J4　情報システム工学実験実習 II　実験報告書

題目　　アセンブリ言語プログラミング

実施年月日　2025　年　4　月　11　日

2025　年　4　月　18　日

2025　年　5　月　9　日

提出年月日　2025　年　6　月　20　日

提出者

学籍番号　22126　氏名　西山 颯

実験実習の概要

この実験では、週を追ってアセンブリ言語の理解を深めた。
1 週目は QtSpim を使い、MIPS の基本命令セット(加算、論理、シフト演算)を学んだ 。
2 週目には、分岐命令による繰り返し処理、lw や sw を用いたメモリ上の配列操作、$ra レジスタの退避を伴うサブルーチンの実装に取り組んだ 。
3 週目では、スタックポインタを操作して引数やレジスタを退避・復元することで、より高度な関数、さらには再帰関数の仕組みを実装し、その動作原理を学んだ 。

1. 目的

- 1 週目　命令セットを理解しプログラムを実行できるようになる
- 2 週目　繰り返し処理，配列，サブルーチンを実装できるようになる
- 3 週目　関数と再帰関数を実装できるようになる

2. 実験実習の内容

- 1 週目
  - MIPS アーキテクチャの命令セットを理解する
  - 簡単なプログラムの実行
- 2 週目
  - 繰り返し処理，配列，サブルーチンの実装
- 3 襲名
  - 関数と再帰関数の実装

3. 環境構築の手順

- QtSpim　v9.1.24
  - プログラムを実行するためのシミュレータ[1]
- VS Code　v1.100.2
  - エディタ
- MIPS Support　v0.1.3
  - VS Code の拡張機能であり，MIPS 命令セットの IntelliSense が使用可能[2]

図 01 は簡単なプログラムの例であり，図 02，図 03 は QtSpim でそれを実行したときのスクリーンショットである．

図 01 のプログラムは$t0 レジスタに 1 を代入するだけプログラムである．図 02 の左側はレジスタの一覧，右側は実行される命令の羅列である．$t1 レジスタを確認してみると赤字で 5 となっている．赤字は更新されたレジスタであり，実行後に値が代入されたことが分かる．また，図 03 はレジスタを 2 進数のバイナリ表示している．論理演算をする際にはこれを使用した．

```
1     .globl main
2   main:
3     addi $t0, $zero, 1
4     jr $ra
```

