システムプログラミング 1
レポート

氏名: 今田　将也 (IMADA, Masaya)
学生番号: 09430509

出題日: 2019 年 10 月 07 日
提出日: 2019 年 11 月 20 日
締切日: 2019 年 11 月 25 日

1　概要

本演習では，PIM という MIPS CPU シミュレータのハードウェア上に C 言語とアセンブリ言語を使用して文字の表示と入力のためのシステムコールライブラリを作成する． さらに，そのライブラリを使用して printf 及び gets 相当を C 言語で作成する． 最後に，それらを利用した応用プログラムを動作させる．

なお、与えられた課題内容を以下に述べる．

1.1　課題内容

以下の課題についてレポートをする． プログラムは，MIPS アセンブリ言語及び C 言語で記述し，SPIM を用いて動作を確認している．

2-1 SPIM が提供するシステムコールを C 言語から実行できるようにしたい． 教科書 A.6 節「手続き呼出し規約」に従って，各種手続きをアセンブラで記述せよ． ファイル名は， syscalls.s とすること．また，記述した syscalls.s の関数を C 言語から呼び出すことで， ハノイの塔 (hanoi.c とする) を完成させよ．

```
 1: void hanoi(int n, int start, int finish, int extra)
 2: {
 3:   if (n != 0){
 4:     hanoi(n - 1, start, extra, finish);
 5:     print_string("Move disk ");
 6:     print_int(n);
 7:     print_string(" from peg ");
 8:     print_int(start);
 9:     print_string(" to peg ");
10:     print_int(finish);
11:     print_string(".\n");
12:     hanoi(n - 1, extra, finish, start);
```

```
13:   }
14: }
15: main()
16: {
17:   int n;
18:   print_string("Enter number of disks> ");
19:   n = read_int();
20:   hanoi(n, 1, 2, 3);
21: }
```

spim-gcc によって hanoi.s ができたら， hanoi.s, syscalls.s の順に SPIM 上でロードして実行.

実行例は以下の通り:

```
Enter number of disks> 3
Move disk 1 from peg 1 to peg 2.
Move disk 2 from peg 1 to peg 3.
Move disk 1 from peg 2 to peg 3.
Move disk 3 from peg 1 to peg 2.
Move disk 1 from peg 3 to peg 1.
Move disk 2 from peg 3 to peg 2.
Move disk 1 from peg 1 to peg 2.
```

2-2 hanoi.s を例に spim-gcc の引数保存に関するスタックの利用方法について，説明せよ． そのことは，規約上許されるスタックフレームの最小値 24 とどう関係しているか． このスタックフレームの最小値規約を守らないとどのような問題が生じるかについて解説せよ.

hanoi.c のソースコード

2-3 以下のプログラム report2-1.c をコンパイルした結果をもとに， auto 変数と static 変数の違い，ポインタと配列の違いについてレポートせよ.

```
 1: int primes_stat[10];
 2:
 3: char * string_ptr   = "ABCDEFG";
 4: char   string_ary[] = "ABCDEFG";
 5:
 6: void print_var(char *name, int val)
 7: {
 8:   print_string(name);
 9:   print_string(" = ");
10:   print_int(val);
11:   print_string("\n");
12: }
```

```
13:
14: main()
15: {
16:   int primes_auto[10];
17:
18:   primes_stat[0] = 2;
19:   primes_auto[0] = 3;
20:
21:   print_var("primes_stat[0]", primes_stat[0]);
22:   print_var("primes_auto[0]", primes_auto[0]);
23: }
```

2-4 printf など，一部の関数は，任意の数の引数を取ることができる． これらの関数を可変引数関数と呼ぶ． MIPS の C コンパイラにおいて可変引数関数の実現方法について考察し，解説せよ．

