前回:
アセンブラ指令, プログラム例(1)

今回:
サブルーチン, 機械命令, 合成命令

キーワード:
サブルーチン, スタック, オペコード, 合成命令

第8回 サブルーチン

4.9 サブルーチン

4.9.1 高級言語の関数/手続きの復習
宣言
```
int sum4(int i, int j, int k, int l){
  int f;
  f = sum2(i,j)+sum2(k,l);
  return f;
}
```

呼び出し
```
...
x = sum4(1,2,k,y+z);
...
```

引数
  仮引数 ((formal) parameter) - 呼ばれ側の引数
    i,j,k,l
    初期値の与えられるローカル変数
    関数内で値を変えてもよい
  実引数 (argument) - 呼び出し側の式
    対応する仮引数の初期値となる
    変数を書いても値を見るだけ
      対応する仮引数への代入は影響しない

ローカル変数
  その呼び出しの間だけ存在
    複数回呼び出されても別物

返り値

式や変数の値は1word(32bit)に納まるとする
  実際にはそうでないケースもあり
    double (64bit 浮動小数点)
    long long int (64bit 整数)

4.9.2 アセンブリ言語では
4.9.2.1 呼び出し
  jal (jump and link)
    戻り番地が$31に格納される

4.9.2.2 リターン
  jr $31 (jump register)
    $31が持つ番地へのジャンプ

あとは好きなように書けばよい、はずだが
  他人の作ったサブルーチンを利用したい
  そもそもシステムの提供するサブルーチンがある
    システムライブラリ
      ファイルの読み書き、キーボード入力、画面表示
    OSのAPI ともいう
      operating system (ex. Windows, Linux, MacOS)
      application program interface

  サブルーチン呼び出しの前後での規約が必要
    呼び出し規約 calling convention という

引数の渡し方
返り値の返し方
レジスタの使い方

OSの種類, CPUの種類によって決まる
以下ではMIPSの場合

4.9.2.3 実引数
先頭から4つまでの実引数は$a0-$a3にいれる
サブルーチン内では仮引数として使用

個数が5つ以上のときどうするか?
ためしてみよう

4.9.2.4 返り値
$v0にいれてリターンする

1wordに納まらない場合
たとえばdouble, long long int
$v1 も使う
さらに足りない場合
例: 構造体そのものを返す場合（ポインタではなくて）
メモリを使う

例(1) 簡単な関数 sum2 (配布資料(5))
ほかの関数を呼び出さない (leaf; 葉)
ローカル変数が(実質的に)ない
作業用のレジスタ不要

```
int sum2(int x, int y){
  int z;
  z = x + y;
  return z;
}
```

```
sum2:
      add     $v0, $a0, $a1   # z = x + y
      jr      $ra             # $ra : $31
```

例(2) 簡単な関数 mulsum (配布資料)
ほかの関数を呼び出さない
ローカル変数が(実質的に)ない
作業用のレジスタ必要(中間結果)

```
int mulsum(int i, int j, int k, int l){
  int f;
  f = (i+j)*(k+l);
  return f;
}
```

```
mulsum:
      add     $t0, $a0, $a1   # $t0 = i + j
      add     $v0, $a2, $a3   # f = k + l
      mul     $v0, $v0, $t0   # f = f * $t0
      jr      $ra
```

$t0 を作業用として使用

4.9.2.5 戻り番地の保存と回復
jal命令は返り番地を $31 ($ra) に保存する

関数内で関数を呼び出す場合
その呼び出しで自分の戻り番地($31)が上書きされる
メモリに保存する必要
保存場所
関数ごとの固定番地では不都合
再帰呼び出しで問題
スタックを利用する

あらかじめスタック領域を確保しておく
メモリ領域の高位番地が普通
若い番地に向かって伸びる
$spがその(使用中の)先頭を指す
保存と回復のタイミング
関数の入口と出口で

4.9.2.6 ローカル変数
一般にはメモリ上にとる(レジスタは少ない)
スタック
そうしないと再帰呼び出しの際に別物にならない
作業用レジスタ(次項)に割り当てることもあり
レジスタには個数の制限
配列とかは無理