図 01　QtSpim の動作確認用ソースリスト

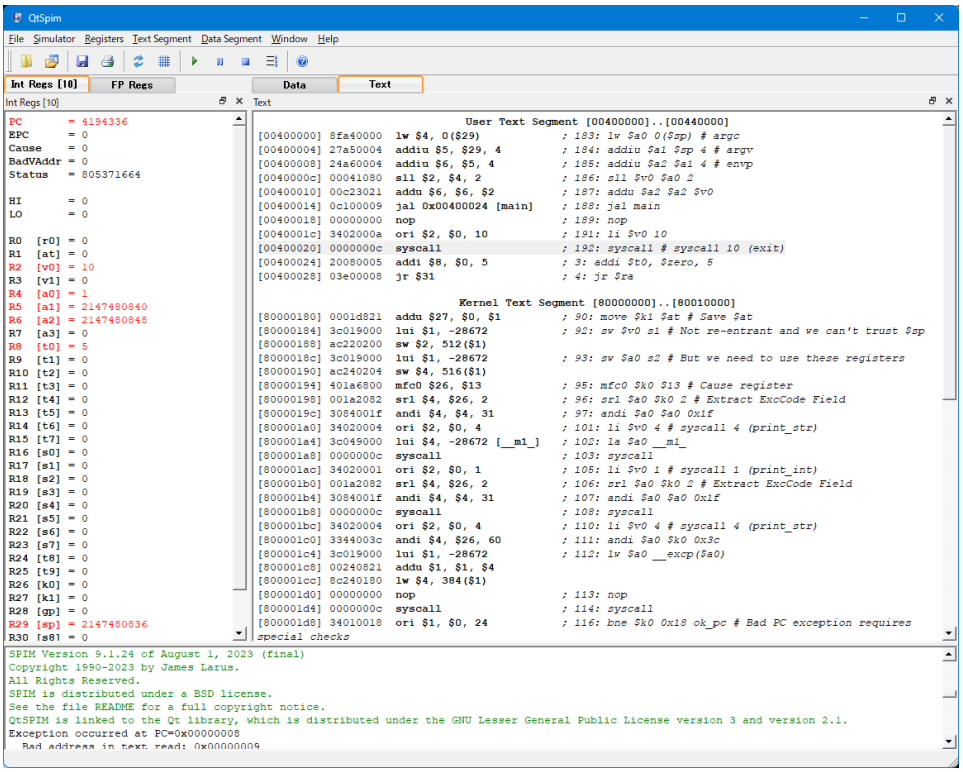


図 02　QtSpim 実行後のスクリーンショット

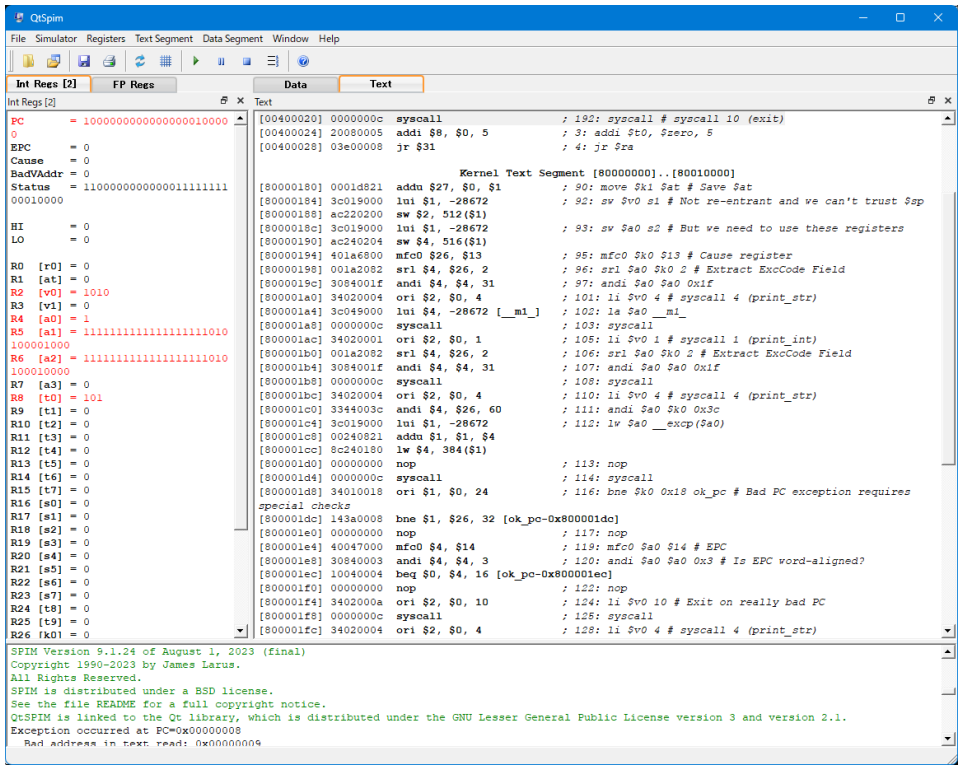


図 03　QtSpim のレジスタを 2 進数にしたスクリーンショット

4. アセンブリ言語プログラミングの演習

4.1 演習 1

1. $t0 レジスタに 100 をセット
2. $1 レジスタに 200 をセット
3. $t0 と$1 の和を$3 に保存

```
1     .globl  main
2   main:
3       addi    $t0, $zero, 100
4       addi    $t1, $zero, 200
5       add     $t3, $t1, $t0
6       jr      $ra
```

図 04　演習 1 のソースリスト

表 01　演習 1 の命令の実行と関係するレジスタの値

PC	命令	$t0	$t1	$t3
4194340	addi $t0, $zero, 100	0	0	0
4194344	addi $t1, $zero, 200	100	0	0
4194348	add $t3, $t1, $t0	100	200	0
4194352	jr $ra	100	200	300

addi 命令は任意の整数と任意のレジスタの和を任意のレジスタに代入する命令である．アドレスの数値を初期化し代入するにはゼロレジスタを使用する必要があり，ソースリスト 3, 4 行目がそれにあたる．

100 と 200 の和は 300 であるため$t3 の 300 は妥当である．

4.2 演習 2

1. $t0 レジスタに 100 をセット
2. $t1 レジスタに-200 をセット
3. $t0 と$t1 の差を$t3 に保存

```
1       .globl  main
2   main:
3       addi    $t0, $zero, 100
4       addi    $t1, $zero, -200
5       sub     $t3, $t0, $t1
6       jr      $ra
```

