システムプログラミング
中間レポート

学籍番号：09425566
氏名：佐藤 佑太

出題日：2014/11/20
提出日：2014/11/27
締切り日：2014/11/27

1 概要

このレポートでは，以下に与えられる 5 つの課題に関しての考察，またそのプログラムを作成する過程を示すものである．ただし，プログラムは MIPS アセンブリ言語で記述し，SPIM を用いて動作を確認する．

また，課題 2 に関しては第 6 章において考察する．

1. 教科書,（以下教科書はコンピュータの構成と設計（パターソン＆ヘネシー）第四版を指すこととする）B.8 節.「入力と出力」に示されている方法と，B.9 節「システムコール」に示されている方法のそれぞれで，”Hello World”を表示せよ．両者の方式を比較し考察せよ．

2. アセンブリ言語中で使用する .data .text および .align とは何か解説せよ．下記コード中の 6 行目の .data がない場合，どうなるかについて考察せよ．

```
 1:         .text
 2:         .align  2
 3: _print_message:
 4:         la      $a0, msg
 5:         li      $v0, 4
 6:         .data
 7:         .align  2
 8: msg:
 9:         .asciiz "Hello!!\n"
10:         .text
11:         syscall
12:         j       $ra
13: main:
14:         subu    $sp, $sp, 24
15:         sw      $ra, 16($sp)
16:         jal     _print_message
17:         lw      $ra, 16($sp)
18:         addu    $sp, $sp, 24
19:         j       $ra
```

3. 教科書 B.6 節「手続き呼び出し規約」に従って，関数 fact を実装せよ.（以降の課題においては，この規約に全て従うこと）fact を C 言語で記述した場合は，以下のようになる．

```
 1: main()
 2: {
 3:   print_string("The factorial of 10 is ");
 4:   print_int(fact(10));
 5:   print_string("\n");
 6: }
 7:
 8: int fact(int n)
 9: {
10:   if (n < 1)
11:     return 1;
12:   else
13:     return n * fact(n - 1);
14: }
```

4. 素数を最初から 100 番目まで求めて表示する MIPS アセンブリ言語プログラムを作成してテストせよ．その際，素数を求めるために下記の 2 つのルーチンを作成すること．

関数名	概要
prime(n)	n が素数なら 1, そうでなければ 0 を返す
main()	整数を順々に素数判定し，100 個プリント

C 言語を用いた例.

```
 1: int prime(int n)
 2: {
 3:   int i;
 4:   for (i = 2; i < n; i++){
 5:     if (n % i == 0)
 6:       return 0;
 7:   }
 8:   return 1;
 9: }
10:
11: int main()
12: {
13:   int match = 0, n = 2;
14:   while (match < 100){
15:     if (prime(n) == 1){
16:       print_int(n);
17:       print_string(" ");
18:       match++;
19:     }
20:     n++;
21:   }
22:   print_string("\n");
23: }
```

実行例：

```
  2   3   5   7  11  13  17  19  23  29
 31  37  41  43  47  53  59  61  67  71
 73  79  83  89  97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541
```

5. 素数を最初から 100 番目まで求めて表示する MIPS アセンブリ言語プログラムを作成してテストせよ．ただし，配列に実行結果を保存するように main 部分を改造し，ユーザの入力によって任意の番目の配列要素を表示可能にせよ．

   C 言語を用いた例.

```
 1: int primes[100];
 2: int main()
 3: {
 4:   int match = 0, n = 2;
 5:   while (match < 100){
 6:     if (prime(n) == 1){
 7:       primes[match++] = n;
 8:     }
 9:     n++;
10:   }
11:   for (;;){
12:     print_string("> ");
13:     print_int(primes[read_int() - 1]);
14:     print_string("\n");
15:   }
16: }
```

   実行例：

```
> 15
47
> 100
541
```

2　プログラムの作成方針

今回はプログラムが比較的大きくなる課題 3,4,5 について，プログラムを作成していくための方針をここで示す．

2.1　課題 3

課題 3 では，２つの部分に分けて関数を作成する．処理の概要は以下の通りに定め，下記でそれぞれについて解説する．

(1) メインとなる処理を行う main 部

(2) 再帰を用いて与えられた引数の階乗を計算する fact 部

(3) 結果を表示する print 部

まず，(1) メインとなる処理を行う main 部では，fact 関数に求めたい階乗の値を引数と与え，返ってきた数値をコンソールへの出力を行う print 関数に渡す処理を行う．