4.9.2.7 レジスタの保存/回復
作業用レジスタ
式評価の途中結果を置く

t0-t7, t8-t9
サブルーチンで上書きしてもよい
呼び側から見ると
サブルーチンを呼んだら壊れるかも
呼ばれ側から見ると
保存と回復の必要なし

s0-s7
サブルーチンで壊すなら保存と回復が必要
呼び側から見ると
サブルーチンを呼んでも壊れない

t0-t7, t8-t9, a0-a3, v0-v1 が壊れては困る場合
呼び側が呼び出しの前後で保存/回復する

例(3) 他の関数を呼ぶ sum4

```
int sum4(int i, int j, int k, int l){
  int f;
  f = sum2(i,k)+sum2(j,l);
  return f;
}
```

```
sum4:
      sub     $sp, $sp, 12    # スタックに3ワード確保
      sw      $s0, 4($sp)     # $s0の保存は呼ばれ側の責任
      sw      $ra, 0($sp)     # 帰り番地を保存

      sw      $a1, 8($sp)     # $a1 を保存
      move    $a1, $a2        # $a1 = k
      jal     sum2            # sum2(i, k)
      lw      $a1, 8($sp)     # $a1 を回復

      move    $s0, $v0        # $s0 = 途中結果

      move    $a0, $a1
      move    $a1, $a3
      jal     sum2            # sum2(j, l)

      add     $v0, $s0, $v0   # 最終結果の計算

      lw      $s0, 4($sp)
      lw      $ra, 0($sp)
      add     $sp, $sp, 12    # 確保したスタックを解放
      jr      $ra
```

例(4) 多少複雑な関数 - factorial(階乗)
再帰呼び出し

```
int fact(int n){
  if(n < 1) return 1;
  else return (n * fact(n-1));
}
```

n < 1 のところは n <= 0 でも同じ

素直に書くと:

```
------------------------------
fact:
        sub     $sp, $sp, 8     # 2ワード確保
        sw      $s0, 4($sp)
        sw      $ra, 0($sp)
        bgtz    $a0, fact2      # if(n > 0)then fact2
        li      $v0, 1
        b       fact3

fact2:
        move    $s0, $a0        # a0 (n) を保存
        sub     $a0, $a0, 1
        jal     fact            # fact(n-1)
        mul     $v0, $s0, $v0   # n * fact(n-1)
fact3:
        lw      $ra, 0($sp)
        lw      $s0, 4($sp)
        add     $sp, $sp, 8
        jr      $ra
------------------------------
```

(参考) いろいろ工夫してみる:

- 再帰の末端(n<1のとき; 関数呼び出しなし)では$raを保存しない
- $a0 を保存しない
  - 再帰のために$a0を-1するが、帰ってきてから+1して回復
  - 再帰呼び出しで$a0を壊していないから可能
- 命令数が短くなり, スタック消費も減った

```
------------------------------
fact:
        bgtz    $a0, fact2      # if(n>0)then fact2
        li      $v0, 1
        jr      $ra             # return 1
fact2:
        sub     $sp, $sp, 4     # 1ワード確保
        sw      $ra, 0($sp)     # $ra だけ保存
        sub     $a0, $a0, 1
        jal     fact            # fact(n-1)
        add     $a0, $a0, 1
        mul     $v0, $a0, $v0   # n * fact(n-1)
        lw      $ra, 0($sp)
        add     $sp, $sp, 4
        jr      $ra
------------------------------
```

4.10 機械命令

実際にCPUが実行する命令

- アセンブリ言語の命令の変換先
- どういう形式になっているか

32ビット 固定長

32ビットをいくつかのフィールドに区切って情報をおさめてある

- 命令形式、命令フォーマットという R形式, I形式, J形式

講義ページに正式なリファレンスマニュアルへのリンクを置いてあります。

4.10.1 R形式

```
  6      5     5     5     5     6
......|.....|.....|.....|.....|......
  op    rs    rt    rd   shamt funct
```

op - 命令の種類を決める
  オペコード(opcode) (operation code, operator code)
rs, rt, rd - オペランドに指定されるレジスタ(5bit - 32個)
shamt - シフト命令におけるシフト量 (5bit - 0から31ビットのシフト幅)
funct - 命令の種類の細分化

