「第一のバージョ

```
    0010
  × 0011
------
    0010
   0010
  0000
 0000
------
 0000110
```

図 7: 筆算による乗算の実行。

ン」に相当)。これはもっとも単純な乗算の実装の例である。また、このアルゴリズムを $0010 \times 0011$ に適用したのが図 9 である。

MIPS CPU にこのアルゴリズムを実装するには

- 64 ビット ALU
- 乗数用 32 ビットレジスタ
- 被乗数用 64 ビットレジスタ
- 積用 64 ビットレジスタ

が必要になる。実際にはアルゴリズムに工夫が加えられ、

- 32 ビット ALU
- 被乗数用 32 ビットレジスタ
- 積用 64 ビットレジスタ (乗数と共用)

のように、乗算ハードウェアはより簡単に実現されている (教科書、授業における「最終バージョン」)。そのアルゴリズムの詳細は教科書や授業を参照して欲しいが、基本は上で見た筆算によるアルゴリズムである。

[**問題**]

1. 図 9 の空欄 (a), (b), (c) は、流れ図 8 において何に対応しているか。

## 2.2 [問題] 符号付き数の乗算

上で取り扱ったアルゴリズムは、正の数のみを取り扱った。符号つきの数を取り扱うもっとも素朴な方法は、全て正の数に変換して乗算を行ない、必要に応じて結果の正負を反転するという方法である。

[**問題**]

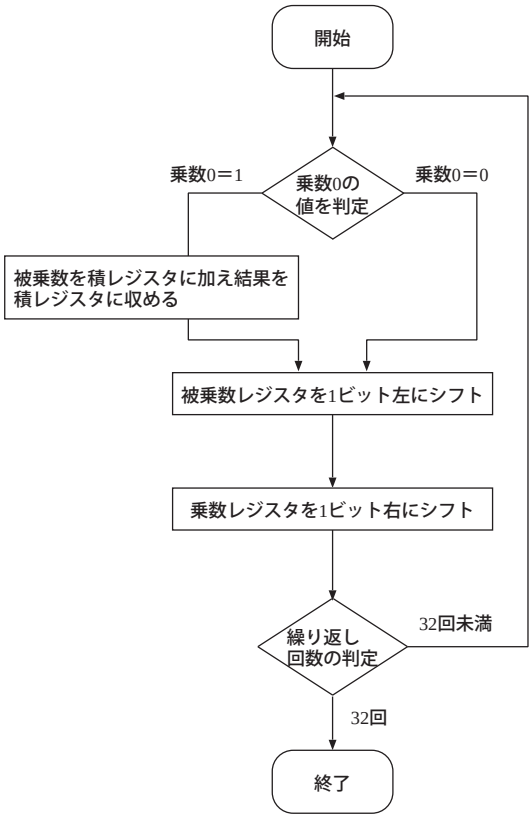


図 8: 乗算の第一のアルゴリズムの流れ図。

1. $0011 \times 1110$ の計算を、一旦正の数に変換する方法で行なえ (答えは 8 ビットで求めよ)。

2. $1110 \times 1110$ の計算を、一旦正の数に変換する方法で行なえ (答えは 8 ビットで求めよ)。

## 2.3 [問題] 符号なし数の除算

やはり、正の数のみによる除算を考える。この場合も筆算による計算が有効である。

図 10 は $0111 \div 0010$ $(7 \div 2)$ を計算を筆算で行い、商 11 と 剰余 1 を得ている。

このアルゴリズムをハードウェア実装するために流れ図で書くと、図 11 のようになる (教科書や授業の「第一のアルゴリズム」である)。

さらに、このアルゴリズムを $0111 \div 0010$ に適用すると図 12 のようになる。初期状態において、除数は 8 ビットの内上位 4 ビットにセットされ、被除数は剰余の下位 4 ビットに収められていることに注意しよう。

