

システムプログラミング 1

期末レポート

氏名: 重近 大智 (SHIGECHIKA, Daichi)
学生番号: 09501527

出題日: 2020 年 10 月 05 日
提出日: 2020 年 11 月 14 日
締切日: 2020 年 11 月 16 日

1 概要

本レポートでは, MIPS 言語を用いて, 提示された 5 つの課題に取り組み, その解答を報告する. 実行結果は xspim による結果である.

本レポートで報告するシステムプログラミング 1 の課題は次の 5 つである.

1. A.8 節「入力と出力」に示されている方法と, A.9 節 最後「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. [1]
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. 6.2 節のコード中の 9 行目の `.data` がいない場合, どうなるかについて考察せよ.
3. A.6 節「手続き呼出し規約」に従って, 再帰関数 `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)[1]
4. 素数を最初から 100 番目まで求めて表示する MIPS の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表 1 に示す 2 つのルーチンを作成すること.
5. 素数を最初から 100 番目まで求めて表示する MIPS の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.

2 プログラムの説明

使用した MIPS アセンブリ言語のソースコードは, 6 章に示す.

表 1 課題 4 で実装する 2 つのルーチン

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

2.1 課題 1-1

まず、A.8 節「入力と出力」に示されている方法で実装したコードについて説明する [1]。このプログラムは起動すると、msg ラベルに対応するメモリ上のアドレスを la 命令により、\$a0 レジスタにロードする。次に move 命令を用いて、main を呼び出したコンソールに戻るためのアドレスを \$s0 レジスタに退避する。続いて、loop ラベルから始まるループ部に突入する。ここでは lb 命令を用いて、\$a0 レジスタが指すメモリ上のアドレスに記録されているバイトを \$a1 レジスタにロードする。ロードしたときの値が 0 であった場合、beqz 命令での分岐条件を満たし、end ラベルに対応するメモリ上のアドレスの命令にジャンプする。条件を満たさない場合は、jal 命令で print ラベルにジャンプするとともに、\$ra レジスタに jal print の次の命令が存在するメモリ上のアドレスを代入する。print ラベルから始まる表示の処理では、まず lw 命令を用いて、メモリ上のアドレス 0xffff0008 に存在するワードを \$t0 レジスタにロードする。このとき、\$t0 レジスタに入っている値は 1 かそれ以外であり、これが 1 であればプリンタは動作可能な状態である。li 命令で、即値 1 を \$t1 レジスタにロードして、and 命令で \$t0 レジスタと \$t1 レジスタの論理積をとって \$t0 レジスタに代入する。次の beqz 命令で \$t0 レジスタの値が 0 である場合、プリンタの準備はできていないものとし、もう一度 print ラベルに対応するメモリ上のアドレスの命令に戻って準備を試みる。準備ができてい場合は、sw 命令でメモリ上のアドレス 0xffff000c に表示したい 1 文字の ASCII コードを送る。送った瞬間に SPIM のコンソールに文字が表示される。そして、j \$ra により呼び出し元である main 中の jal print の次の命令にジャンプする。ここまでで 1 文字表示できたので、次の文字情報を得るために addi 命令で \$a0 レジスタの中身に 1 を加え、次に lb 命令でロードする ASCII コードの参照アドレスを変化させる。そして j 命令で loop のラベルにジャンプする。ループを抜けた end ラベル部では、\$s0 に退避していたコンソールに戻るためのアドレスを \$ra から復元し、j 命令でコンソールに戻る。

続いて、A.9 節 最後「システムコール」に示されている方法で実装したコードについて説明する [1]。このプログラムは起動すると、msg ラベルに対応するメモリ上のアドレスを la 命令により、\$a0 レジスタにロードする。次に syscall 命令を実行したときに参照されるシステム・コール・コードを \$v0 レジスタに予めロードしておく。今回のシステム・コール・コードは 4 で、print_string サービスを指定する。syscall 命令でシステムコールを実行し、文字列が出力される。文字列を出力できるため、アドレスを 1 文字ずつずらすような処理は行わない。j 命令でコンソールに戻る。

2.2 課題 1-2

プログラムは、main ラベルから動き始める。まず subu 命令でスタックを 24 ビット確保する。そして、sw 命令でスタックに \$ra レジスタの値をバックアップする。jal 命令で _print_data ラベルに対応するメモリ上のアドレスの命令にジャンプするとともに、\$ra レジスタに自身の次の命令のアドレスを格納する。_print_data ラベルから始まる処理で、la 命令で ghost ラベルに対応するメモリ上のアドレスを \$a0 レジスタにロードする。lw 命令でワードを \$a0 レジスタにロードする。そして li 命令で \$v0 レジスタに即値 1 をロードする。syscall 命令でシステムコールを実行し、コンソールに表示する。j \$ra で main ラベル内の処理に戻り、lw 命令でスタックから \$ra レジスタの値を復元する。そして、addu 命令で \$sp に 24 を加えてスタックを解放する。そして j 命令でコンソールに戻る。