例: add rd, rs, rt (レジスタ間の加算)

add $10, $8, $9

000000 01000 01001 01010 00000 100000

4ビットずつ区切って16進だと: 0x01095020

4.10.2 I形式

メモリに対するロード、ストア
即値の演算 (レジスタ+定数 とか)
分岐命令

16ビットの即値データを含む

```
  6      5     5          16
......|.....|.....|................
  op    rs    rt        immediate
```

rs, rt - レジスタ指定
  (rdの指定箇所はないので、レジスタに書き込む命令では
   結果はrtのレジスタに格納される)
immediate - 即値16ビット.
        メモリのアドレシングにおける変位
        即値
        分岐の変位

例: add rt, rs, si16   (レジスタ+符号つき定数)

(加算命令はレジスタ同士の加算もレジスタと即値の加算も
 同じ add とアセンブリ言語で書けたが、機械命令としては
 add と addi という別の命令である。
 アセンブラが自動的に変換してくれていた。)

addi $9, $10, 1

001000 01010 01001 0000 0000 0000 0001

0x21490001

例: 条件分岐
```
       bne    $t3, $t4, skip  # これ
       nop
skip:
```

bne $11, $12, skip

000101 01011 01100 0000 0000 0000 0002

0x156c0002

分岐の変位は、分岐命令自身が起点なので +8
命令には2ビット右シフトして入るので、結局 2 になる

4.10.3 J形式

ジャンプ命令

```
  6              26
......|..........................
  op          target
```

target - 飛び先の番地

命令の種類は、まずopフィールドで大別され、
  三つの形式で op フィールドは共通 (先頭6ビット)
さらにopによってはfunctフィールドで細分される

4.11 合成命令
「アセンブリ言語の命令は機械命令と一対一対応」と説明してきたが
実は...

アセンブリ言語の命令を 別の(1つ以上の)機械語命令に変換
  機械語命令ではできない操作
  別の機械語命令がすでにある操作
などを人間が記述しやすくするため
(配布資料「MIPSへの招待」では疑似命令と表記)

シミュレータSPIMでは, 変換された機械命令が
表示されている
(もとの命令はコメントで付記されている)

例1: move rd, rs (レジスタrsの内容をrdにコピーする)
    -> add rd, $0, rs
    (rd = $0 + rs のこと; $0 は常に0が入っているレジスタ)
      or rd, $0, rs でも可能
    つまり、機械命令に move は存在しない

例2: li   rd, 0x1234
ロードする値が16ビットに収まっている場合
$0 は常に0, を利用して
    -> ori   rd, $0, 0x1234

例3: li   rd, 0x12345678
ロードする値が32ビットある場合
そもそも命令が32ビットしかないのだから, 32ビットの定数は入らない
16ビットずつ二命令に分割

    -> lui $1, <上位16ビット>
       ori   rd, $1, <下位16ビット>

例4: blt $t0, $t1, label ($t0<$t1 ならジャンプ)

機械命令には存在しない条件分岐
  比較命令で結果の真偽を$1にセットし
  $1の真偽(1/0)に従って条件ジャンプ

    -> slt $1, $t0, $t1
       bne $1, $0, label

複数の機械語命令に展開するとき、作業用として$1を使うことあり
  プログラマから見ると「いつの間にか$1が変化している」
    人間は使わないようにする

2018-05-30
(ここから中間試験範囲外)
4.12 本物のコンパイラの出力
ここまで説明してきたアセンブリ言語プログラムは人手で作ったもの
本物のCコンパイラが出力するコードとは少し異なる

4.12.1 ソースプログラム simple.c

```
int empty(int i){
}

int add(int x, int y){
  int z = x + y;
  empty(z);
```

```
    return z;
  }
```

4.12.2 コンパイル結果（最適化なし）
以下のコマンドを実行した結果

```
  mips64-linux-gnu-gcc -mabi=32 -march=r3000 -fno-delayed-branch -S simple.c
```