2-5 printf のサブセットを実装し， SPIM 上でその動作を確認する応用プログラム (自由なデモプログラム) を作成せよ． フルセットにどれだけ近いか，あるいは，よく使う重要な仕様だけをうまく切り出して， 実用的なサブセットを実装しているかについて評価する． ただし，浮動小数は対応しなくてもよい (SPIM 自体がうまく対応していない)． 加えて，この printf を利用した応用プログラムの出来も評価の対象とする．

1.2 xspim の実行方法

```
$ xspim -mapped_io&
```

でコンソール上で実行後，必要なアセンブリファイルを load し，run することで実行した．

1.3 c ソースコードからアセンブリファイルへの変換方法

```
$ spim-gcc file.c
```

でコンソール上で実行後，file.c に対応する file.s というアセンブリファイルが作られる．

2 課題 2-1

以下に作成したプログラムと，作成内容、また作成時の考察を記載する．

2.1 作成したプログラム

syscalls.s

```
1         .text
2         .align  2
3
4 _print_int:
```

```
 5         subu  $sp, $sp, 24
 6         sw    $ra, 20($sp)
 7
 8         li    $v0, 1  # 1: print_int
 9         syscall
10
11         lw    $ra, 20($sp)
12         addu  $sp, $sp, 24
13         j     $ra
14
15 _print_string:
16         subu  $sp, $sp, 24
17         sw    $ra, 20($sp)
18
19         li    $v0, 4  # 4: print_string
20         syscall
21
22         lw    $ra, 20($sp)
23         addu  $sp, $sp, 24
24         j     $ra
25
26 _read_int:
27         subu  $sp, $sp, 24
28         sw    $ra, 20($sp)
29
30         li    $v0, 5  # 5: read_int
31         syscall
32
33         lw    $ra, 20($sp)
34         addu  $sp, $sp, 24
35         j     $ra
36
37 _read_string:
38         subu  $sp, $sp, 24
39         sw    $ra, 20($sp)
40
41         li    $v0, 8  # 8: read_string
42         syscall
43
44         lw    $ra, 20($sp)
45         addu  $sp, $sp, 24
46         j     $ra
47
48 _exit:
```

```
49          subu  $sp, $sp, 24
50          sw    $ra, 20($sp)
51
52          li    $v0, 10  # 10: exit
53          syscall
54
55          lw    $ra, 20($sp)
56          addu  $sp, $sp, 24
57          j     $ra
58
59 _print_char:
60          subu  $sp, $sp, 24
61          sw    $ra, 20($sp)
62
63          li    $v0, 11  # 11: print_char
64          syscall
65
66          lw    $ra, 20($sp)
67          addu  $sp, $sp, 24
68          j     $ra
69
70 _read_char:
71          subu  $sp, $sp, 24
72          sw    $ra, 20($sp)
73
74          li    $v0, 12  # 12: _read_char
75          syscall
76
77          lw    $ra, 20($sp)
78          addu  $sp, $sp, 24
79          j     $ra
```

2.2 ハノイの塔について

ハノイの塔とは 3 本の杭と、中央に穴の開いた大きさの異なる複数の円盤から構成され，最初はすべての円盤が左端の杭に小さいものが上になるように順に積み重ねられている。円盤を一回に一枚ずつどれかの杭に移動させることができるが、小さな円盤の上に大きな円盤を乗せることはできないというルールに従いすべての円盤を右端の杭に移動させられれば完成。

解法に再帰的アルゴリズムが有効な問題として有名であり、プログラミングにおける再帰的呼出しの例題としてもよく用いられる。

2.3 プログラムの説明及び作成時の考察

作成は，手続き呼出し規約に基づいて，各ルーチンごとにスタックポインタをルーチンの開始時に確保し，終了時に破棄して呼び出された関数に戻る設計にしている．`syscall` でカーネルに

