前回:
アセンブリ言語
シミュレータ SPIM
MIPSの命令

今回:
分岐命令, サブルーチン呼出し命令,
アセンブラ指令, プログラム例

キーワード:
条件分岐, 相対ジャンプ, アセンブラ指令

第7回 MIPSプロセッサの命令(2)

4.6.4 分岐命令 (branch)
  ひとつの手続きや関数の中で指定した場所に飛ぶ
  飛び先は現在の番地の「比較的近く」

4.6.4.1 無条件分岐
    b label

label はアセンブリ言語で指定したラベル
  機械命令では16ビット定数

動作： PC = PC + (16ビット定数<<2)の符号拡張
  つまり現在位置から ±2^17 番地まで(「比較的近く」)
    なぜ2ビット左シフトしているのか?
  相対ジャンプともいう

4.6.4.2 条件分岐 (conditional branch)
指定した条件が真である場合だけ飛ぶ
  偽だったら飛ばずに次の番地の命令を実行

  beq rs, rt, label
  (branch equal)
    (ふたつのレジスタの内容が等しいか)
    bne もある

  bgez rs, label
  (branch greater than or equal to zero)
    (レジスタ rs とゼロとの符号つき比較)
    bgtz, blez, bltz もある

  blt, ble, bgt, bge
    ふたつのレジスタを大小比較して飛ぶ命令

  bltu, bleu, bgtu, bgeu
    符号なし比較バージョン (0x80000000 > 0x7fffffff ?)
    (u for unsigned)

4.6.4.3 ジャンプ命令
分岐命令と異なり, 遠くに飛べる
  別の関数や手続きへ
  無条件ジャンプのみ

4.6.4.3.1 ラベルへのジャンプ
    j label
label はアセンブリ言語で指定したラベル
  機械命令では26ビット定数

動作: PC = 26ビット定数<<2
  相対ジャンプではない (現在のPCは使わない)

4.6.4.3.2 レジスタで指定した番地へのジャンプ
    jr rt
    (j rt と書いてもOK)
飛び先が固定されない

4.6.5 サブルーチン呼出しと戻り (プリント p.11)
4.6.5.1 サブルーチン呼出し
  帰ってくる番地を覚えておく必要
  単なる分岐命令やジャンプ命令ではダメ

  jal label
  "jump and link"
戻り番地をレジスタ $31 に覚えてからジャンプ
  飛び先はジャンプと同じ28ビット
  $31 の機能名は $ra (return address)

4.6.5.2 サブルーチンからの戻り
レジスタ $31 に入っている戻り番地に戻ればよい
つまり
  jr $31
でOK

★戻り番地はスタックに保存するのではなかったのか
  後ほど説明

4.6.5.3 バリエーションいくつか
  bal label
  (branch and link)
    戻り番地をレジスタ $31 に覚えてからブランチ
    飛び先は16ビット, 相対ジャンプ

  bgezal, bltzal
  (branch greater than or equal to zero and link)
    条件つき呼出し

  jalr rt
    レジスタで指定した番地へ呼出し

4.6.6 なにもしない命令
  nop
  "no operation"

  機械命令としては全ビット0

4.7 アセンブラ指令 (資料(3)の 13, p.12)

(アセンブラとは
  アセンブリ言語でのプログラムを機械語に翻訳するソフトウェア)

ふつうの命令とは別に, アセンブラに対する指令がある
  機械語への変換を制御するため
  機械命令には変換されない

形式: ピリオドで始まる

指令の種類
4.7.1 メモリ領域の切り替え
    実行コード(命令)をおく領域(テキスト領域) .text
    データをおく領域(データ領域) .data

4.7.2 定数データをメモリに確保
    .word, .byte, ..

4.7.3 データ領域の確保
    .space
    指定したサイズ(byte)の領域を確保

4.7.4 アラインメントをあわせる
    .align
    あるデータを倍語境界にあわせて配置したいとか

4.7.5 グローバルラベルの宣言 .global
    ラベルを別のファイルからアクセスできるか

4.8 命令をどう使うか - プログラム例

実際に命令の使用例を見ないと理解が難しい

まずは演算, 繰り返し, 配列あたりで
はじめにCのコードを示す

4.8.1 条件判断(if文)
4.8.1.1 elseのないif文
条件を否定した条件分岐命令で, thenの処理を飛び越す

```
  if(条件){
    ...
    ....
  }
```

は, アセンブリ言語ではこうする

```
    条件分岐命令(飛び先: skip) 元の条件の否定で skip へ
    ... に対応する命令
    .... に対応する命令
  skip:
```

(ここでの skip などのラベルは、出現ごとに異なるものを作る
 - そうしないと二重定義になってしまう)