[**問題**]

| サイクル | ステップ | 乗数 | 被乗数 | 積 |
|---|---|---|---|---|
| 0 | 初期値 | 0011 | 0000 0010 | 0000 0000 |
| 1 | (a) | 0011 | 0000 0010 | 0000 0010 |
| | (b) | 0011 | 0000 0100 | 0000 0010 |
| | (c) | 0001 | 0000 0100 | 0000 0010 |
| 2 | (a) | 0001 | 0000 0100 | 0000 0110 |
| | (b) | 0001 | 0000 1000 | 0000 0110 |
| | (c) | 0000 | 0000 1000 | 0000 0110 |
| 3 | (a) | 0000 | 0000 1000 | 0000 0110 |
| | (b) | 0000 | 0001 0000 | 0000 0110 |
| | (c) | 0000 | 0001 0000 | 0000 0110 |
| 4 | (a) | 0000 | 0001 0000 | 0000 0110 |
| | (b) | 0000 | 0010 0000 | 0000 0110 |
| | (c) | 0000 | 0010 0000 | 0000 0110 |

図 9: 第一の乗算アルゴリズムを 4 ビットの乗算に適用した様子。

```
          11
      ------
0010 ) 0111
       10
      ----
       0011
       0010
      ----
          1
```

図 10: 筆算による除算の実行。

1. 図 12 における (a)、(b)、(c) は図 11 の流れ図の何に対応しているか。

## 2.4 符号付き数の除算

符号付き数の除算の場合は商と剰余のそれぞれに符号を適用しなければならないことに注意する。

以下の規則を適用する。

- $+7 \div +2 = +3$ 剰余 $+1$
- $-7 \div +2 = -3$ 剰余 $-1$
- $+7 \div -2 = -3$ 剰余 $+1$
- $-7 \div -2 = +3$ 剰余 $-1$

被除数の符号と剰余の符号が同じであること、どの場合も商の絶対値が等しいことに注意しよう。

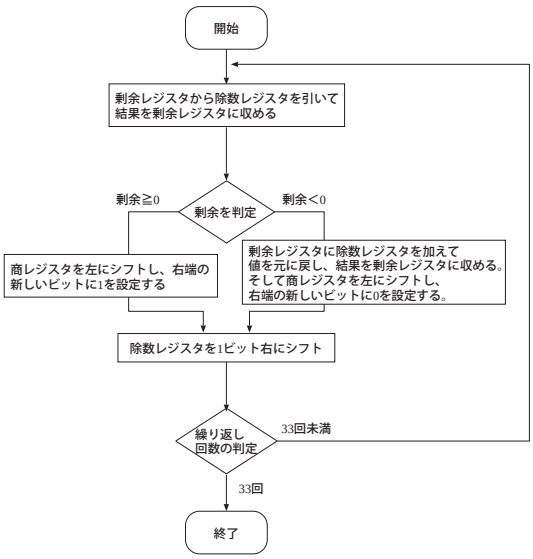


図 11: 第一の除算アルゴリズムの実行の流れ図

## 2.5 [SPIM] MIPS における乗算と除算

上でみたように乗算および除算には対象となる数の倍の長さを持つレジスタが必要となる (MIPS なら、64 ビットレジスタが必要)。

この 64 ビットレジスタは、(工夫された) アルゴリズムでは以下のように使われる。

- 乗算
  - 32 ビットレジスタ：被乗数
  - 64 ビットレジスタ：乗数・積 共用
- 除算
  - 32 ビットレジスタ：除数
  - 64 ビットレジスタ：被除数・商・剰余 共通

これらのレジスタは具体的には図 13 のように使用される。

さて、上記の 64 ビットレジスタであるが、MIPS では特別な 32 ビットレジスタ Hi と Lo が用意されていて、それぞれ 64 ビットの上位 32 ビット、 下位 32 ビットを表している。

これらを用いる MIPS 命令には、例えば以下のものがある。

- `mult $t0, $t1` ： 符号付き数 $t0, $t1 の乗算を行ない結果を Hi、Lo に収める。
- `multu $t0, $t1` ： 符号なし数 $t0, $t1 の乗算を行ない結果を Hi、Lo に収める。