2.3 課題 1-3

まず、move 命令を使って、\$ra レジスタの中身を \$s0 レジスタに退避する。次に la 命令で msg のラベルに対応するメモリ上のアドレスを a0 レジスタにロードする。jal 命令を使ってこの命令の次の命令に対応するメモリ上のアドレスを \$ra レジスタに保存し、ラベル print_str に対応するメモリ上のアドレスの命令にジャンプする。print_str のラベルからの処理では、li 命令で \$v0 レジスタに 4 を代入して、syscall 命令でシステムコールを実行する。j \$ra で、print_str を呼び出したその一つ次の命令に戻る。li 命令で \$a0 レジスタに 10

を代入する。再び `jal` 命令で、この命令の次の命令に対応するメモリ上のアドレスを `$ra` レジスタに保存し、ラベル `fact` に対応するメモリ上のアドレスの命令にジャンプする。ここで `fact` ラベルから始まる処理は、再起呼び出しを受ける可能性のある処理であるから自身を呼び出したときの `$ra` レジスタの値や、再起呼び出しから戻る際に必要となる数値の情報はスタックに保存しておく必要がある。`subu $sp, $sp, 32` から始まる 5 行の処理はスタックに関する命令である。まず、`subu` 命令でスタックポインタが指すメモリ上のアドレスを 32 バイト下げる。これによりスタックが 32 バイト確保された。続いて、`sw` 命令でメモリ上に対応するレジスタの値をコピーする。ここではスタックポインタを基準として 20 番地上の（つまりオフセットが 20）アドレスに `$ra` レジスタのコピー、16 番地上のアドレスに `$fp` レジスタのコピー、24 番地上のアドレスに `$a0` レジスタのコピーを取っている。なお、この `$a0` レジスタは C 言語における `fact()` 関数の引数部に相当する値である。最後に `addu` 命令でフレームポインタを更新している。ここからは `fact` ラベル部の内部処理に移る。まず、`li` 命令で `$t0` レジスタに 1 を代入する。これは後の分岐命令で用いる値である。`bgeu` 命令で `$a0` レジスタの値が `$t0` レジスタより小さいとき、すなわち `$a0` の値が 1 より小さいとき、`else` のラベルに対応するメモリ上のアドレスの命令にジャンプする。そうでない場合は `li` 命令で `$v0` レジスタに 1 を代入する。その後、`lw` 命令でスタックの内容を復元し、スタックを開放し、`j $ra` で呼び出し元に戻る。`else` ラベルにジャンプした場合は、`addi` 命令で `$a0` レジスタに `-1` を加えて（すなわち 1 を引いて）、`jal` 命令で `fact` ラベルにジャンプする。再起呼び出しが終了し、戻ってくると 40 行目から下の `lw` 命令からスタックの復元と解放の処理が始まる。`$ra` レジスタ、`$fp` レジスタ、`$a0` レジスタに対応する値を復元する。そして、`fact` の計算のために、`mulo` 命令で `$v0` レジスタと `$a0` レジスタを乗算する。そして `j $ra` で復元した `$ra` レジスタの情報を元に呼び出し元に戻る。再起呼び出しから戻る過程で、`$v0` レジスタは自身ともう一つのレジスタの値でしか更新されないようになっている。再起呼び出しから完全に戻り、ラベル `main` まで戻ると、`move` 命令で `$v0` レジスタの値を `$a0` レジスタにコピーする。次に実行する `syscall` 命令の引数で `$v0` レジスタの値は上書きされるため、別のレジスタにコピーするとともに実際に表示するときに使う `$a0` レジスタにコピーしている。そして、`jal` 命令を使って、`print_int` ラベルに対応するメモリ上のアドレスの命令にジャンプする。このとき、`$ra` レジスタにこの次の命令のアドレスを格納する。`print_int` のラベルからの処理では、`li` 命令で `$v0` レジスタに 1 を代入して、`syscall` 命令でシステムコールを実行する。`j $ra` で、`print_int` を呼び出したその一つ次の命令に戻る。最後に改行するために `la` 命令で `msg2` のアドレスをロードする。そして `jal print_str` で `print_str` ラベルに対応する命令にジャンプし、`syscall` 命令を実行する。最後に `move` 命令で `$s0` レジスタからコンソールに戻るためのアドレスを `$ra` から復元し、`j` 命令でコンソールに戻る。