そして (2) の fact 部では，再帰を用いて与えられた引数の階乗を計算し，それを値とし最終的な値として返す関数を作成する．

(3) では main 関数から与えられた数値をもとにコンソールへの出力を行う．

また，出力は以下のように定める．

```
The factorial of 10 is 3628800
```

2.2　課題 4

課題 4 では，以下の２つの関数を作成する．処理の概要は以下の通りに定め，下記でそれぞれについて解説する

(1) 与えられた引数が素数なら 1，そうでなければ 0 を返す関数 prime(n)

(2) 整数を順々に素数判定し，先頭から 100 個をプリントする main 関数

(1)prime(n) 関数では，与えられた値に対してその数を素数かどうか判定し，引数が素数であれば 1 を，そうでなければ 0 を返す．(2)main 関数では，順に値を prime(n) 関数に与えていきた，100 番目までの素数を数えて素数であるものだけを出力する．

2.3　課題 5

課題 5 では，課題 4 で作成したプログラムの main 部分を改造し，素数を配列に実行しながらかつユーザの入力 n に対して n 番目の配列要素を出力するプログラムを作成する．

また，ユーザが実際に用いることを想定し，簡潔なエラー処理なども行う．

3　プログラムリストおよび、その説明

それぞれの課題について，完成したプログラムを末尾に添付する．このセクションでは，プログラムの主な構造について説明する．

3.1 課題 1

まず，添付したプログラム putc.s で”Hello World”がコンソール上に出力される仕組みについて説明する．

端末装置は レシーバと トランスミッタの２つの独立したユニットで構成され、それぞれの役割は以下となる。

(1) レシーバ：キーバードから入力された文字を読み取る

(2) トランスミッタ：コンソールに文字を表示する。

プログラムを作成する際には，この２つを独立したものとして切り離して考えることが重要である．

これは MIPS の「メモリマップ方式」を利用した表示の仕方である.「メモリマップ方式」とは各レジスタに特定のメモリロケーションが割り当てられていることで、レシーバ制御レジスタ (Receiver Control register) のアドレスは ffff0000(16) で、読み出し専用。ビット 0（1 桁目）を「レディ」と呼び、1 だとキーボードから文字が入力されたけど Reciver Data register からまだ読み出されてない、という意味になる。

Reciever Data register にはキーボードからの入力が格納され、このレジスタからデータが呼び出されるとレディビットは 0 に設定し直される。

これと逆で、トランスミッタ制御レジスタとトランスミッタデータレジスタもそれぞれ反対に今度は出力の作業をするためのレジスタとして割り振られている。

Transmitter Control register はのアドレスは ffff0008(16) であり，このレジスタも下位 2 ビットのみが用いられる．同様にビット 0 が「レディ」と呼ばれ，読み出し専用である．このビットが 1 であると，トランスミッタは出力用の新しい文字を受け取る用意ができているという意味である．このビットが 0 であると，トランスミッタはまだ前の文字を書き出し中である．ビット 1 は「割り込み許可」ビットであり，読み出しも書き出しも可能である．このビットが 1 に設定されると新しい文字の準備ができ，レディビットが 1 であれば端末はハードウェアレベル 1 の割り込みを要求する．

このプログラム作成にあたり特に重要となるポイントは以下である．

1 アドレス 0xffff0008 はトランスミッタ制御に割り当てられている

2 アドレス 0xffff000c はトランスミッタデータに割り当てられている

これらのことを理解しておけばどこのレジスタに書き込みを行った時に t ランスみったがコンソール上に文字を表示してくれるのか把握することができる．

4 行目の main 部分では，各文字を引数と取り扱うレジスタである$a0 に格納している．それを一文字ずつ putc 関数に渡し，表示の作業を任せる．それを一文字ずつ繰り返し，”Hello World” という文字列を最終的にコンソールに表示している．5 行目では，戻りアドレス$ra が putc 関数内で上書きされてしまわないよう，$s レジスタに退避している．

34 行目以降の putc 部分では，35 行目でアドレス 0xffff0008 のデータ，Transmitter Control register のビット 0「レディ」を$t0 にロードしている．

36-38 行目で$t0 の値を調べ,「レディ」が 1 であれば 39 行目でアドレス 0xffff000c のレジスタ，つまり Transmitter Data register に main から受け取った文字のデータを格納する．Transmitter