4.8.1.2 elseのあるif文
条件を否定した条件分岐でthenの処理を飛び越してelseの処理へ進む.
then処理の末尾では無条件分岐で else処理との合流点へ進む.

```
  if(条件){
    ...
    ....
  } else {
    ***
    ****
  }
```

は, アセンブリ言語ではこうする

```
    条件分岐命令(飛び先: else) 元の条件の否定で else へ
    ... に対応する命令
    .... に対応する命令
    無条件分岐(飛び先: join)
  else:
    *** に対応する命令
    **** に対応する命令
  join:
```

4.8.2 繰り返し(while文)
条件分岐と無条件分岐の組み合わせで

```
  while(条件){
    ...
    ....
  }
```

は, アセンブリ言語ではこうする

```
  loop:
    条件分岐命令(飛び先: exit) 元の条件の否定で exit へ
    ... に対応する命令
    .... に対応する命令
    無条件分岐(飛び先: loop)
  exit:
```

==========

4.8.3 1から10までの和を計算する
レジスタだけを使う (データ用のメモリを使わない)
どのレジスタを使うか

```
#       t0=0;
#       t1=1;
#       t2=10;
#       while(t1<=t2){
#         t0=t0+t1;
#         t1=t1+1;
#       }

# パターン1: すなおにアセンブリ言語で書いてみる
        li      $t0, 0
        li      $t1, 1
        li      $t2, 10
loop0:
        bgt     $t1, $t2, exit0
        add     $t0, $t0, $t1
        add     $t1, $t1, 1
        b       loop0
exit0:

# パターン2: ループを「後判定」にしてみる
# 末尾の無条件分岐が不要になる
# ただし, 最低一回は実行してしまうので要注意
        li      $t0, 0
        li      $t1, 1
        li      $t2, 10
loop2:
        add     $t0, $t0, $t1
        add     $t1, $t1, 1
        ble     $t1, $t2, loop2
==================================
```

4.8.4 配列の要素の和を計算する
配列はメモリのデータ領域の連続領域にとる
配列要素の読み出しはロード命令
  intは4バイトであることに注意

```
#       int a[10] = {1,2,3,4,5,6,7,8,9,10};
#       t0=0;
#       t1=0;
#       t2=10;
#       while(t1<t2){
#         t0=t0+a[t1];
#         t1=t1+1;
#       }

# データの宣言
        .data
a:      .word   1,2,3,4,5,6,7,8,9,10

        .text

# パターン1: すなおにアセンブリ言語で書いてみる
        li      $t0, 0
        li      $t1, 0
        li      $t2, 10
        la      $t3, a
loop0:
        bge     $t1, $t2, exit0
        sll     $t4, $t1, 2     # 配列要素ひとつは4バイト
        add     $t4, $t4, $t3   # a[t1] の番地ができた
        lw      $t4, 0($t4)     # a[t1] の値を読み出す
        add     $t0, $t0, $t4   # 足し込む
        add     $t1, $t1, 1
        b       loop0
```

```
exit0:

2018-05-23

# パターン2: いろいろチューンアップしてみる
        li      $t0, 0
        li      $t2, 10         # まわった回数のカウンタに使う
        la      $t3, a          # いつも a[t1] の番地を指させる
loop1:
        lw      $t4, 0($t3)     # a[t1] を読み出して
        add     $t0, $t0, $t4   # 足し込む
        add     $t3, $t3, 4     # a[t1] の次の要素の番地に指しかえ
        sub     $t2, $t2, 1     # カウンタを1減らして
        bne     $t2, $0, loop1  # 0になってないならループする
exit1:
==================================
4.8.5 指定した数の約数の個数をもとめる
for文は, まずwhile文に直して考える
  ----
  for(初期化; 繰り返し条件; 更新処理){
    本体
  }
  ----
  初期化
  while(繰り返し条件){
    本体
    更新処理
  }
  ----
if文の例も含んでいる
(条件が複雑なので, 単一の条件分岐命令にはできないケース)

#       /* 「指定した数」は28とする */
#       t0 = 28;
#       t2 = 0;
#       for(t1=1; t1<t0; t1++){
#         if(t0 % t1 == 0){
#           t2 = t2 + 1;
#         }
#       }

        li      $t0, 28
        li      $t2, 0          # 約数の個数
        li      $t1, 1          # 割ってみる数
loop0:
        bge     $t1, $t0, exit0 # ループの終了チェック
        div     $t0, $t1        # 割り算
        mfhi    $t3             # 余りは HI レジスタ
        bne     $t3, $0, skip0
        add     $t2, $t2, 1     # 約数発見でカウントアップ
skip0:
        add     $t1, $t1, 1
        b       loop0
exit0:
==================================
```