2.4 課題 1-4

まず、`move` 命令を使って、`$ra` レジスタの中身を `$s2` レジスタに退避する。続いて、`li` 命令で `$s0` レジスタに 2、`$s1` レジスタに 1 を代入する。このときの `$s0` レジスタの値は検索する数、`$s1` レジスタの値は素数の個数カウントに使用する。`move` 命令を使って、`$s0` レジスタの値を `$a0` レジスタにコピーする。そして、`jal test_prime` で `test_prime` ラベルに対応するメモリ上のアドレスの命令にジャンプするとともに、この次の命令のアドレスを `$ra` レジスタに格納する。`test_prime` ラベルから始まる処理では、まず `li` 命令で即値 2 を `$t0` レジスタにロードする。そして、`move` 命令で `$a0` レジスタの値を `$t1` レジスタにコピーする。ここから先は `prime_loop` ラベルの後の命令となり、ループされる処理である。まず、`rem` 命令で `$t1` レジスタの値を `$t0` レジスタの値で割って、その余りを `$t2` レジスタに代入する。続いて分岐命令 `beq` で、`$t0` レジスタと `$t1` レジスタの値が等しいとき、すなわち素数であるとき、`j` 命令で `prime_r1` ラベルに対応するメモリ上のアドレスの命令にジャンプする。その次の `beqz` 命令は、`$t2` レジスタが 0 のとき、すなわち割り切れて素数ではないとき、`prime_r2` ラベルに対応するメモリ上のアドレスの命令にジャンプする。いずれの条件にも合致しない場合は `addi` 命令で `$t0` レジスタに 1 を加えて、`j prime_loop` でループの初めにジャンプする。`prime_r1` ラベルから始まる処理では、`li` 命令で `$v0` レジスタに即値 1 をロードし、`prime_end` ラベルにジャンプする。`prime_r2` ラベルから始まる処理は、`li` 命令で `$v0` レジスタに即値 0 をロードする。`prime_end` の処理は、`j $ra` により呼び出し元に戻る処理

である。main のループ部に処理が戻ってくると、beqz 命令が実行される。\$v0 レジスタの値が 0 ならば、r1 ラベルに対応するメモリ上のアドレスの命令にジャンプする。r1 ラベル内の処理では、bge 命令で \$s1 レジスタが 100 以上のときに、end ラベルに対応するメモリ上のアドレスの命令にジャンプする。そうでなければ addi 命令で \$s0 レジスタに 1 を加え、j loop でループの初めにジャンプする。r1 に分岐しなかった場合は、数値が素数だったときで、addi 命令で \$s1 レジスタに 1 を加える。そして、move 命令で \$s0 レジスタの値を \$a0 レジスタにコピーする。print_int ラベルと print_str ラベルに対応した処理が行われる。このとき、print_str で表示する文字列は space ラベルに対応するメモリ上のアドレスの文字列である。print_int ラベルと print_str ラベルの処理は、2.3 節で説明したものと同様の処理のため、ここでは説明しない。end ラベルの処理では、la 命令で line ラベルに対応するメモリ上のアドレスを \$a0 レジスタにロードする。そして、print_str で文字列を表示する。最後に move 命令で \$s0 レジスタからコンソールに戻るためのアドレスを \$ra から復元し、j 命令でコンソールに戻る。