| サイクル | ステップ | 商 | 除数 | 剰余 |
|---|---|---|---|---|
| 0 | 初期値 | 0000 | 0010 0000 | 0000 0111 |
| 1 | (a) | 0000 | 0010 0000 | 1110 0111 |
| | (b) | 0000 | 0010 0000 | 0000 0111 |
| | (c) | 0000 | 0001 0000 | 0000 0111 |
| 2 | (a) | 0000 | 0001 0000 | 1111 0111 |
| | (b) | 0000 | 0001 0000 | 0000 0111 |
| | (c) | 0000 | 0000 1000 | 0000 0111 |
| 3 | (a) | 0000 | 0000 1000 | 1111 1111 |
| | (b) | 0000 | 0000 1000 | 0000 0111 |
| | (c) | 0000 | 0000 0100 | 0000 0111 |
| 4 | (a) | 0000 | 0000 0100 | 0000 0011 |
| | (b) | 0001 | 0000 0100 | 0000 0011 |
| | (c) | 0001 | 0000 0010 | 0000 0011 |
| 5 | (a) | 0001 | 0000 0010 | 0000 0001 |
| | (b) | 0011 | 0000 0010 | 0000 0001 |
| | (c) | 0011 | 0000 0001 | 0000 0001 |

図 12: 第一の除算アルゴリズムを 4 ビットの除算に適用した様子。

- `div $t0, $t1` ： 符号付き数 $t0, $t1 の除算を行ない結果を Hi、Lo に収める。
- `divu $t0, $t1` ： 符号なし数 $t0, $t1 の除算を行ない結果を Hi、Lo に収める。

[**問題**]

1. 演習用の Web ページより、data08.zip をダウンロード。配布ファイルは MulDiv.asm。
2. MulDiv.asm の中身の指示に従い、以下の計算を実行させてみよ。
   「$7\times 2$」、「$7\times(-2)$」、「$7\div 2$」、「$7\div(-2)$」等など。なお、結果は Windows 版 PCspim の上部のウィンドウで Hi レジスタと Lo レジスタを参照することで確認せよ。

# 3 浮動小数点表現

## 3.1 [問題] 浮動小数点表現

以上において、整数に対する四則演算のアルゴリズムを説明してきた。しかし、コンピュータで扱う数値の形式は整数だけではなく、小数もあり得る。

それらは浮動小数点形式の数値として実現されている。浮動小数点形式の数値とは、10 進数では

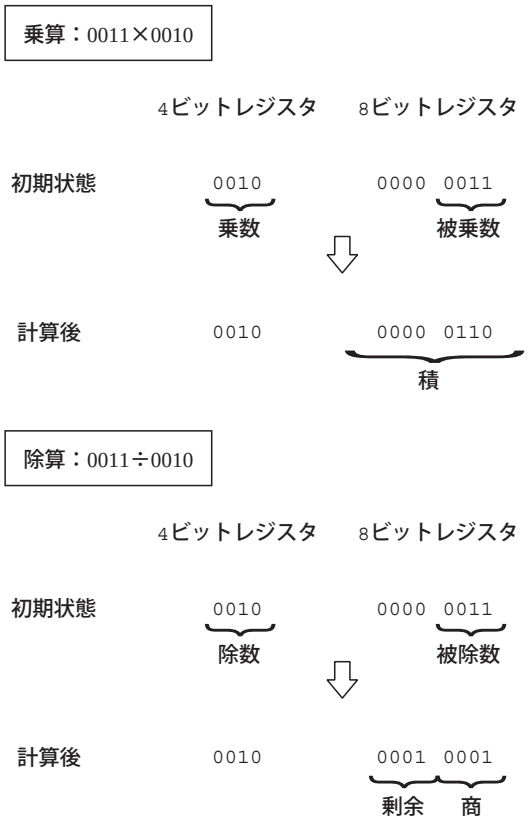


図 13: MIPS における乗算と除算。実際にはもちろん、「4ビットレジスタ」→「32ビットレジスタ」、「8ビットレジスタ」→「64 ビットレジスタ」である。

$-2.5\times 10^5$ のような数である。– は**符号**、2.5 の部分は**仮数**、5 の部分は **指数**と呼ばれる。

実際にはコンピュータ上では数値は 2 進数で表現されるので、浮動小数点形式の数値は以下の形をしている。

$$(+\text{or}-)1.xxxxx\times 2^{yyy} \qquad (1)$$

なお、仮数は小数点の左のビットがただ一つの 1 となるように配置する。このように、小数点の左に 0 が来ないようにした数を、正規化されている、と言う。(ただし、数値 “0” は小数点の左に 1 がくることがないので特別扱いをする)