図 05　演習 2 のソースリスト

表 02　演習 2 の命令の実行と関係するレジスタの値

PC	命令	$t0	$t1	$t3
4194340	addi $t0, $zero, 100	0	0	0
4194344	addi $9, $zero, -200	100	0	0
4194348	sub $t3, $t1, $t0	100	-200	0
4194352	jr $ra	100	-200	300

MIPS アーキテクチャには subi 命令がないため，メモリ同士の引き算のみができる．初めに addi で-200 と初期化し，それらを sub 命令で引くことで差を求めることができる．このとき sub 命令のレジスタの順番に注意する．

100 と-200 の差は 300 であるため，結果は妥当である．

4.3 演習 3

- 4 ワード分のデータ領域を用意し、その中に数値を保存する（値は適当)。1 ワード目のデータを 4 ワード目、3 ワード目のデータを 2 ワード目にコピーするプログラムを作成する。

```
1       .data
2       .globl v
3   v:
4       .word   10
5       .word   20
6       .word   30
7       .word   40
8
9       .text
10      .globl  main
11  main:
12      la      $t2, v
13      lw      $t0, 0($t2)
14      lw      $t1, 8($t2)
15      sw      $t0, 12($t2)
16      sw      $t1, 4($t2)
17      jr      $ra
```

図 06　演習 3 のソースリスト

表 03　演習 3 の命令の実行と関係するレジスタの値

PC	命令	$t0	$t1	$t2
4194340	la $t2, v	0	0	0
4194344	lw $t0, 0($t2)	0	0	268500992
4194348	lw $t1, 8($t2)	10	0	268500992
4194352	sw $t0, 12($t2)	10	30	268500992
4194356	sw $t1, 4($t2)	10	30	268500992
4194360	jr $ra	10	30	268500992

まず初めにソースコード 4~7 行目はデータ部に 4 つのデータを格納している．これはメモリ上に保存されている．

12 行目の la は road address の略で，メモリ上のアドレスを格納する命令である．ここで

はデータ v のアドレス、268500992 が格納された.

13, 14 行目の lw は load word の略でメモリからレジスタへデータを移動する命令である. 1 ワード目と 3 ワード目をコピーする必要があるため, それぞれをロードし, 一時的にレジスタに保存している. ここで t2 レジスタは v のデータ領域のアドレスを指しているに過ぎないため, v の中のデータは相対的なオフセットで指定する必要がある. 1 つのデータは 4 バイトの長さであるため, 1 ワード目は先頭から 0 バイト, 3 ワード目は先頭から 8 バイトのオフセットを指定する必要がある. これはロードとストア, 両方に共通している.

15, 16 の sw は store word の略でレジスタからメモリへデータを移動する命令で, lw と対の命令となる. 1 ワードが入った$t0 を v の 4 ワードに, 3 ワードが入った$t1 を v の 2 ワードに移動, すなわちコピーをしている.

ここで QtSpim の Simulator -> Display Symbol (図 07)を選択すると、画面下のメッセージコンソールにシンボルテーブルが表示される. これはプログラムに関連したデータのアドレスの場所を示しており, 先ほど定義したデータ v も例外ではない.

図 08 で確認すると"g v at 0x10010000"と表示され, v のアドレスは 0x10010000 であることが確認できる(g は global の頭文字である). ここで画面右, データセグメントの User data segment を確認してみると, 同じく 10010000 の表示があり, この右の部分のデータを確認すると 10, 20, 30, 40 の表示がある. これは先のソースコードで定義したものであることは明白である.

このコードを実行した後の画面が図 9 である. 同じ部分を確認すると, 10, 30, 30, 10 であることが確認でき, これは課題 3 の「1 ワード目のデータを 4 ワード目、3 ワード目のデータを 2 ワード目にコピー」が正常に行えたことが分かる.