2.5 課題 1-5

主に 2.4 節とは異なる処理をしている部分に着目して説明する。

7 行目で la 命令を使い、配列の先頭部分を指すメモリ上のアドレスを \$s3 レジスタにロードする。12 行目から始まる配列へのデータ格納処理は、まず move 命令で \$s1 レジスタの値を \$t0 レジスタにコピーする。次に mulo 命令でこの値を 4 倍する。ここまですオフセットの計算を行う。そして add 命令で \$t0 レジスタと s3 レジスタの値を足して、配列の添字に対応するメモリ上のアドレスに変換する。この値を \$t0 レジスタに代入する。そして sw 命令で \$t0 の値に対応するメモリ上のアドレスに \$s0 の値をコピーする。新たに追加された loop2 ラベルから始まる処理ではユーザからの入力を受け付ける。まず入力を受け付けていることを > により示す。これを表示するために la 命令で indicate ラベルに対応するメモリ上のアドレスを \$a0 レジスタにロードする。次に print_str に対応する処理を行う。そして、jal 命令で read_int のアドレスにジャンプする。read_int ラベルに対応する処理では、li 命令で 5 を代入し syscall 命令でシステムコールを実行する。このとき、入力された整数値は \$v0 レジスタに代入される。j \$ra で呼び出し元である loop2 ラベル内の処理に戻る。bge 命令と ble 命令で例外用の分岐処理を行う。bge 命令は \$v0 レジスタが 101 以上のとき、ble 命令は \$v0 レジスタが 0 以下のとき、end ラベルにジャンプする。そうでなければ \$v0 レジスタの整数値を元に対応するメモリ上のアドレスを計算する。まず、\$v0 レジスタの値を \$t0 レジスタにコピーする。そして、\$t0 レジスタから 1 を引く。次に mulo 命令でこの値を 4 倍する。そして add 命令で \$t0 レジスタと s3 レジスタの値を足して、配列の添字に対応するメモリ上のアドレスに変換する。そして \$t0 レジスタの情報を元にメモリ上の値にアクセスし、lw 命令で \$a0 レジスタに値をロードする。そして print_int に対応する処理を行う。続いて la 命令で line に対応するメモリ上のアドレスを \$a0 レジスタにロードし、print_str に対応する処理を行う。そして j loop2 で loop2 の処理の先頭にジャンプする。素数であれば 1 を返す処理である 45-60 行の部分は、6.4 節に示す課題 1-4 のコードの 26-41 行目と全く同じ処理をするため、ここでは説明しない。end ラベルに対応する処理では、最 move 命令で \$s2 レジスタからコンソールに戻るためのアドレスを \$ra から復元し、j 命令でコンソールに戻る。

3 プログラムの使用法と実行結果

プログラムは、CentOS 7.6.1810 (Core) の xspim で動作を確認している。まず、ターミナルに xspim -mapped_io& と打ち込んで、xspim を実行する。実行後に load の機能を使い、拡張子が .s のアセンブリファイルを読み込む。run の機能で読み込んだプログラムを走らせる。プログラムを走らせた後、もう一度プログラムを走らせる場合には clear でメモリとレジスタの値を初期化した後、再度ロードする必要がある。

3.1 課題 1-1

A.8 節「入力と出力」に示されている方法と、A.9 節 最後「システムコール」に示されている方法のいずれの方法でも SPIM のコンソールの出力は Hello World となることを確認した。

3.2 課題 1-2

6.2 節に示したコードを実行すると、コンソール出力は 12 となる。それに対して、9 行目の `.data` がない場合は、コンソール出力は 1212 となった。表示した後、プログラムは終了する。

3.3 課題 1-3

実行すると、コンソール出力は The factorial of 10 is 3628800 となる。表示した後、プログラムは終了する。

3.4 課題 1-4

実行すると、2 から始まる合計 100 個の素数をコンソールに表示し、終了する。コンソールに表示される素数は次のとおりであり、適宜改行している。実際は 541 の表示後に一度だけ改行する。表示した後、プログラムは終了する。

```
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
```

3.5 課題 1-5

実行すると、コンソールに>が表示され、入力待ちになる。ここに 1-100 の数値を入力できる。入力後に Enter キーを押すと、対応する素数が 1 つ表示される。0 以下か 101 以上の値を入力するとプログラムが終了する。以下に入力と出力の一例を示す。

```
>23
83
>81
419
>0
```

4 考察

4.1 課題 1-1

A.8 節「入力と出力」に示されている方法に示されている方法で実装する場合、空行を除くコードは 25 行である。それに対して、A.9 節 最後「システムコール」に示されている方法で実装する場合、空行を除くコードはたった 12 行となった [1]。これは前者が 1 文字ずつしか表示できないため、1 文字ずつ読み込む処理とアドレス

を 1 加える処理 (12-16 行目) がループになっていることと、プリンタを利用する際にプリンタが使用可能な状態になっていることを確認するために 0xffff0008 番地から値を読み、1 になっているか確認する処理 (18-21 行目) があることに起因する。また jal 命令を用いているため、main を呼び出したときの \$ra レジスタの値が破壊されるため、そのコピーをするために move 命令を使っていることも行数が増える原因の一端となっている。後者は、文字列を表示できるため、アドレスに 1 を加えてずらす処理が必要なく、表示するにあたって特定番地の値の確認を取る必要もない。必要なのは、syscall 命令の動作を規定する引数を \$v0 レジスタに事前に代入するだけである。そのため、非常に簡潔なコードとなっている。