MIPS ではこの 32 ビットのレジスタを図 14 のように配分している。(なお、この形式は MIPS 専用のものではなく、IEEE 754 浮動小数点規格により定められている)

この形式は単精度の浮動小数点形式と呼ばれる (C 言語での float)。仮数は (1) 式の “$xxxxx$” の部分のみを 2 進表現して 23 ビットに収める。


図 14: IEEE 754 における浮動小数形式 (単精度) の数値の表現。

指数部は一般に正負の値をとりうる (10 進数なら $10^3$ と $10^{-3}$ とがあるように)。指数は 2 の補数表現ではなく、図 15 のような**ゲタばき表現**がなされ、$-127 \sim 128$ の数値を表現している。すなわち、実

| 表現 | 実際の値 |
|---|---|
| 1111 1111 | 128 |
| 1111 1110 | 127 |
| ··· | |
| 1000 0000 | 1 |
| 0111 1111 | 0 |
| 0111 1110 | −1 |
| ··· | |
| 0000 0001 | −126 |
| 0000 0000 | −127 |

図 15: ゲタばき表現による指数部の表現

際の指数に 127 のゲタ (bias) を履かせ、その数値を 2 進表現するのである。

以上から、浮動小数表現された 32 ビットの数値が与えられた時、その意味する数は以下のように計算できる。

$$(-1)^S \times (1 + \text{仮数}) \times 2^{(\text{指数}-\text{ゲタ})} \qquad (2)$$

ただし、$S$ は符号ビットである。

[**問題**]

1. 10 進数の数値 −0.75 の IEEE 754 での 2 進表現 (単精度) で表せ。

### 3.2 MIPS における浮動小数点形式

MIPS には浮動小数を表す専用のレジスタが 32 個存在する。

PCspim の一番上のウィンドウにある FP0、FP1、···、FP31 がそれである。

浮動小数点形式には単精度の他に倍精度形式 (C 言語でいう double) もあり、その場合は上記レジスタが (FP0、FP1)、(FP2、FP3)、··· のように組になって使われる。(すなわち、一つの数値に 64 ビットが割り当てられる)

## A [付録] 数値の表現 (復習)

### A.1 [問題] 2 の補数表現

本章ではコンピュータ内での数値の表現を復習する。本章の内容の多くは論理回路の授業や演習で扱った内容の復習であるはずなので、付録として収録する。

コンピュータ内では数値は 2 進数で表現される。数値は多くの場合正負の数の両方を取り扱うが、例えばメモリアドレスのように常に正の数しかあつかわない場合がある。前者は「**符号付き数**」として表現され、後者は「**符号なし数**」として表現される。

符号なし数は常に正の数であるので取り扱いは簡単である。たとえば 4 ビットの 2 進数 1010 は、$2^3 + 2^1$ であるから 10 進数の 10 を表している。

符号付き数、すなわち正負の数の 2 進数での表現は、コンピュータの黎明期には様々な手法が試みられたが、現在は「**2 の補数表現**」が用いられる。2 の補数表現された 4 ビットの 2 進数と 10 進数の対応は以下のようになっている。

| | | | | | |
|---|---|---|---|---|---|
| 0111 | = | 7 | 1111 | = | −1 |
| 0110 | = | 6 | 1110 | = | −2 |
| 0101 | = | 5 | 1101 | = | −3 |
| 0100 | = | 4 | 1100 | = | −4 |
| 0011 | = | 3 | 1011 | = | −5 |
| 0010 | = | 2 | 1010 | = | −6 |
| 0001 | = | 1 | 1001 | = | −7 |
| 0000 | = | 0 | 1000 | = | −8 |

正の数は符号なしの場合と同じである。ただし、最上位ビット (most significant bit: MSB) はつねに 0 でなければならないので、表せる最大の数は 7 である。一方、負の数の場合は 最上位ビットは 1 であることに注意しよう。