Data register はコンソールに出力するための部分にデータが格納されたため，その文字をコンソール上に表示する．

次に，OS のサービスを使ってコンソール上に文字列を表示する方法を説明する．

SPIM でオペレーティングシステム的なサービスを実行するためには，システムコール (syscall) という命令を使うことになっている．

syscall を使い，サービスを要求する手順は以下のようにまとめることができる．

1 1. レジスタ$v0 に使いたいサービスのシステムコールコードを格納

2 2. 引数をレジスタ$a0 から$a3（浮動小数点数の値は$f12）にロード

3 3. 値を返すシステムコールは結果をレジスタ$v0（浮動小数点数の値の場合は$f0）に収める

4 4. syscall で実行

サービスを使うためには手順 1 で$v0 に使いたいサービスの対応するデータを int で渡さなければならない．これをシステムコールコードという．(参考文献 1 P.781 参照)

次に，作成したプログラムの説明をする．

1-3 行目で”Hello World”という文字列を，str というラベルをつけて保存している．

5 行目からの main 部分では，上記の手順に従い OS のサービスを要求している．Print の string のシステムコールコードは 4 であり，これを$v0 に格納して syscall を実行すればよい．また，引数は$a0 から$3 にロードする．6 行目が手順 1，7 行目は手順 2，8 行目は手順 3 にあたる．

3.2 課題 3

末尾に添付したプログラム factorial.s について説明する．

まず，再帰的な構造を持つプログラムを作るにあたって、手続き呼び出し規約をしっかりと理解しなければいけない。そのために最初に手続き呼び出し規約に従って処理をする 3 つの局面をおさえておくことにする．

参考文献 2 によると，

> This convention comes into play at three pointes during a procedure call: immediately before the caller invokes the callee, just as the callee starts executing, and immediately before the callee returns to the caller.
>
> (参考文献 2 Location 14781)

とある．まとめると

- 1. 呼び出し側が手続きを呼び出す直前
- 2. 被呼び出し側がスタートした直後
- 3. 被呼び出し側が呼び出し側に戻る直前

の 3 つで規約に従った処理を行わなければならない．

まず，3 行目の main の部分で始めにこれらの規約に従った処理を行う．今回は main が被呼び出し側になっていることに注意したい．被呼び出し側がスタックフレームを生成する過程で最初にすることは以下である．これは最初に言及した局面の 2 にあたる．

Before a called routine starts running, it must take the following steps to set up its stack frame:

1. Allocate memory for the frame by subtracting the frame's size from the stack pointer. 2. Save callee-saved registers in the frame. A callee must save the values in these registers ($s0-$s7, $fp, and $ra) before altering them, since the caller expects to fnd these registers unchanged after the call. Register $fp is saved by every procedure that allocates a new stac frame. However, register $ra only needs to be saved if the callee itself makes a call. The other callee-saved registers that are used also must be saved. 3. Establish the frame pointer by adding the stack frame's size minus 4 to $sp and storing the sum in register $fp. (Location 14787)

コメントのカッコ内の数字は上記 1-3 の処理にあたる．

続いて 10 行目では main から fact 関数を呼び出すため，今度は main が呼び出し側となる．これは上記局面 1 にあたり，呼び出し側は，関数を呼び出す直前に手続き呼び出し規約に従ってスタックフレームを生成しなければならない．

することは以下の 3 つである．

1. Pass arguments. By convention, the first four arguments are passed in registers $a0-$a3. Any remaining arguments are pushed on the stack and appear at the beginning of the called procedure's stack frame. 2. Save caller-saved registers. The called procedure can use these registers($a0-$a3 and $t0-$t9) without first saving their value. If the caller expects to use one of these registers after a call, it must save its value before the call. 3. Execute a jal instruction (see Section 2.8 of Chapter 2), which jumps to the callee's first instruciton and saves the return address in register $ra.

(参考文献 2)

9 行目で階乗を求める 10 を$a0 に入れる。$a0 は引数として用いるレジスタである．fact 関数で使う値を渡していて，手順 1 にあたる．

ここでは呼び出し側で，被呼び出し側で使う値（$a0-$a3 と$t0-$t9) を事前に保存（退避）しておく．これは $t 系のレジスタが被呼び出し側で使われて，変更されたとしても呼び出し側に戻ってきたときにそれを復元してもとの値として使えるようにするためだ．要するに MIPS アセンブリでは「$a と$t 系レジスタは呼び出し間では保存されない」という規約を自分で厳守できるよう設定しなければならない．

そして 10 行目で jal 命令で fact 関数を呼び出している．これは上記手順 3 にあたる．

fact 関数から帰ってきたあと，12-14 行目で main 関数では適切なデータをレジスタに格納した後に printf 関数を呼び，コンソール上に結果を表示している．

最後に，手続き呼び出し規約に従って処理を行う局面 3 があるので，それを実行している．main が呼び出し側に戻る直前である．

局面 3 で行う処理の概要は以下である．

Finally, after pointing the factorial, main returns. But first, it must restore the registers it saved and pop its stack frame.