特に「入力と出力」に示されている方法で実装する場合には、0xffff0008 番地の確認を怠ると、プリンタの故障につながることがあったり、計算機によって異なるアドレスをとる可能性のある 0xffff0008 番地をプログラムのコード内に書いてしまうことになったりするなどの問題もある。また、他のプログラムと同時に走行した場合の優先順序をつけられないために競合が発生することがある。システムコールを用いていれば、表示するタイミングや他のプログラムとの優先順序もオペレーティングシステムが決めてくれるため、プログラムを書く際に深く意識する必要はない。

4.2 課題 1-2

.text は、これに後続する項目をテキスト・セグメントの中に格納する指令である [1]。テキスト・セグメントとは、プログラムの命令を格納する領域のことを指す。

.data は、これに後続する項目をデータ・セグメントの中に格納する指令である [1]。データ・セグメントは、テキスト・セグメント上に位置していて、特に静的データ・セグメントは開始アドレス 0x10000000 から始まる領域に格納される。今回の .word 12 をデータ・セグメントに配置するようにした場合、このアドレスの領域に格納されることになる。

.align は、これに後続するデータを 2^n バイト境界に整列化する命令である [1]。MIPS R2000 の多くのロード・ストア命令は、整列化されたデータのみを操作の対象としているため、ワードのデータアドレスは 4 の倍数でなければならない。つまり .align 2 をすれば、 2^2 すなわち 4 バイト境界に整列化することになり、ロード・ストア命令が正しく実行できるようになる。また、SPIM シミュレータは MIPS32 アーキテクチャを実現したプロセッサ用に書かれたアセンブリ言語を実行するシミュレータであるため、ワード単位は 32bit すなわち 4 バイトである。このため、どの命令もデータも 4 バイト境界から始まっておくべきである。

6.2 節のコード中の 9 行目の .data がいない場合、.word 12 はテキスト・セグメント中に配置されることとなり、プログラムの一部となる。ジャンプ命令を実行しない場合は、プログラムカウンタを 4 バイトずつ増やしながら命令を実行していくため、.word 12 も命令であるかのように実行しようとする。MIPS32 アーキテクチャの命令は、4 バイト長であり、.word 12 もバイト長は 4 バイトでそこに 10 進数の 12 があるだけである。同じ長さであるから命令として解釈することも可能になる。数値 12 を命令として解釈すると p.611 の図 A.10.2 では funct(5:0) の syscall が対応する [1]。これは 12 を下位 2 桁と考えることになるためである。これで、.word 12 で syscall 命令を実行し、その後の 15 行目でも syscall 命令を実行するため、2 回 syscall 命令を実行することになる。このため、コンソールの出力が 1212 となった。もし .data を書いていれば、開始アドレス 0x10000000 から始まる静的データ・セグメント領域に格納されるため、プログラムの実行途中にプログラムカウンタに指されることは起こらない。そのため、コンソールの出力は 12 となる。逆を言えば、プログラムカウンタに指されないことを確約できれば、テキスト・セグメント中に格納していても問題にならないとも考えられる。

5 感想

アセンブリ言語を書くのは難しく感じたが、課題に取り組んでいるうちに使い方やプログラムの動きも分かるようになり、楽しかった。課題 1-2 を考察するのが大変だった。

6 作成したプログラムのソースコード

使用したプログラムを以下に添付する.

6.1 課題 1-1 で用いたコード

下記は,「入力と出力」に示されている方法で実装した例.

```
1:      .data                # データセグメント
2:      .align 2             # 4 バイト境界に揃える
3: msg:
4:      .ascii "Hello World" # 出力文字情報
5:
6:      .text                # テキストセグメント
7:      .align 2             # 4 バイト境界に揃える
8: main:
9:      la      $a0, msg      # a0 に msg のアドレスを格納する
10:     move     $s0, $ra      # $s0 に main を呼び出した元のアドレスを格納
11: loop:
12:     lb       $a1, 0($a0)   # a0 の指し先の値を$a1 にロードする
13:     beqz     $a1, end      # a1 が 0 のとき, end へ
14:     jal      print         # print のアドレスにジャンプ (次の命令のアドレスを$ra に)
15:     addi     $a0, 1        # $a0 に 1 を加える
16:     j        loop         # main のアドレスにジャンプ
17: print:
18:     lw       $t0, 0xffff0008 # 0xffff0008 番地にあるワードを$t0 にロードする
19:     li       $t1, 1        # $t1 に 1 を代入する
20:     and      $t0, $t0, $t1  # $t0 と $t1 の論理積をとって$t0 に代入する ($t0 が 1 か確認する)
21:     beqz     $t0, print    # $t0 が 0 のとき, もう一度準備を試みる
22:     sw       $a1, 0xffff000c # 0xffff000c 番地に$a1 にあるワードを送る
23:     j        $ra          # 呼び出し元に戻る
24: end:
25:     move     $ra, $s0      # main を呼び出した元のアドレスを$ra に復元
26:     j        $ra          # コンソールに戻る
```