一般に $n$ ビットの場合、$-2^{n-1} \sim 2^{n-1} - 1$ の範囲の数値を表現できる。(0 があるので、正の数のほうが 1 つだけ少ない)

負の数をこのように 表現する理由は、ハードウェア (加算器) の構成が簡単になるからである。例えば 「負の数を含んだ加算を統一的なアルゴリズムで実現できる」、また「正負の反転が (次章でみるように) 簡単にできるため、減算器も加算器を利用して簡単に構成できる」などのメリットがある。

[**問題**]

1. 4 ビットの 2 進数 1010 がある。これが符号なし数であるとすると、10 進数でいくつになるか。
2. 4 ビットの 2 進数 1010 がある。これが符号付き数であるとすると、10 進数でいくつになるか。

## A.2 [問題] 正負の反転

2 の補数表現された符号付き数の正負を反転する簡便法は以下の通りである。

- 与えられた 2 進数の 0/1 を反転 (ビット反転) し、 1 を足す。

例えば、前節で扱った 1110 (10 進数で $-2$) をビット反転すると 0001 となり、これに 1 を足すと 0010 (10 進数で 2) となる。

この方法は、負の数 (最上位ビットが 1 である数) が与えられたときに、その 10 進数での表現を得たいときに便利である。

[**問題**]

1. 5 ビットの符号付き数値 00101 がある。これは 10 進数で何を表すか。
2. 5 ビットの符号付き数値 10101 がある。これは 10 進数で何を表すか。
3. 10 進数の数値 $-5$ を、5 ビットの符号付き数値で表せ。

## A.3 [SPIM] 符号なし数と符号付き数の区別

上で見たように、最上位ビットが 1 である数、例えば 1010 が与えられたとき、これを 符号なし数とみなせば正の数となるし、これを符号付き数とみなせば 負の数となる。この区別はプログラマが意識して与えなければならない。その例を以下の問題で見よう。

[**問題**]

1. 圧縮ファイルの中の Slt.asm の中身を確認し、実行。プログラム内でレジスタ $t0 に 1 を、レジスタ $t1 に $-1$ を 代入している。Windows 版 SPIM の一番上のウィンドウで、$t0 および $t1 に何が格納されているか確かめよ。
2. 命令「slt $t2,$t1,$t0」は、$t1 < $t0 ならば $t2 に 1 を代入し、そうでなければ $t2 に 0 を代入する。このとき、slt 命令は$t0 と $t1 が符号付き数であると想定して動作する。一方、命令「sltu $t3,$t1,$t0」は、$t0 と $t1 が符号なし数であると想定して slt と同じ動作を行う。さて、Windows 版 SPIM において $t2 と $t3 には何が代入されているか調べよ。また、そのような結果になったはなぜか。

このように、与えられた 2 進数の意味を意識しなければならないのは、アセンブリ言語だけではない。C 言語においても 符号付き整数型 int と 符号なし整数型 unsigned int があり、用途に応じて使い分けなければならない。

## A.4 [問題] 符号拡張

今、4 ビットの数 0010 を 8 ビットに拡張することを考えよう。0010 は正の整数であるから、00000010 とすればよいと簡単にわかる。

では、2 の補数表現された 4 ビットの負の数 1110 を 8 ビットに拡張するにはどうしたらよいだろうか。

答えは 11111110 である。実は、正負の符号 (正なら 0、負なら 1) を必要な数だけ左にコピーすれば、簡単にビット幅を広げることができる。ビット幅を広げる際のこの手法を**符号拡張**という。

[**問題**]

1. 上の 1110 と 11111110 をそれぞれ 10 進数で表し、2 つが確かに同じ数になっていることを確かめよ。
2. 4 ビットの数 0100 を 8 ビットに拡張せよ。これは 10 進数でいくつか。

3. 4 ビットの数 1000 を 8 ビットに拡張せよ。これは 10 進数でいくつか。

### A.5 [SPIM] 符号拡張 (SPIM における例)

符号拡張はハードウェア中で頻繁に行われている。

先程のプログラム Slt.asm においてこのことを確認しよう。この中に、「addi $t1, $zero, −1」という命令がある。これは、$t1 = 0 + (−1) を実行する (すなわち $t1 = −1)。