「-S」オプションはアセンブリ言語コードを出力する指示

=====

```
 1      .file   1 "simple.c"
 2      .section .mdebug.abi32
 3      .previous
 4      .gnu_attribute 4, 1
 5      .abicalls
 6      .text
 7      .align  2
 8      .globl  empty
 9      .set    nomips16
10      .ent    empty
11      .type   empty, @function
12  empty:
13      .frame  $fp,8,$31               # vars= 0, regs= 1/0, args= 0, gp= 0
14      .mask   0x40000000,-4
15      .fmask  0x00000000,0
16      .set    noreorder
17      .set    nomacro
18      addiu   $sp,$sp,-8
19      sw      $fp,4($sp)
20      move    $fp,$sp
21      sw      $4,8($fp)
22      move    $sp,$fp
23      lw      $fp,4($sp)
24      addiu   $sp,$sp,8
25      j       $31
26      nop
27
28      .set    macro
29      .set    reorder
30      .end    empty
31      .size   empty, .-empty
32      .align  2
33      .globl  add
34      .set    nomips16
35      .ent    add
36      .type   add, @function
37  add:
38      .frame  $fp,40,$31              # vars= 8, regs= 2/0, args= 16, gp= 8
39      .mask   0xc0000000,-4
40      .fmask  0x00000000,0
41      .set    noreorder
42      .set    nomacro
43      addiu   $sp,$sp,-40
44      sw      $31,36($sp)
45      sw      $fp,32($sp)
46      move    $fp,$sp
47      sw      $4,40($fp)
48      sw      $5,44($fp)
49      lw      $3,40($fp)
50      lw      $2,44($fp)
51      nop
52      addu    $2,$3,$2
53      sw      $2,24($fp)
54      lw      $4,24($fp)
55      .option pic0
56      jal     empty
57      nop
58
59      .option pic2
60      lw      $2,24($fp)
```

```
61      move    $sp,$fp
62      lw      $31,36($sp)
63      lw      $fp,32($sp)
64      addiu   $sp,$sp,40
65      j       $31
66      nop
67
68      .set    macro
69      .set    reorder
70      .end    add
71      .size   add, .-add
72      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-16)"
```

=====

説明してない疑似命令が大量にあるが、それらは気にしない

addの本質的なコードは L.43 から L.66

コンパイラが出力するコードの基本形がわかる

これまで出てこなかった $fp が使われている

- スタックフレームの基準点となる番地を指す
- スタックフレームへの参照はすべて $fp を使う
  - ローカル変数と引数
    - x： $4($a0)で渡ってくる；スタックでは40($fp)；L.47で保存；L.49で使用
    - y： $5($a1)で渡ってくる；スタックでは44($fp)；L.48で保存；L.50で使用
    - z： スタックでは24($fp)；L.53で保存；L.54、L.60で使用
  - 戻り番地：$31に入っている；L.44で36($fp)に保存；L.62で回復
  - 呼出側のfp：L.45で保存；L.63で回復

4.12.3 コンパイル結果（最適化あり）

以下のコマンドを実行した結果

```
mips64-linux-gnu-gcc -mabi=32 -march=r3000 -fno-delayed-branch -O -S simple.c
```

「-O」オプションは最適化を指示 - 無駄なコードを出さなくなる

=====

```
 1      .file   1 "simple.c"
 2      .section .mdebug.abi32
 3      .previous
 4      .gnu_attribute 4, 1
 5      .abicalls
 6      .text
 7      .align  2
 8      .globl  empty
 9      .set    nomips16
10      .ent    empty
11      .type   empty, @function
12  empty:
13      .frame  $sp,0,$31               # vars= 0, regs= 0/0, args= 0, gp= 0
14      .mask   0x00000000,0
15      .fmask  0x00000000,0
16      .set    noreorder
17      .set    nomacro
18      j       $31
19      nop
20
21      .set    macro
22      .set    reorder
23      .end    empty
24      .size   empty, .-empty
25      .align  2
26      .globl  add
27      .set    nomips16
28      .ent    add
29      .type   add, @function
30  add:
31      .frame  $sp,0,$31               # vars= 0, regs= 0/0, args= 0, gp= 0
32      .mask   0x00000000,0
33      .fmask  0x00000000,0
34      .set    noreorder
35      .set    nomacro
```

```
36      addu    $2,$4,$5
37      j       $31
38      nop
39
40      .set    macro
41      .set    reorder
42      .end    add
43      .size   add, .-add
44      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-16)"
=====
```
add の本体は L36-L38 だけになってしまった。
  $2: $v0 関数の返り値
  $4: $a0 関数の第一引数
  $5: $a1 関数の第二引数
  手続き empty の呼び出しも何もしていないのがわかって削除された

これではあんまりなので、資料(5)の裏面にもう少し複雑なプログラムの
処理結果を載せた。
最適化してもなくならない程度に複雑。