下記は,「システムコール」に示されている方法で実装した例.

```
1:      .data                # データセグメント
2:      .align 2             # 4 バイト境界に揃える
3: msg:
4:      .ascii "Hello World" # 出力文字情報
5:
6:      .text                # テキストセグメント
7:      .align 2             # 4 バイト境界に揃える
8: main:
9:     la       $a0, msg      # msg のアドレスを $a0 に格納
10:    li       $v0, 4         # print_string を指定
11:    syscall   # システムコールの実行
12:    j        $ra          # コンソールに戻る
```

6.2 課題 1-2 で用いたコード

```
1:      .text                # テキストセグメント
2:      .align 2             # 4 バイト境界に揃える
3:
4: _print_data:
5:     la       $a0, ghost    # ghost のアドレスを$a0 にロード
6:     lw       $a0, 0($a0)   # $a0 レジスタに対応するメモリ上のアドレスのワードを自身にロード
7:     li       $v0, 1        # $v0 レジスタに 1 を代入
8:
9:      .data                # データセグメント
10:     .align 2             # 4 バイト境界に揃える
11: ghost:
```

```

12:      .word 12                # 12 (1 ワード)
13:
14:      .text                   # テキストセグメント
15:      syscall                 # システムコールの実行
16:      j      $ra              # 呼び出し元に戻る
17:
18: main:                        # スタート
19:      subu    $sp, $sp, 24     # スタックを積む
20:      sw      $ra, 16($sp)     # $sp + 16 番地のアドレスに$ra の値をバックアップ
21:      jal     _print_data     # _print_data のアドレスにジャンプ (次の命令のアドレスを$ra に)
22:      lw      $ra, 16($sp)     # $sp + 16 番地のアドレスから$ra の値を復元
23:      addu    $sp, $sp, 24     # スタックを解放
24:      j      $ra              # 呼び出し元に戻る

```

6.3 課題 1-3 で用いたコード

```

1:      .text                   # テキストセグメント
2:      .align 2                # 4 バイト境界に揃える
3: main:
4:      move    $s0, $ra        # main を呼んだ戻り先のアドレスを$s0 に保存しておく
5:
6:      la      $a0, msg         # msg のアドレスを$a0 にロード
7:      jal     print_str       # print_str のアドレスにジャンプ (次の命令のアドレスを$ra に)
8:
9:      li      $a0, 10          # $a0 に 10 を代入 (任意の int 値)
10:
11:     jal      fact             # fact に飛ぶ
12:     move     $a0, $v0         # $a0 に$v0 の値をコピー
13:     jal      print_int       # print_int のアドレスにジャンプ (次の命令のアドレスを$ra に)
14:
15:     la      $a0, msg2        # msg2 のアドレスを$a0 にロード
16:     jal      print_str       # print_str のアドレスにジャンプ (次の命令のアドレスを$ra に)
17:
18:     move     $ra, $s0        # main を呼んだ戻り先のアドレスを$ra に代入
19:     j      $ra              # コンソールに戻る
20:
21: fact:
22:     subu     $sp, $sp, 32     # 32 バイト確保
23:     sw       $ra, 20($sp)     # $sp + 20 のアドレスにもともとの呼出しアドレスを保存
24:     sw       $fp, 16($sp)     # $sp + 16 のアドレスに$fp を保存
25:     sw       $a0, 24($sp)     # $sp + 24 のアドレスに$a0 を保存
26:     addu     $fp, $sp, 28     # 新しく$fp を設定
27:
28:     li      $t0, 1           # $t0 に 1 を代入 (分岐命令用)
29:
30:     bgeu     $a0, $t0, else   # $a0 が$t0 より小さいとき, else へ
31:     li      $v0, 1           # $v0 に 1 を代入
32:     lw       $ra, 20($sp)     # 保存しておいた$sp を復元
33:     lw       $fp, 16($sp)     # 保存しておいた$fp を復元
34:     addu     $sp, $sp, 32     # スタックを解放
35:     j      $ra              # 呼び出し元に戻る
36: else:
37:     addi     $a0, -1          # $a0 から 1 を引く
38:     jal      fact            # 再帰呼び出しをする
39:
40:     lw       $ra, 20($sp)     # 保存しておいた$sp を復元
41:     lw       $fp, 16($sp)     # 保存しておいた$fp を復元
42:     lw       $a0, 24($sp)     # 対応する$a0 の値を復元
43:     addu     $sp, $sp, 32     # スタックを解放
44:
45:     mulo     $v0, $a0, $v0    # $v0 と$a0 をかけて$v0 に代入
46:     j      $ra              # 呼び出し元に戻る
47:
48: print_int:
49:     li      $v0, 1           # $v0 に 1 を代入
50:     syscall                 # システムコールの実行
51:     j      $ra              # 呼び出し元に戻る