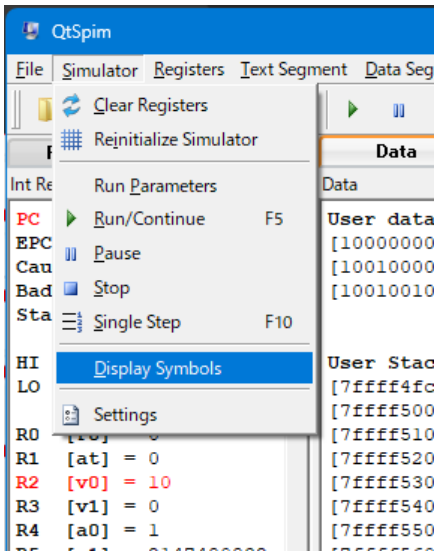


図 07　Display Symbols 表示方法

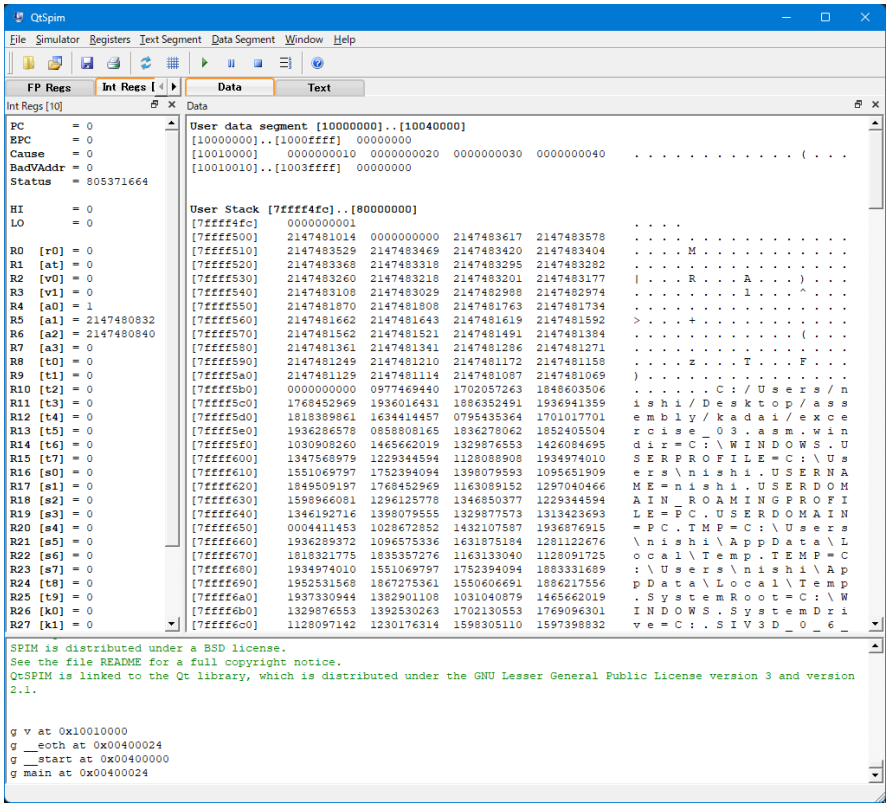


図 08　実行前のメモリ

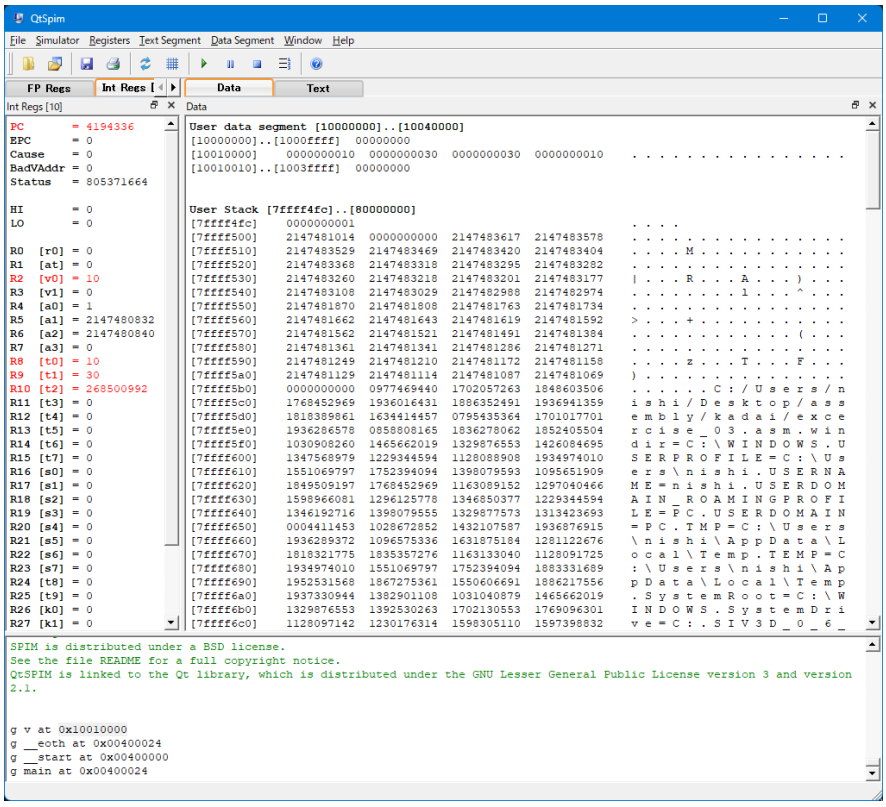


図 09　実行後のメモリ

4.4 演習 4

- not 演算に相当する処理をするプログラムを作成する。$t0 レジスタの値を$t1 レジスタに not 演算を施した結果を保存する。

```
1       .data
2       .text
3
4       .globl  main
5   main:
6       addi    $t0, $zero, 8
7       addi    $t1, $zero, 2
8       nor     $t2, $t0, $t1
9       jr      $ra
```

図 10　演習 4 のソースリスト

表 04　演習 4 の命令の実行と関係するレジスタの値

PC	命令	$t0	$t1	$t2
4194340	addi $t0, $zero, 8	0	0	0
4194344	addi $t1, $zero, 2	1000	0	0
4194348	nor $t2, $t0, $t1	1000	10	0
4194352	jr $ra	1000	10	11111111111111111111111111110101

ここでは説明上，PC 以外のレジスタを二進数で表記し，ゼロ埋めはしていない．

8 行目の nor 命令はレジスタ同士の nor 演算の結果を別のレジスタの格納する命令である。$t0 と$t1 で演算をした結果が$t2 である．$t0 と$t1 の or 演算の結果が 1010，それの not 演算の結果が$t2 になった．1 つのレジスタが 4 バイトで 32 ビットであるため 32 ビット長の結果が表示された．

4.5 演習 5

- シフト演算対象の数値を$t0 レジスタに保存し、$t0 の値を 16 倍して、$結果を$t2 レジスタに保存する。$t0 の値を 1/4 倍して$t3 レジスタに保存するプログラムを作成する。

```
1       .data
2       .text
3
4       .globl  main