このとき、addi は I 形式であるため、この命令を機械語によって表現すると定数 (即値) に割り当てられるのは 16 ビットである。16 ビットの −1 は 2 進数では 11···11 (1 が 16 個)、16 進数では 0xffff である。

つまり、上の命令「addi $t1, $zero, −1」は、レジスタ $zero に格納されている 32 ビットの 0 と、16 ビットの −1 (0xffff) を足し算していることになる。この時、シミュレータや実際のハードウェアでは 16 ビット の −1 は 32 ビットに符号拡張されてから、32 ビットの 0 と足されるのである。

このように、符号拡張は I 形式と J 形式 (即値を含む) の命令をハードウェアで実現する際に関係してくることがわかる。これは教科書の第 5 章や授業の「データパス」の回において詳しく見ることになるだろう。

[**問題**]

1. Slt.asm を実行した時のレジスタ $t1 の値をシミュレータで確認し、符号拡張が行われたのを確かめよ。

## B [付録] 加算・減算および論理演算 (復習)

### B.1 [問題] 加算・減算

本章では加算・減算および論理演算 (AND 、OR 等) を取り扱う。やはり論理回路の復習が多くなるだろう。

MIPS CPU (あるいはシミュレータ SPIM) が、32 ビットの数値の加算、減算等の算術演算を行えることはいままで見てきたとおりである。これらがハードウェアとしてどのように実現されているかを学ぶため、まず 6 ビットの数値に対する加算・減算を手計算で行なってみよう。

[**問題**]

1. 010101 + 000101 の計算を筆算で行なってみよ。答えを求めたあと、その計算が 10 進数の何に対応していたか確認してみよ。

2. 010101 + 001111 について、上と同様のことを行なえ。

3. 000010 + 111010 について、上と同様のことを行なえ。(ただし、符号つき数の計算であるとする。)

4. 000010 − 111010 について、上と同様のことを行なえ。ただし、$X - Y$ の計算は $X + (-Y)$ のように加算として取り扱う。$(-Y)$ は前回の「正負の反転」を用いる。

加算は下位ビット (least significant bit : LSB) から順に計算して行けば良く、このとき桁上がりが上位ビットに伝播してゆくことに注意しよう。

また、減算は事前に正負の反転を行なえば加算と同じアルゴリズムで実現できることもわかる。

### B.2 [問題] 加算・減算のオーバーフロー

ハードウェア的には、$n$ ビットの加算 (減算) はその演算結果も $n$ ビットに限定される。例えば、MIPS R2000 CPU のレジスタは 32 ビットであるから、32 ビットで表される最大数より大きい数は格納できない。そのため、正の大きい数や負の大きい数が演算結果に出る場合、結果に矛盾がでることがある。これがオーバーフローである。これを確かめてみよう。

[**問題**]

1. 010101 + 010101 を筆算で行なってみよ。10 進数でこの計算を表せ。結果は負の数になるが、その理由を考察せよ。

2. 100000 + 111111 を筆算で行なってみよ。ただし、最上位ビット (MSB) からの桁上がりは捨てられることに注意せよ。10 進数でこの計算を表し、結果について考察せよ。

## B.3　論理演算： AND、OR、シフト

今までの演習では取り扱わなかったが、シミュレータ SPIM には論理演算 AND 、 OR、シフト等の命令も含まれている。

AND、OR に対応する命令 and 、or はは各ビットごとに AND、OR をとる。例えば (0101 AND 0011) は 0001 、(0101 OR 0011) は 0111、といった具合である。

シフト命令 sll (左シフト) は、語中の全てのビットを左シフトし、空いた部分に 0 を埋める。シフト命令 srl (右シフト) は、語中の全てのビットを右シフトし、空いた部分に 0 を埋める。

例えば、00001100 を 左に 2 ビットシフトすると 00110000 であり、右に 2 ビットシフトすると 00000011 となる。シフトは乗算の実現に使われるが、今回は取り扱わない。

なお、上では 4 ビットや 8 ビットでの例を挙げたが、実際の MIPS 命令 and 、 or 、sll 、 srl ではもちろん 32 ビットの数 (具体的にはレジスタの中身) が対象となっている。