(参考文献 2)

そして具体的には以下の 4 つの処理を行う.

> 1. If the calle is a function that returns a value, place the returned value in register $v0. 2. Restore all callee-saved registers that were saved upon procedure entry. 3. Pop the stack frame by adding the frame size to $sp 4. Return by jumping to the address in register $ra (Location 14804)
>
> (参考文献 2)

main は返り値がないので 1 の処理は行っていない.

16 行目で lw でスタックに格納（退避）しておいた戻りアドレス$ra とフレームポインタ$fp を元あるべき場所に戻す。(手順 2)

17 行目でスタックフレームを全て元どおり（サイズ 0）にする。$sp に 32 足したら前確保した分が詰まり，元どおりになる。(手順 3)

18 行目で jr で main の値を返してプログラムは終了となる。(手順 4)

3.3 課題 4

3.4 課題 5

4 プログラムの使用例・テスト

このセクションでは，プログラムの使用例を示しながら実際にテストを行う過程を示す.

4.1 課題 1

プログラムを実行すると，Hello World という文字列をコンソール上に以下のように表示する.

```
Hello World
```

4.2 課題 3

プログラムを実行すると，10 の階乗の値をコンソール上に以下のように表示する.

```
The factorial 10 is 3628800
```

4.3 課題 4

プログラムを実行すると，以下のように 1〜100 番目までの素数をコンソール上に表示する.

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163
167 173 179 181 191 193 197 199 211 223 227 229 233 239 241
251 257 263 269 271 277 281 283 293 307 311 313 317 331 337
347 349 353 359 367 373 379 383 389 397 401 409 419 421 431
433 439 443 449 457 461 463 467 479 487 491 499 503 509 521
523 541
```

4.4 課題 5

プログラムを起動すると，まず以下のような簡単な使い方とインターフェースが表示される．

```
Prime Number Searcher

Which number is the prime number you want to know?
- Type 1-100
- To quit, type 0


>
```

ユーザは’ >’ のあとに，知りたい n 番目の素数に対する n を入力する．実際にいくつかの数値を入力してみた際の出力は以下となる．

```
> 2
3
> 100
541
>
```

また，このプログラムではユーザが使用することを想定し，0 でプログラムを正常終了し，100 以上の数値を入力するとエラー文を表示するようにした．