5   main:
6       addi    $t0, $zero, 16
7       sll     $t2, $t0, 4
8       srl     $t3, $t0, 2
9       jr      $ra
```

図 11　演習 5 のソースリスト

表 05　演習 3 の命令の実行と関係するレジスタの値

PC	命令	$t0	$t2	$t3
4194340	addi $t0, $zero, 16	0	0	0
4194344	sll $t2, $t0, 4	16	0	0
4194348	srl $t3, $t0, 2	16	256	0
4194352	jr $ra	16	256	4

ソースコード 7, 8 行目の sll, srl はそれぞれレジスタの値を左，右へ論理シフトする命令である．2 進数で処理されているので桁が増えると 2 倍，4 倍，8 倍と 2 の指数倍で値が増加する．また 1 つ桁が減ると 1/2 倍, 1/4 倍, 1/8 倍と 2 の指数の逆数倍で値が減少する(図 12)．これは 10 進数において桁が増えると 10 倍，100 倍, 1000 倍と増えるのと同じようなことである. また, ここではビックエンディアンとして扱われるため, 右シフトで値が増加、左シフトで値が減少する．

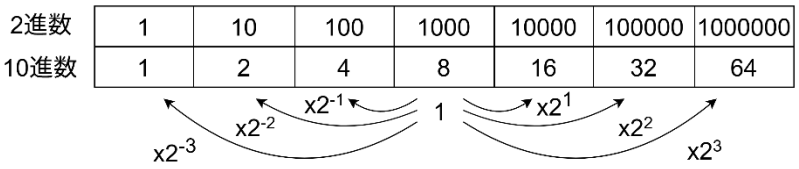


図 12　論理シフトによる数値の増減

4.6 演習 6

- 比較対象の数値を$t0、$t1、$t2 レジスタに保存する。$t0 > $t1 かつ$t0 > $t2 が成立する例を考えて、比較結果を$t3 レジスタに保存するプログラムを作成する。この 2 つの関係が満たされた回数を$t3 に保存する。

```
1       .data
2       .text
3
4       .globl  main
5       main:
6       addi    $t0, $zero, 12
7       addi    $t1, $zero, 9
8       addi    $t2, $zero, 5
9
10      slt     $t3, $t1, $t0
11      slt     $t4, $t2, $t0
12      add     $t3, $t3, $t4
13
14      jr      $ra
```