所望することを$v0 レジスタへ格納し，syscall を呼び出している．

print_int に対応する関数は，4 行目から 13 行目に記載している．print_string に対応する関数は，15 行目から 24 行目に記載している．read_int に対応する関数は，26 行目から 35 行目に記載している．read_string に対応する関数は，37 行目から 46 行目に記載している．exit に対応する関数は，48 行目から 57 行目に記載している．print_char に対応する関数は，59 行目から 68 行目に記載している．read_char に対応する関数は，70 行目から 79 行目に記載している．

なお、今回の hanoi.c には用いないが，文字列をユーザから受け付ける read_string，数値をユーザから受け付ける read_int と文字を表示する print_char と文字をユーザから受け付ける read_char，そして，プログラムを終了する exit を作成した．

作成したプログラム中のラベルの先頭にアンダーバーをつけているがこれは，本演習で用いた gcc のルールでコンパイラに依存するものであるが，アセンブリ中で_function_name と記述しておくと，C 言語から function_name で呼び出すことができるからである．

3 課題 2-2

以下に課題内容に対する考察を記載する．

3.1 spim-gcc の引数保存に関するスタックの利用方法

説明のために，以下に hanoi.s の冒頭の数行を抜粋する．

```
30 _hanoi:
31 subu $sp,$sp,24
32 sw     $ra,20($sp)
33 sw     $fp,16($sp)
34 move $fp,$sp
35 sw     $a0,24($fp)
36 sw     $a1,28($fp)
37 sw     $a2,32($fp)
38 sw     $a3,36($fp)
39 lw     $v0,24($fp)
```

31 行目で，スタックを 24 バイト分確保していることが分かる．しかし，35 行目から利用されているレジスタ$a0 $a3 の 4 つは，確保したスタックよりも後方の_hanoi を呼び出した側の関数が確保したスタックを使用している．ここで，新しく関数から呼び出された表 3.1 にスタックの様子を表に表してみる．MIPS のコンパイラは,1 つ目の引数は$a0 に,2 つ目の引数は$a1 にという具合に$a のレジスタを使って引数を渡すことになっている．しかし，$a0〜$a3 の 4 つしかないため，5 つ目の引数は，スタックに保存して渡す．また，手続き呼出し規約に基づくと，offset =0〜＋15 の領域が必要になる．

まとめると，関数を呼び出す側は$a0〜$a3 を保存する領域を余分に確保しておき，呼び出された側がその領域を使って引数を保存することになっている．

3.2 最小値規約について

最小値規約とは spim-gcc において，規約上許されるスタックフレームの最小値が 24 であるという規約である．全 24 バイトのうち 16 バイトは$a レジスタの 4 語分であり，残りの 8 バイトは

$sp	offset	内容	備考
新 sp	-24	-	未使用
	-20	-	未使用
	-16	-	未使用
	-12	-	未使用
	-8	$fp	フレームポインタ
	-4	$ra	戻りアドレス
旧$sp	0	$a0	第 1 引数
	+4	$a1	第 2 引数
	+8	$a2	第 3 引数
	+12	$a3	第 4 引数
	+16	??	呼出側で使用
	+20	??	呼出側で使用
	...	??	呼出側で使用

表 1: スタックの様子

フレームポインタに利用される`$fp`レジスタの 1 語分と戻りアドレスに利用される`$ra`レジスタである．

この決まりを守らない関数が，仮に呼出される側であった場合は，`$a0`〜`$a3`の保存に自分で確保した領域しか使わないであろうから， 他の関数のスタック領域を破壊することがない．そのため，gcc から呼出しても問題がない．しかし，逆の場合, すなわち呼び出す側だった場合には，自分の関数のために確保したスタックを呼出し先が破壊することになるという問題がある．

この方法には利点がある．

利点 1 被呼出し関数が`$a0`〜`$a3`の保存をするかしないかを決定できるので， 関数内で`$a0`〜`$a3`を書換えなければ，この保存は省略できるため，メモリへの書込み処理が減り，高速化が望める．`$a0`〜`$a3`を呼び出す側で保存することにしてしまうと，上記の 4 つの引数をメモリに格納する操作が必ず必要になる．これでは，引数をレジスタ渡しではなく，実体として渡していることになる．