```
> 102

Please type correct number

> 0

Good bye :)
```

5 プログラム作成における考察

各課題において，プログラム作成過程における考察を以下に示す．

5.1 課題 1

プログラムを作成している過程で，以下のエラーが出た．

> Instruction references undefined symbol at 0x00400014

ステップ実行で確認してみたところ，jal 命令において main がないというメッセージが出ていたので main を追加したところ，さらに次のようなエラーが出力された．

> Attempt to execute non-instruction at 0x0040003c

抜け出す処理が必要だと推定し，jr 命令で main から戻る設定を行うようにプログラムを書き直すと，正常に動作するようになった．

以上のことから，main 部分はプログラム開始時に，プログラムの内容に関わらず一番最初に呼ばれる処理が実行されていることがわかる．またこのことから，main 部分に関しても呼び出し側に戻る必要があることもわかる．

また，jal 命令には，実行されると同時に$ra に自動で戻りアドレスを格納する機能があるということも推測できる．

MIPS アセンブリ言語において，戻りアドレスを保持することは非常に重要なことであり，これをプログラマーが任意で値を書き換えないように注意しなければならないだろう．

5.2 課題 3

5.2.1 疑問点 1

なぜスタックフレームを生成するには$sp から値を「引く」のか？

> スタックポインタからフレームのサイズを引いて、フレーム用にメモリを割り当てる
>
> (参考文献 1 p. 764)

スタックフレームは低位から高位のメモリアドレスにフレームを生成することを考えればわかりやすい．$fp は最初初期値が$sp と同じように設定されていて、$sp から引いた数の分だけ$sp と$fp の差は開き、そこがフーレムとして生成されることを考えればよい．

これより，$sp から欲しいバイト分だけ値を引けば，その分下に伸びるイメージで確保領域が広くなっている．そして，$fp はスタックからのデータ取得を簡易化するためのものなので，この原理を知り，4 バイトずつ計算することができれば$sp のみでスタックを利用することもできる．

5.2.2 疑問点 2

なぜスタックフレームの大きさは 24 バイト以上で，8 バイト単位なのか？

> この最低限のフレームに 4 つの引数レジスタ ($a0 - $a3) と戻りアドレス$ra を、倍長語の境界に整列化して持たせることができる。main では$fp も退避する必要があるので、スタックフレームを 2 語分大きくしなければならない。(スタックフレームは倍長語の境界に整列化されることに留意）(参考文献 1 P.766)
>
> The frame is larger than required for these two register because the calling convention requires the minimum size of a stack frame to be 24 bytes. This minimum frame can hold four argument registers ($a0-$a3) and the return address $ra, padded to a double-word boundary (24 bytes). Since main also needs to save $fp, its stack frame must be two words larger (remember: the stack pointer is kept doubleword aligned). (参考文献 2)

$ra と$fa，またコンパイルする際に最低限ひつようであるから，スタックフレームは最低限 24 バイト必要である．

よって，さらにデータを確保したい場合は，32 バイト，40 バイトとスタックフレームのサイズを大きくする必要がある．

これは MIPS が 32 ビットコンピュータとして作動していることに由来している．

整列化，つまり倍長語の 8 バイトずつデータを区切るのは 32 ビットコンピュータがデータの区切りを明確に理解しながらデータを読み込むことでより効率的に処理できるからである．

5.3 課題 4

prime(n) 関数では，今回はチェックしたい値に対して全ての数で割り，その余りを算出することで素数であるかどうかを判定したが，その部分のアルゴリズムはもっと簡略に書けるだろう．今回は時間を確保できなかったので，次の課題として挑戦したい．

5.4 課題 5

今回の課題プログラムの中で唯一ユーザからの入力に対して動作を行うプログラムであったため，ユーザの想定を考えインストラクションとエラー処理，プログラムの正常終了を行うコマンドを実装した．

エラー処理では，100 より大きいあたいを入力すると「Please type correct number」というようなエラー文が表示され，次の処理に移るようプログラムを作成した．

また，0 でプログラムを正常終了させれるようにした．

6 得られた結果に関する，あるいは試問に対する回答

各課題において，得られた結果に関する，あるいは試問に対する回答を以下に示す．また，結果に対してわいたいくつかの疑問点とそれについての考察を記述する．

6.1 課題 2

ピリオドで始まるシンボル名はアセンブラ指令と呼ばれ，これはプログラムを翻訳する方法をアセンブラに指示するものである．つまり，CPU が直接実行するわけではないが，プログラム実行に不可欠なデータの用意や，メモリ中におけるプログラムの配置をコントロールするここことがアセンブラ指令の目的である．

それぞれに意味があり，簡略にまとめると以下のようになる．

シンボル	意味
.text	後続行に命令が含まれることを示す
.data	後続の行にデータが含まれることを示す
.align n	後続行中の項目を 2 の n 乗バイトの境界に合わせて整列化しなければならい
.global main	main をグローバルな記号として宣言
.asciiz	メモリ中に格納する文字列の末尾に null を付加することを示す

特に，.data と.text，また.align n について詳しく考察してみることにする．

まず，.data と.text は，メモリのどこにデータやテキストを格納するかを制御するためのものである．

テキストデータは ASCII table に従った数字として最終的に保存されるため，テキストとしてのデータなのかそれ以外のデータなのか，コンピュータは判断することができない．そこで，アセンブラ指令を用いてもともとテキストとデータを保存する場所を区別するために.data と.text を使い分ける必要があるのだろう．

そのため，.data がない場合，コンピュータは text 型のデータと勘違いしてしまい適切なデー t あセグメントにデータを配置できず，予期せぬ出力を行う可能性が有ると考えられる．

次に，.align n というアセンブラ指令について考察してみる．

MIPS は 32 ビットアーキテクチャである．それはつまり，4 バイト分ずつデータを読み込んでいく，ということだ．このため，CPU が効率的に動くためには，4 バイト分ずつきちんとデータが区切れている必要がある．

データのサイズに合わせて，手動で間を埋めて整列させることもできるが，データの量が多くなると厳しい．そこで，それらの整列化を自動でやってくれるのが.align n というアセンブラ指令である．

6.2 課題 3

手続き呼び出し規約におけるスタックフレームの生成に関して，

> Register $fpissavedbyeveryprocedurethatallocatesanewstacframe.However,registerra$ only needs to be saved if the callee itself makes a call.
>
> (参考文献 2)

という記述に気をつけたい．$fp は被呼び出し側がさらに呼び出し側にならないケースでも必ず退避させなければならない．

また，7 行目で addiu 命令で$fp に格納しているのが$sp+28 であるという部分．ここでは 32 バイト分のスタックフレームを生成しているが，フレームポインタの指し示す部分はそのマイナス 4 バイトである 28 となることでスタックの先頭，一つ目を$fp が指すようにフレームポインタを設定できる．

疑問点 1: 関数が返せる引数はひとつまでなのか

スタックを使えば複数の値を擬似的に返すことはできるかもしれないが，規約には反しているかもしれない．

疑問点 2: なぜ$fp は必ず退避させなければならないのか

呼び出し側がさらに呼び出し側になる場合$ra が退避させる必要はあるのは納得できるが，それ以外の時は退避する必要がないように思える．

これに関しては「後半で$fp を存分に使うのでそこで学べる」とうい回答をいただいた．

疑問点 3: main から fact を呼ぶとき$a0 は退避しないでよいのか

$a0-$a3 は退避する必要がないのか．

7　作成したプログラムのソースコード

それぞれについて，作成したプログラムは以下である．

7.1　課題 1

Listing 1: putc.s