```



```

52: print_str:
53:     li      $v0, 4          # $v0 に 4 を代入
54:     syscall          # システムコールの実行
55:     j       $ra            # 呼び出し元に戻る
56:
57:     .data              # データセグメント
58:     .align 2           # 4 バイト境界に揃える
59: msg:
60:     .asciiiz "The factorial of 10 is "
61:
62:     .align 2           # 4 バイト境界に揃える
63: msg2:
64:     .asciiiz "\n"

```

6.4 課題 1-4 で用いたコード

```

1:     .text              # テキストセグメント
2:     .align 2           # 4 バイト境界に揃える
3: main:
4:     move    $s2, $ra    # $s2 に main を呼び出した元のアドレスを格納
5:     li      $s0, 2      # $s0 に 2 を格納 (検索する数)
6:     li      $s1, 0      # $s1 に 0 を格納 (素数の個数カウント)
7: loop:
8:     move    $a0, $s0    # $s0 の値を $a0 にコピー
9:     jal     test_prime  # test_prime のアドレスにジャンプ (次の命令のアドレスを $ra に)
10:    beqz     $v0, r1     # $v0 が 0 ならば, r1 に分岐
11:    addi     $s1, 1      # $s1 に 1 を加える
12:    move     $a0, $s0    # $s0 の値を $a0 にコピー
13:    jal     print_int    # print_int のアドレスにジャンプ (次の命令のアドレスを $ra に)
14:    la       $a0, space  # space のアドレスを $a0 にロード
15:    jal     print_str    # print_str のアドレスにジャンプ (次の命令のアドレスを $ra に)
16: r1:
17:    bge      $s1, 100, end # $s1 が 100 以上のとき, end にジャンプ
18:    addi     $s0, 1      # $s0 に 1 を加える
19:    j        loop       # loop のアドレスにジャンプ
20: end:
21:    la       $a0, line   # line のアドレスを $a0 にロード
22:    jal     print_str    # print_str のアドレスにジャンプ (次の命令のアドレスを $ra に)
23:    move     $ra, $s2    # main を呼び出した元のアドレスを $ra に復元
24:    j        $ra        # コンソールに戻る
25:
26: test_prime:
27:    li      $t0, 2      # $t0 に 2 を代入
28:    move     $t1, $a0    # $t1 に $a0 の値をコピー
29: prime_loop:
30:    rem      $t2, $t1, $t0 # $t1 を $t0 で割った余りを $t2 に代入する
31:    beq      $t0, $t1, prime_r1 # $t0 と $t1 が等しいとき prime_r1 のアドレスにジャンプ
32:    beqz     $t2, prime_r2 # $t2 が 0 なら prime_r2 のアドレスにジャンプ
33:    addi     $t0, 1      # $t0 に 1 を加える
34:    j        prime_loop  # prime_loop のアドレスにジャンプ
35: prime_r1:
36:    li      $v0, 1      # $v0 に 1 を代入
37:    j        prime_end   # prime_end のアドレスにジャンプ
38: prime_r2:
39:    li      $v0, 0      # $v0 に 0 を代入
40: prime_end:
41:    j        $ra        # 呼び出し元に戻る
42:
43: print_int:
44:    li      $v0, 1      # $v0 に 1 を代入
45:    syscall          # システムコールの実行
46:    j       $ra        # 呼び出し元に戻る
47: print_str:
48:    li      $v0, 4      # $v0 に 4 を代入
49:    syscall          # システムコールの実行
50:    j       $ra        # 呼び出し元に戻る
51:

```

```

52:      .data                # データセグメント
53:      .align 2             # 4 バイト境界に揃える
54: space:
55:      .ascii " "
56:      .align 2             # 4 バイト境界に揃える
57: line:
58:      .ascii "\n"