利点 2 第 5 引数以降が第 4 引数までの確保領域と連続するため，被呼出し関数から見れば，第 1 引数からのすべての引数が規則正しくメモリ上に並ぶことになる．そのため，コンパイラの実装が容易になる．

C 言語との連携には，この規約を守る必要があるため， 最小のスタックフレームサイズは，24 バイトとなっている．(引数 1 つ目〜4 つ目 (`$a0`〜`$a3`),`$ra`,`$fp`の 6 レジスタ*4 バイト = 24 バイト)

4 課題 2-3

以下に課題内容に対する考察を記載している．先に，auto 変数と static 変数の違いについて述べた後にポインタと配列の違いについて C 言語とアセンブリの観点から述べる．

4.1 report2-1.c のコンパイル結果

```
1 .file 1 "report2-1.c"
```

```
 2
 3  # -G value = 0, Arch = r2000, ISA = 1
 4  # GNU C version 2.96 20000731 (Red Hat Linux 7.3 2.96-113.2) (mipsel-linux)
compiled by GNU C version 2.96 20000731 (Red Hat Linux 7.3 2.96-113.2).
 5  # options passed:  -mno-abicalls -mrnames -mmips-as
 6  # -mno-check-zero-division -march=r2000 -O0 -fleading-underscore
 7  # -finhibit-size-directive -fverbose-asm
 8  # options enabled:  -fpeephole -ffunction-cse -fkeep-static-consts
 9  # -fpcc-struct-return -fsched-interblock -fsched-spec -fbranch-count-reg
10  # -fnew-exceptions -fcommon -finhibit-size-directive -fverbose-asm
11  # -fgnu-linker -fargument-alias -fleading-underscore -fident -fmath-errno
12  # -mrnames -mno-check-zero-division -march=r2000
13
14
15 .rdata
16 .align 2
17 $LC0:
18 .asciiz "ABCDEFG"
19 .data
20 .align 2
21 _string_ptr:
22 .word $LC0
23 .align 2
24 _string_ary:
25 .asciiz "ABCDEFG"
26 .rdata
27 .align 2
28 $LC1:
29 .asciiz " = "
30 .align 2
31 $LC2:
32 .asciiz "\n"
33 .text
34 .align 2
35 _print_var:
36 subu $sp,$sp,24
37 sw $ra,20($sp)
38 sw $fp,16($sp)
39 move $fp,$sp
40 sw $a0,24($fp)
41 sw $a1,28($fp)
42 lw $a0,24($fp)
43 jal _print_string
44 la $a0,$LC1
```

```
45 jal _print_string
46 lw $a0,28($fp)
47 jal _print_int
48 la $a0,$LC2
49 jal _print_string
50 move $sp,$fp
51 lw $ra,20($sp)
52 lw $fp,16($sp)
53 addu $sp,$sp,24
54 j $ra
55 .rdata
56 .align 2
57 $LC3:
58 .asciiz "primes_stat[0]"
59 .align 2
60 $LC4:
61 .asciiz "primes_auto[0]"
62 .text
63 .align 2
64 main:
65 subu $sp,$sp,64
66 sw $ra,60($sp)
67 sw $fp,56($sp)
68 move $fp,$sp
69 li $v0,2 # 0x2
70 sw $v0,_primes_stat
71 li $v0,3 # 0x3
72 sw $v0,16($fp)
73 la $a0,$LC3
74 lw $a1,_primes_stat
75 jal _print_var
76 la $a0,$LC4
77 lw $a1,16($fp)
78 jal _print_var
79 move $sp,$fp
80 lw $ra,60($sp)
81 lw $fp,56($sp)
82 addu $sp,$sp,64
83 j $ra
84
85 .comm _primes_stat,40
```