```
1          .text
2          .align 2
3
4  main:
5          move $s0, $ra
6
7          li $a0, 'H'
8          jal putc
9          li $a0, 'e'
10         jal putc
11         li $a0, 'l'
12         jal putc
13         li $a0, 'l'
14         jal putc
15         li $a0, 'o'
16         jal putc
17         li $a0, ' '
18         jal putc
19         li $a0, 'W'
20         jal putc
21         li $a0, 'o'
22         jal putc
23         li $a0, 'r'
24         jal putc
25         li $a0, 'l'
26         jal putc
27         li $a0, 'd'
28         jal putc
29
30         move $ra, $s0
31
32         j $ra
33
34 putc:
35         lw $t0, 0xffff0008 # $t0 = *(0xffff0008)
36         li $t1, 1 # $t1 = 1
37         and $t0, $t0, $t1 # $t0 &= $t1
38         beqz $t0, putc # if ($t0 == 0) goto putc
39         sw $a0, 0xffff000c # *(0xffff000c) = $a
40         j $ra
```

Listing 2: putc2.s

```
1         .data
2  str:
3         .asciiz  "Hello World"
4         .text
5  main:
6         li   $v0, 4  # 手順１：のシステムコールコードは print_string4
7         la   $a0, str # 手順２：で保存したプリントする文字列のアドレスをに格納 asciiz$a0
8         syscall     # 手順４：文字列をプリント（サービス実行！）
9
10   j $ra
```

7.2 課題 3

Listing 3: factorial.s

```
  .text
  .globl main
main:
  subu $sp, $sp, 32 # Length of stack frame: 32 bytes
  sw $ra, 20($sp) # Save return address 戻りアドレスを退避 ()
  sw $fp, 16($sp) # Save old frame pointer 古いフレームポインタを退避 ()
  addiu $fp, $sp, 28 # Set up frame pointer

  li $a0, 10 # Put argument (10) in $a0
  jal fact # Call factorial function

  la $a0, str # Put format string in $a0
  move $a1, $v0 # Move fact result to $a1
  jal printf # Call the print function

  lw $ra, 20($sp) # Restore return address
  lw $fp, 16($sp) # Restore frame pointer
  addiu $sp, $sp, 32 # Pop stack frame
  jr $ra # Return to caller

  .data
str:
  .asciiz "The factorial of 10 is "

  .text
fact:
  subu $sp, $sp, 32 # Stack frame is 32 bytes long
  sw $ra, 20($sp) # Save return address
  sw $fp, 16($sp) # Save frame pointer
  addiu $fp, $sp, 28 # Set up frame pointer
  sw $a0, 0($fp) # Save argument (n)

  lw $v0, 0($fp) # Load n
  bgtz $v0, $L2 # Branch if n > 0
  li $v0, 1 # Return 1
  jr $L1 # Jump to code to return