図 13　演習 6 のソースリスト

表 06　演習 6 の命令の実行と関係する実行前のレジスタの値

$t0	$t1	$t2	$t3	$t4
0	0	0	0	0

表 07　演習 6 の命令の実行と関係する実行後のレジスタの値

$t0	$t1	$t2	$t3	$t4
12	9	5	2	1

ソースコード 10, 11 行目の slt 命令はレジスタ同士を比較した結果を 1, または 0 で表現する命令である．10 行目は$t0 と$t1 を比較し，$t1 のほうが大きいため，$t3 に 1 が格納された．また 11 行目では仮として$t4 に結果を格納し，12 行目でそれらの結果を足し合わせたものが$t4 に格納される．$t0 は$t1, $t2 両方より大きいため，2 が格納された．

4.7 演習 7

- 比較対象の数値を$t0、$t1 に保存する。$t0＜$t1 の場合は、$t1 の値を 1000 加算するプログラムを作成する。

```
1     .data
2     .text
3
4     .globl  addplus
5   addplus:
6     addi    $t1, $t1, 1000
7     jr      $ra
8
9     .globl  main
10  main:
11    addi    $t0, $zero, 4
12    addi    $t1, $zero, 16
13    addi    $t2, $zero, 1
14    slt     $t3, $t0, $t1
15    beq     $t2, $t3, addplus
16    jr      $ra
```

図 14　演習 7 のソースリスト

表 08 演習 7 の命令の実行と関係するレジスタの値

PC	命令	$t0	$t1	$t2	$t3	$ra
4194348	addi $t0, $zero, 4	0	0	0	0	4194328
4194352	addi $t1, $zero, 16	4	0	0	0	4194328
4194356	addi $t2, $zero, 1	4	16	0	0	4194328
4194360	slt $t3, $t0, $t1	4	16	1	0	4194328
4194364	beq $t2, $t3, addplus	4	16	1	1	4194328
4194340	addi $t1, $t1, 1000	4	16	1	1	4194328
4194344	jr $ra	4	1016	1	1	4194328

15 行目は $t2 と$t3 が等しければほかの命令にジャンプするという命令である．ここで$t3 は$t0＜$t1 の判定をしており，1 が格納されている．$t2 には事前に 1 を格納しているため，beq 命令では分岐され，ジャンプされた．beq 命令では戻る先を示す$ra を変更しないため，ジャンプしても$ra は変更されない．そのため，ジャンプ先で jr$ra を実行すると，作成したプログラムは終了する。

4.8 演習 8

- 繰り返し処理を実装して 0 から 5 の数字を$t3 レジスタに順番に格納するプログラムを作成する。

```
1       .data
2       .text
3
4       .globl  main
5   main:
6       add     $t3, $zero, $zero
7       addi    $t1, $zero, 5
8
9   loop:
10      beq     $t3, $t1, proc
11      addi    $t3, $t3, 1
12      j       loop
13
14      .globl proc
15  proc:
16      jr      $ra
```

図 15　演習 8 のソースリスト

表 09 演習 8 の命令の実行と関係するレジスタの値

PC	命令	$t1	$t3
4194340	add $t3, $zero, $zero	0	0
4194344	addi $t1, $zero, 5	0	0
4194348	beq $11, $9, proc	5	0
4194352	addi $t3, $t3, 1	5	0
4194356	j loop	5	1
4194348	beq $11, $9, proc	5	1
4194352	addi $t3, $t3, 1	5	1
4194356	j loop	5	2
4194348	beq $11, $9, proc	5	2
4194352	addi $t3, $t3, 1	5	2
4194356	j loop	5	3
4194348	beq $11, $9, proc	5	3

4194352	addi $t3, $t3, 1	5	3
4194356	j loop	5	4
4194348	beq $11, $9, proc	5	4
4194352	addi $t3, $t3, 1	5	4
4194356	j loop	5	5
4194348	beq $11, $9, proc	5	5
4194360	Jr $ra	5	5

ループ変数を$t3, ループ回数の比較用の変数を$t1 とし, main ブロックで定義している.

main ブロックが終了すると loop ブロックに移動し，ループを開始する．loop ブロックでは初めに$t1 と$t3 が等しいかを判定する．$t1 は 5 で初期化してあるため，$t3 が 5 である場合のみ proc ブロックに移動することが可能となる。次の 11 行目でループ変数をインクリメントし，12 行目で自分自身である loop ブロックへとジャンプする.

また jr は行き先がアドレスで示されている可変であるのに対し, j は静的なコードブロックを指定するため不変である.