4.2 C 言語から見た static と auto の違い

説明のために，以下に課題の C のソースコードを一部抜粋する．

```
 1 int primes_stat[10];
 2 char * string_ptr   = "ABCDEFG";
 3 char   string_ary[] = "ABCDEFG";
 4 main()
 5 {
 6 int primes_auto[10];
 7 primes_stat[0] = 2;
 8 primes_auto[0] = 3;
 9 print_var("primes_stat[0]", primes_stat[0]);
10 print_var("primes_auto[0]", primes_auto[0]);
11 }
```

1 行目の関数外で宣言されている変数は，static(静的) 変数である．また，5 行目の関数内で宣言されている変数は，auto(自動) 変数であるという．以下にそれぞれの変数の特徴を示してみる．

auto 変数 関数の中で宣言され，その関数の実行開始時から 終了時までの間，その値を保持する．

static 変数 プログラムの開始から終了まで，値を保持しつづける．

両者の違いをアセンブラのソースコードを元に次節から調べてみる．

4.3 アセンブリにおける auto 変数

ソースコード内の auto 変数である `primes_auto` は 1.1 節の `report2-1.c` には明らかに区別されて存在している．しかし，アセンブリのソース 4.1 節からは該当の部分を簡単に発見はできなかった．60 行目にある文字列からラベル`$LC4` が使われているところを辿ってみると，78 行目において呼び出している`_print_var` の第 2 引数の内容が `primes_auto[0]` の値だと推測した．77 行目の`$a1` に入っている値すなわち，`16($fp)` のことである．つまり，新`$sp+16` バイト目であり，スタック上に存在していることになる．

そして，82 行目の操作によって，スタックを解放しているためこれ以降は値が使えなくなる．main 関数における自動変数宣言は main 関数の終了とプログラムの終了がほぼ同じような意味を持つため，意識をする必要はないように思う．以下にスタックの様子を示す.

$sp	offsset	内容	備考
	-16	primes_auto	新$sp + 16 バイト目
..	..	..	..
新$sp →	-04	$ra	戻りアドレス
..	..	..	..
旧$sp →	+00	$a0	第 1 引数
	+04	$a1	primes_auto[0] の値

表 2: スタックの様子

4.4 アセンブリにおける static 変数

report2-1.c をアセンブリに変えたコンパイル結果より，85 行目にて以下の記述を見つけた．

```
85 .comm _primes_stat,40
```

この宣言でデータセグメント内にデータを 40 バイト確保していた．これは，_primes_stat のみ仕様されるもので，プログラムの開始から終了まで，値を保持しつづけるという性質を持つことになる．primes_stat は常にその領域しか使用しないので，関数などが再帰的に呼び出された場合は，その領域を上書きすることがある．

そのため，プログラムの開始から終了まで，値を保持しつづける一方で，固定された領域 (static な領域) のみを使用するので，再帰やスレッドによる並行処理では，上書きの危険があるといえる．

4.5 C 言語における static というキーワード

C 言語において static という言葉は，2 つの意味を持っていた．1 つは，スタック上ではなく，プログラム中に静的に存在する領域にデータを確保するという意味である．もう 1 つは，static を付けると変数が外部から参照できる範囲が変化するということである．具体的には，関数外で static を付けて宣言した変数は，外部のファイルからは参照できない．簡単にいうと，複数の C 言語のファイルから構成されるプログラムにおいて，あるファイル内だけからしか参照できない変数を宣言できる．

static は，関数内でも有効に働くので，その場合は変数の有効範囲ではなく，記憶クラスを指定する．以下にその宣言と解釈した内容を表に示す．

宣言例	スコープ：見える範囲	記憶クラス（寿命）
static int a;（関数内）	関数内	静的（プログラム中）
static int a;（関数外）	ファイル全体	静的（プログラム中）
int a;（関数内）	関数内	自動（関数中）
int a;（関数外）	プログラム全体	静的（プログラム中）