$L2:
  lw $v1, 0($fp) # Load n
  subu $v0, $v1, 1 # Compute n - 1
  move $a0, $v0 # Move value to $a0
  jal fact # Call factorial function

  lw $v1, 0($fp) # Load n
  mul $v0, $v0, $v1 # Compute fact(n-1) * n

$L1: # Result is in $v0
  lw $ra, 20($sp) # Restore $ra
  lw $fp, 16($sp) # Restore $fp
  addiu $sp, $sp, 32 # Pop stack
  jr $ra

  .text
printf:
  subu $sp, $sp, 32 # Stack frame is 32 bytes long
  sw $ra, 20($sp) # Save return address
  sw $fp, 16($sp) # Save frame pointer
  addiu $fp, $sp, 28 # Set up frame pointer

  li $v0, 4 # syscall of print_string
  syscall # Print format string
  li $v0, 1 # syscal of print_int
  move $a0, $a1 # Move $a1 to $a0 for syscall
  syscall # Print n

  lw $ra, 20($sp) # Restore $ra
```

```
67   lw $fp, 16($sp) # Restore $fp
68   addiu $sp, $sp, 32 # Pop stack
69   jr $ra
```

7.3 課題 4

Listing 4: prime.s

```
1   .text
2   .globl main
3  main:
4   subu $sp, $sp, 32 # Length of stack frame: 32 bytes
5   sw $ra, 20($sp) # Save return address 戻りアドレスを退避 ()
6   sw $fp, 16($sp) # Save old frame pointer 古いフレームポインタを退避 ()
7   addiu $fp, $sp, 28 # Set up frame pointer
8
9   # li $a0, 10 # Put argument (10) in $a0
10  li $a0, 10 # Put argument (10) in $a0
11  jal fact # Call factorial function
12
13  la $a0, str # Put format string in $a0
14  move $a1, $v0 # Move fact result to $a1
15  jal printf # Call the print function
16
17  lw $ra, 20($sp) # Restore return address
18  lw $fp, 16($sp) # Restore frame pointer
19  addiu $sp, $sp, 32 # Pop stack frame
20  jr $ra # Return to caller
21
22  .data
23 str:
24  .asciiz "The factorial of 10 is "
25
26  .text
27 fact:
28  subu $sp, $sp, 32 # Stack frame is 32 bytes long
29  sw $ra, 20($sp) # Save return address
30  sw $fp, 16($sp) # Save frame pointer
31  addiu $fp, $sp, 28 # Set up frame pointer
32  sw $a0, 0($fp) # Save argument (n)
33
34  lw $v0, 0($fp) # Load n
35  bgtz $v0, $L2 # Branch if n > 0
36  li $v0, 1 # Return 1
37  jr $L1 # Jump to code to return
38
39 $L2:
40  lw $v1, 0($fp) # Load n
41  subu $v0, $v1, 1 # Compute n - 1
42  move $a0, $v0 # Move value to $a0
43  jal fact # Call factorial function
44
45  lw $v1, 0($fp) # Load n
46  mul $v0, $v0, $v1 # Compute fact(n-1) * n
47
48 $L1: # Result is in $v0
49  lw $ra, 20($sp) # Restore $ra
50  lw $fp, 16($sp) # Restore $fp
51  addiu $sp, $sp, 32 # Pop stack
52  jr $ra
53
54  .text
55 printf:
56  subu $sp, $sp, 32 # Stack frame is 32 bytes long
57  sw $ra, 20($sp) # Save return address
58  sw $fp, 16($sp) # Save frame pointer
59  addiu $fp, $sp, 28 # Set up frame pointer
```

```
60
61   li $v0, 4 # syscall of print_string
62   syscall # Print format string
63   li $v0, 1 # syscal of print_int
64   move $a0, $a1 # Move $a1 to $a0 for syscall
65   syscall # Print n
66
67   lw $ra, 20($sp) # Restore $ra
68   lw $fp, 16($sp) # Restore $fp
69   addiu $sp, $sp, 32 # Pop stack
70   jr $ra
```

7.4 課題 5

Listing 5: prime2.s