4.9 演習 9

- シフト演算対象の数値を$t0 レジスタに保存し、$t1 レジスタで指定されたビット数だけ右シフトする。結果を$t2 レジスタに保存するプログラムを作成する。

```
1     .data
2     .text
3
4     .globl      main
5   main:
6     addi    $t0, $zero, 1024
7     add     $t2, $zero, $t0
8     addi    $t1, $zero, 5
9     add     $t5, $zero, $zero
10
11  loop:
12    beq     $t1, $t5, proc
13    srl     $t2, $t2, 1
14    addi    $t5, $t5, 1
15    j       loop
16
17    .globl      proc
18  proc:
19    jr      $ra
```

図 16　演習 9 のソースリスト

表 10　演習 9 の命令の実行と関係する実行前のレジスタの値

$t0	$t1	$t2	$t5
0	0	0	0

表 11　演習 9 の命令の実行と関係する実行後のレジスタの値

$t0	$t1	$t2	$t5
1024	5	32	5

$t0 は演算対象の数値，$t1 は右シフトする回数兼ループの終了条件，$t2 は計算対象のデータ，$t5 は繰り返し変数として機能している．最初に$t0 の値が$t2 へと保存され，初期化された後は$t2 を使用して計算が行われる．1024 を 5 回右シフトすると，課題 5 より，2^{-5}倍されるということが分かる．1024 の 2^{-5}倍は 32 であるため，これは$t5 と一致する．

4.10　演習 10

- C 言語プログラムの二重ループに相当するプログラムを作成する。下記のプログラムをアセンブリ言語で実装する。プログラム実行終了時の v の値は$t8 レジスタに保存する。

```
1   #include <stdio.h>
2
3   int main(void) {
4       int i = 0;
5       int j = 0;
6       int v = 0;
7       int m = 2;
8       int n = 2;
9
10      for (i = 0; i <= m; i++) {
11          for (j = 0; j <= n; j++) {
12              v = v + 1;
13          }
14      }
15  }
```

図 17　演習 10 の実装前の C 言語ソースリスト

```
1       .data
2       .text
3
4       .globl      main
5   main:
6       add     $t0, $zero, $zero
7       add     $t1, $zero, $zero
8       add     $t8, $zero, $zero
9       addi    $t3, $zero, 3
10      addi    $t4, $zero, 3
11
12  loop1:
13      beq     $t0, $t3, proc
14      addi    $t0, $t0, 1
```

```
15      add     $t1, $zero, $zero
16      j       loop2
17
18  loop2:
19      beq     $t1, $t4, loop1
20      addi    $t1, $t1, 1
21      addi    $t8, $t8, 1
22      j       loop2
23
24      .globl      proc
25  proc:
26      jr      $ra
```

図 18　演習 10 の実装後のアセンブリ言語ソースリスト

表 12　演習 10 の命令の実行と関係するレジスタの値

PC	命令	$t0	$t1	$t3	$t4	$8
省略						
4194376	beq $t1, $t4, loop1	1	0	3	3	0
省略						
4194376	beq $t1, $t4, loop1	1	1	3	3	1
省略						
4194392	jr $ra	3	3	3	3	9

表 13　演習 10 のレジスタと C 言語の対応

レジスタ	$t0	$t1	$t3	$t4	$8
C 言語との対応	i	j	m+1	n+1	v

表 13 はアセンブリソースコードと，課題で示された C 言語のコードの対応を示している．$t0,$t1 は繰り返し変数，$t3,$t4 は繰り返しの回数で，最終的な結果は$8 に格納されており，表 12 からも確認できる．

2 重ループを実現するためにループ用のコードブロックを 2 つ使用した．loop1 ブロックは外側のループ，loop2 ブロックは内側のループを表している，それぞれのブロックの最初には$t0,$t1 の回数を$t3,$t4 を使用し判定し，ループを抜ける処理をしている．また loop2 に入る前には j の繰り返し変数を初期化する必要があるので 0 で初期化をしている(15 行目)．