表 3: static と int

4.6 ポインタと配列の C 言語での違い

C 言語でのポインタと配列の違いについて 4.6.1 節のソースコードを作り考察した．

配列 array とポインタ変数 pointer の値を表示する 9 行目と 10 行目の結果はいずれも同じであった．つまり配列は配列名だけだと，その配列の先頭アドレスを指すという事がわかる．すなわち，pointer と array で array 配列の値には同じようにアクセスすることができる．これは，12・13 行目，15・16 行目を表示した結果からわかる．

続いて，pointer と array のアドレスを見てみた．すると，両者は異なっていたが，&array と array は同じ値になっていた．一方，pointer は別のアドレスから配列 array の先頭アドレスを指していた．これは，ポインタにはアドレスを保存するメモリがあるが，配列にはアドレスを格納するメモリがないと言えるだろう．

つまり，C 言語においてポインタはアドレスを格納する変数であるのに対し、配列は単なるアドレスであると考える．

4.6.1 作成したプログラム

```
 1  #include<stdio.h>
 2
 3  int main(void){
 4      char array[3] ='abc';//char 型３つ分と
 5      char *pointer;//char* 型１つ分のメモリが確保
 6
 7      pointer = array;//ポインタ変数が配列の先頭アドレスを指す
 8
 9      printf("array   = %p\n", array);
10      printf("pointer = %p\n", pointer);
11
12      printf("array[2]   = %c\n", array[2]);
13      printf("pointer[2] = %c\n", array[2]);
14
15      printf("*array   = %c\n", *array);
16      printf("*pointer = %c\n", *pointer);
17
18      printf("&array   = %p\n", &array);
19      printf("&pointer = %p\n", &pointer);
20
21      return 0;
22  }
```

4.6.2 出力結果

```
array   = 0061FF1D
pointer = 0061FF1D
array[2]   = c
pointer[2] = c
*array   = a
*pointer = a
&array   = 0061FF1D
&pointer = 0061FF18
```

4.7 ポインタと配列のアセンブラでの違い

アセンブラでの違いについて report2-1.s を見てみる．すると，17 行目から 27 行目にその違いが現れていた．ポインタで宣言した string_ptr は.word というアセンブリ指令にて，32 ビットの数値をメモリに順番に配置されている．その数値は，"ABCDEFG"ではなく，そのワードが示すラベルのアドレスが格納されている．一方，配列として宣言した string_ary には"ABCDEFG"というデータ自体が格納されている．

4.8 考察

以上のコードより，配列とは、多数の変数を順番つけでまとめて扱う方法で，ポインタとは、変数のショートカットを作る方法であると考える。ポインタと配列が似たような使い方が出来るのは配列の設計と関係あるのではないかと考えた。実際，C 言語では、配列を実現する手段として、ポインタを利用している。従って、ポインタ変数では、配列と同等のことが出来ると考える．

5 課題 2-4

5.1 概要

この節では，まずはじめに可変引数について説明し，その後 C 言語における可変引数関数の実現方法と，MIPS における可変引数の実現方法について考察し，解説を行う．

5.2 可変引数とは

可変引数とはプログラミング言語において，関数やメソッドやマクロの引数が固定ではなく任意の個数となっている引数のことである．可変長引数，可変個引数とも呼ばれる．そのような関数を可変長引数関数と言う．C 言語では，可変長の引数を扱うために，…を使った構文が用意されている．例えば以下のような記述である．

```
int myfunction(char *fmt, ...)
```