```
1  array:
2    .space 400
3
4    .text
5    .align 2
6  prime:
7    subu $sp, $sp, 32 # Length of stack frame: 32 bytes
8    sw $ra, 20($sp) # Save return address 戻りアドレスを退避 ()
9    sw $fp, 16($sp) # Save old frame pointer 古いフレームポインタを退避 ()
10   addiu $fp, $sp, 28 # Set up frame pointer
11
12   li $t0, 2 # number to loop and divide (i)
13 Loop_prime:
14   beq $t0, $a0, return1 # return1 if n is prime number (i==n)
15   bgt $t0, $a0, Exit_prime # break the loop if n > i
16   rem $t1, $a0, $t0 # $t1 = n % i
17   beqz $t1, Exit_prime # goto Exit_prime if $t1 == 0
18   addi $t0, $t0, 1 # increment i
19   j Loop_prime # loop again with incremented i
20 return1:
21   li $v0, 1 # Prime is true
22   lw $ra, 20($sp) # Restore return address
23   lw $fp, 16($sp) # Restore frame pointer
24   addiu $sp, $sp, 32 # Pop stack frame
25   jr $ra # Return to caller
26 Exit_prime:
27   li $v0, 0 # Prime is false
28   lw $ra, 20($sp) # Restore return address
29   lw $fp, 16($sp) # Restore frame pointer
30   addiu $sp, $sp, 32 # Pop stack frame
31   jr $ra # Return to caller
32
33   .data
34 info:
35   .asciiz "Prime Number Searcher\n\nWhich number is the prime number you want to
         know?\n- Type 1-100\n- To quit, type 0\n\n"
36 mark:
37   .asciiz "\n> "
38 end:
39   .asciiz "\nGood bye :)\n\n"
40 error:
41   .asciiz "\nPlease type correct number\n"
42
43   .text
44   .align 2
45 main:
46   subu $sp, $sp, 32 # Length of stack frame: 32 bytes
47   sw $ra, 20($sp) # Save return address 戻りアドレスを退避 ()
48   sw $fp, 16($sp) # Save old frame pointer 古いフレームポインタを退避 ()
49   addiu $fp, $sp, 28 # Set up frame pointer
```

```
50
51   li $v0, 4 # for syscall of print_string
52   la $a0, info # Introduction
53   syscall # print info
54
55   li $s0, 100 # set maximum number (match number)
56   li $s1, 0 # number of loop
57   li $s3, 2 # number to check and print (n)
58   li $s7, 1 # for comparing
59   la $a1, array # Set array (400 bytes)
60 Loop:
61   beq $s0, $s1, Exit # break if $s1==100
62   move $a0, $s3 # Put argument (2) in $a0
63   jal prime # $v0 = prime
64   bne $v0, $s7, Else # go to Else if prime(n)!=1
65
66   li $t4, 4 # For array increasing
67   addu $a1, $a1, $t4 # $a1 = $a1 + 4
68   sw $s3, 0($a1) # Put prime number into array
69
70   addi $s1, $s1, 1 # increase the number of matched prime number
71 Else:
72   addi $s3, $s3, 1 # n = n + 1
73   j Loop # go to Loop
74 Exit:
75   li $v0, 4 # for syscall of print_string
76   la $a0, mark # Mark of ">"
77   syscall # print info
78   la $a1, array # Initialize $a1
79   li $v0, 5 # For syscall of read_int
80   syscall # read_int for number of prime
81   beqz $v0, End # goto Exit if $v0 == 0
82   bltz $v0, Error # goto Error if $v0 < 0
83   slti $t4, $v0, 101 # Put boolean of $v0 > 100
84   beqz $t4, Error # goto Error if $v0 > 100
85   move $t3, $v0 # Put argument into $t3
86   addu $t3, $t3, $t3 # $t3 = $t3 * 2
87   addu $t3, $t3, $t3 # $t3 = $t3 * 2
88   addu $a1, $a1, $t3 # Add address to fetch
89   lw $a0, 0($a1) # Fetch selected prime
90   li $v0, 1 # For syscall of print_int
91   syscall # Print nth prime
92   j Exit
93
94 Error:
95   li $v0, 4 # for syscall of print_string
96   la $a0, error # Print error message
97   syscall # print info
98   j Exit
99
100
101 End:
102   li $v0, 4 # for syscall of print_string
103   la $a0, end # Mark of ">"
104   syscall # print info
105   lw $ra, 20($sp) # Restore return address
106   lw $fp, 16($sp) # Restore frame pointer
107   addiu $sp, $sp, 32 # Pop stack frame
108   jr $ra # End this program
```

8 参考文献

1. コンピュータの構成と設計 第 4 版 上・下 ジョン・L. ヘネシー (著), デイビッド・A. パターソン (著), 成田 光彰 (翻訳)

2. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design) by David A. Patterson (Author), John L. Hennessy (Author)