```

6.5 課題 1-5 で用いたコード

```

1:      .text                # テキストセグメント
2:      .align 2             # 4 バイト境界に揃える
3: main:
4:      move    $s2, $ra      # $s2 に main を呼び出した元のアドレスを格納
5:      li      $s0, 2        # $s0 に 2 を格納 (検索する数)
6:      li      $s1, 0        # $s1 に 0 を格納 (素数の個数カウント)
7:      la      $s3, prime_array # $s3 に prime_array のアドレスをロード
8: loop:
9:      move    $a0, $s0      # $s0 の値を $a0 にコピー
10:     jal      test_prime    # test_prime のアドレスにジャンプ (次の命令のアドレスを $ra に)
11:     beqz     $v0, r1       # $v0 が 0 ならば, r1 に分岐
12:     # ここで配列にデータを格納
13:     move     $t0, $s1      # $s1 の値を $t0 にコピー
14:     mulo     $t0, $t0, 4    # $t0 を 4 倍 (4byte 区切り)
15:     add      $t0, $t0, $s3  # $t0 と $s3 を足して $t0 に格納
16:     sw       $s0, 0($t0)    # $s0 の値を配列上にコピー
17:     addi     $s1, 1        # $s1 に 1 を加える
18: r1:
19:     bgt      $s1, 100, loop2 # $s1 が 100 より大きいとき, loop2 にジャンプ
20:     addi     $s0, 1        # $s0 に 1 を加える
21:     j        loop         # loop のアドレスにジャンプ
22: loop2:
23:     la       $a0, indicate  # indicate のアドレスを $a0 にロード
24:     jal      print_str     # print_str のアドレスにジャンプ (次の命令のアドレスを $ra に)
25:     # 入力受付
26:     jal      read_int      # read_int のアドレスにジャンプ (次の命令のアドレスを $ra に)
27:     # 例外確認
28:     bge      $v0, 101, end  # 入力が 101 以上の場合, 終了
29:     ble      $v0, 0, end    # 入力が 0 以下の場合, 終了
30:     # 正しい番地を計算
31:     move     $t0, $v0      # $v0 の値を $t0 にコピー
32:     addi     $t0, -1       # $t0 から 1 を引く
33:     mulo     $t0, $t0, 4    # $t0 を 4 倍 (4byte 区切り)
34:     add      $t0, $t0, $s3  # $t0 と $s3 を足して $t0 に格納
35:     # 配列から読みだした値を表示
36:     lw       $a0, 0($t0)    # $a0 に対応する配列上の値をロード
37:     jal      print_int     # print_int のアドレスにジャンプ (次の命令のアドレスを $ra に)
38:     la       $a0, line     # line のアドレスを $a0 にロード
39:     jal      print_str     # print_str のアドレスにジャンプ (次の命令のアドレスを $ra に)
40:     j        loop2
41: end:
42:     move     $ra, $s2      # main を呼び出した元のアドレスを $ra に復元
43:     j        $ra          # コンソールに戻る
44:
45: test_prime:
46:     li       $t0, 2        # $t0 に 2 を代入
47:     move     $t1, $a0      # $t1 に $a0 の値をコピー
48: prime_loop:
49:     rem      $t2, $t1, $t0  # $t1 を $t0 で割った余りを $t2 に代入する
50:     beq      $t0, $t1, prime_r1 # $t0 と $t1 が等しいとき prime_r1 のアドレスにジャンプ
51:     beqz     $t2, prime_r2  # $t2 が 0 なら prime_r2 のアドレスにジャンプ
52:     addi     $t0, 1        # $t0 に 1 を加える
53:     j        prime_loop    # prime_loop のアドレスにジャンプ
54: prime_r1:
55:     li       $v0, 1        # $v0 に 1 を代入
56:     j        prime_end     # prime_end のアドレスにジャンプ
57: prime_r2:

```

```

58:    li    $v0, 0          # $v0 に 0 を代入
59: prime_end:
60:    j     $ra             # 呼び出し元に戻る
61:
62: print_int:
63:    li    $v0, 1          # $v0 に 1 を代入
64:    syscall              # システムコールの実行
65:    j     $ra             # 呼び出し元に戻る
66: print_str:
67:    li    $v0, 4          # $v0 に 4 を代入
68:    syscall              # システムコールの実行
69:    j     $ra             # 呼び出し元に戻る
70: read_int:
71:    li    $v0, 5          # read_int (戻り値は$v0 に, 1-100 のみ入力可)
72:    syscall              # 読み取り
73:    j     $ra             # 呼び出し元に戻る
74:
75:    .data                # データセグメント
76:    .align 2              # 4 バイト境界に揃える
77: indicate:
78:    .ascii ">"
79:    .align 2              # 4 バイト境界に揃える
80: line:
81:    .ascii "\n"
82:    .align 2              # 4 バイト境界に揃える
83: prime_array:
84:    .space 400

```

参考文献

- [1] David A. Patterson, John L. Hennessy, コンピュータの構成と設計 第 5 版 [下] -ハードウェアとソフトウェア-, 日経 BP 社, 2014.