第 2 以降の引数の個数は不定で，0 個でも構わない．代表的な使用例としては，printf がある．

5.3 C 言語における可変引数関数の実現方法

可変引数を宣言した関数ではいくつか疑問がある．

疑問 1 呼び出された関数内で，引数をどう参照すればいいのか．第 1 引数は，変数名で参照できそうだが，第 2 引数以降を名前で参照することができない．

疑問 2 いくつの引数が呼ばれたかをどう判断するのか．また，それぞれの引数の型をどうやって知ればよいのか．

5.3.1 疑問 1 の考察

ここで再度，課題 2-1 の `hanoi()` 関数の冒頭を見てみる．

```
30 _hanoi:
31 subu $sp,$sp,24
32 sw     $ra,20($sp)
33 sw     $fp,16($sp)
34 move $fp,$sp
35 sw     $a0,24($fp)
36 sw     $a1,28($fp)
```

```
37 sw     $a2,32($fp)
38 sw     $a3,36($fp)
39 lw     $v0,24($fp)
```

上記の出力内容と表 3.1 より，第 2 引数は (旧$sp + 04) からの 4 バイトに順次格納されている．つまり，C 言語で可変引数関数を記述して第 2 引数以降の値を得ようとすると，(旧$sp + 04) の値を C 言語で取得する必要がある．しかし，C 言語からレジスタの値を直接得る方法がわからない．そこで，第 1 引数が名前で参照できることを利用する．

5.2 節の例の旧$sp ＝第 1 引数のアドレスすなわち&fmt となることから第 2 引数のアドレス＝&fmt から 4 バイト先として求められる．よって，第 n 引数のアドレス＝&fmt から 4 ×（ n − 1 ）バイト先として求められそうである．

5.3.2 疑問 2 の考察

前節で第 2 引数のアドレスを知る方法がわかった．しかし，C 言語で&fmt はポインタとして扱われるため，正確にはアドレスとは異なる．よって C 言語で記述するなら以下のようになる．

```
第 2 引数のアドレス = ((char*)&fmt) + ((sizeof(fmt) + 3) / 4) * 4
```

C 言語で (あるポインタ)+1 が実際のアドレスとしていくつ増えるかはポインタが指す型によって異なる．つまり，あるポインタ p のアドレスが 5000 のとき，p の型が int*の場合は，p+1 は 5004 である．具体的には，5000+sizeof(int) また，p の型が char*の場合は p+1 は 5001 である．この仕組みのおかげで， *(p+1) とした場合に p の型に基づいて，適切なアドレスから正しい値を取り出すことができる．

char*という型は，p+1 がそのままアドレス上で 1 増える．そのため，(char*)&fmt と型を指定することで，値をアドレスと同じように，+1 がそのままアドレスの+1 に相当する操作ができる．第 2 引数は，((sizeof(fmt)+3)/4)*4 バイト分先にあるので， 上記の式になる．

単なる sizeof(fmt) ではないのは，MIPS の gcc では引数の sizeof が 3 以下の場合は，4 の倍数に切り上げるようにメモリを使って引数を配置するので，それを考慮して，((x+3)/4*4) という操作をする必要がある．このようにして利用することで第 2 引数の値をレジスタ$a2 に得ることができる．

```
第 2 引数のアドレス = ((char*)&fmt) + ((sizeof(fmt) + 3) / 4) * 4;
レジスタ$a2 = *(int*) 第 2 引数のアドレス;
```

p2 は char* 型であるので，実際に中身を取り出す場合は，第 2 引数の型の ポインタにキャストしておく必要があります． つまり第 2 引数が int の場合は， a2 を int 型として上記のようになるわけです．同様に，第 3 引数以降も

```
第 3 引数のアドレス  = 第 2 引数のアドレス + ((sizeof(第 2 引数の型) + 3) / 4) * 4;
レジスタ$a3 = *(第 3 引数のポインタ型) 第 3 引数のアドレス;
```

となる

5.4 MIPS における可変引数の実現方法

GCC でどのようにコンパイルされているかマクロを用いて MIPS で調査を行った.

6 2-5

6.0.1 作成したプログラム

6.0.2 考察

6.0.3 テスト結果・評価結果

7 感想