4.11　演習 11

- メモリ上に配列に相当するデータを準備し、配列の総和を求める処理を実装する。総和の結果は$t9 レジスタに保存する。

```
1     .data
2     .globl      v
3   v:
4     .word   3
5     .word   2
6     .word   1
7     .word   4
8     .word   5
9     .word   8
10    .word   17
11    .word   12
12    .word   10
13    .word   5
14
15    .text
16    .globl  main
17  main:
18    la      $t0, v
19    addi    $t9, $zero, 0
20
21    lw      $t2, 0($t0)
22    add     $t9, $t9, $t2
23    lw      $t2, 4($t0)
24    add     $t9, $t9, $t2
25    lw      $t2, 8($t0)
26    add     $t9, $t9, $t2
27    lw      $t2, 12($t0)
28    add     $t9, $t9, $t2
29    lw      $t2, 16($t0)
30    add     $t9, $t9, $t2
31    lw      $t2, 20($t0)
32    add     $t9, $t9, $t2
```

```
33      lw      $t2, 24($t0)
34      add     $t9, $t9, $t2
35      lw      $t2, 28($t0)
36      add     $t9, $t9, $t2
37      lw      $t2, 32($t0)
38      add     $t9, $t9, $t2
39      lw      $t2, 36($t0)
40      add     $t9, $t9, $t2
41
42      jr      $ra
```

図 19　演習 11 のソースリスト

表 13　演習 11 の命令の実行と関係するレジスタの値

PC	命令	$t0	$t2	$9
4194340	la $t0, v	0	0	0
4194344	addi $t9, $zero, 0	268500992	0	0
4194348	lw $t2, 0($t0)	268500992	0	0
4194352	add $t9, $t9, $t2	268500992	3	0
4194356	lw $t2, 4($t0)	268500992	3	3
4194360	add $t9, $t9, $t2	268500992	2	3
4194364	lw $t2, 8($t0)	268500992	2	5
省略				
4194428	jr $ra	3	5	67

これは演習 3 を応用した内容である．主にロードを中心にデータ領域 v の和を求めている．ロードと加算は同時にできないので，一度$t2 にメモリの内容をコピーした後，$t9 と $t2 の和を$t9 に保存する処理にしている．

最終的な結果は$t9 にある 67 である．実際にデータ領域 v の和を計算すると 67 であるため，適切に計算ができている．

4.12　演習 12

- メモリ上に配列に相当するデータを準備し、配列の総和を求めるサブルーチンを実装する。総和の結果は$t9 レジスタに保存する。

```
1     .data
2
3     .globl v
4   v:  .word   3
5     .word   2
6     .word   1
7     .word   4
8     .word   5
9     .word   8
10    .word   17
11    .word   12
12    .word   10
13    .word   5
14
15
16    .text
17
18    .globl      sum
19  sum:
20    addi    $t1, $zero, 0
21    lw      $t2, 0($t0)
22    add     $t9, $t9, $t2
23    lw      $t2, 4($t0)
24    add     $t9, $t9, $t2
25    lw      $t2, 8($t0)
26    add     $t9, $t9, $t2
27    lw      $t2, 12($t0)
28    add     $t9, $t9, $t2
29    lw      $t2, 16($t0)
30    add     $t9, $t9, $t2
31    lw      $t2, 20($t0)
32    add     $t9, $t9, $t2
```

```
33      lw      $t2, 24($t0)
34      add     $t9, $t9, $t2
35      lw      $t2, 28($t0)
36      add     $t9, $t9, $t2
37      lw      $t2, 32($t0)
38      add     $t9, $t9, $t2
39      lw      $t2, 36($t0)
40      add     $t9, $t9, $t2
41      jr      $ra
42
43      .globl main
44  main:
45      la      $t0, v
46      move    $t3, $ra
47      jal     sum
48      move    $ra, $t3
49      jr      $ra
```

図 20　演習 12 のソースリスト

表 14 演習 12 の命令の実行と関係するレジスタの値

PC	命令	$t0	$t2	$t3	$t9	$ra
4194428	la $t0, v	0	0	0	0	4194328
4194432	move $t3, $ra	268500992	0	0	0	4194328
4194436	jal sum	268500992	0	4194328	0	4194328
4194340	addi $t1, $zero, 0	268500992	0	4194328	0	4194440
4194344	lw $t2, 0($t0)	268500992	0	4194328	0	4194440
4194348	add $t9, $t9, $t2	268500992	3	4194328	0	4194440
4194352	lw $t2, 4($t0)	268500992	3	4194328	3	4194440
省略						
4194424	jr $ra	268500992	5	4194328	67	4194440
4194440	move $ra, $t3	268500992	5	4194328	67	4194440
4194444	jr $ra	268500992	5	4194328	67	4194328

課題 11 のメモリの総和を求めるプログラムを sum ブロックとしてルーチン化したものである．ルーチン中の動作は課題 11 を参照する．

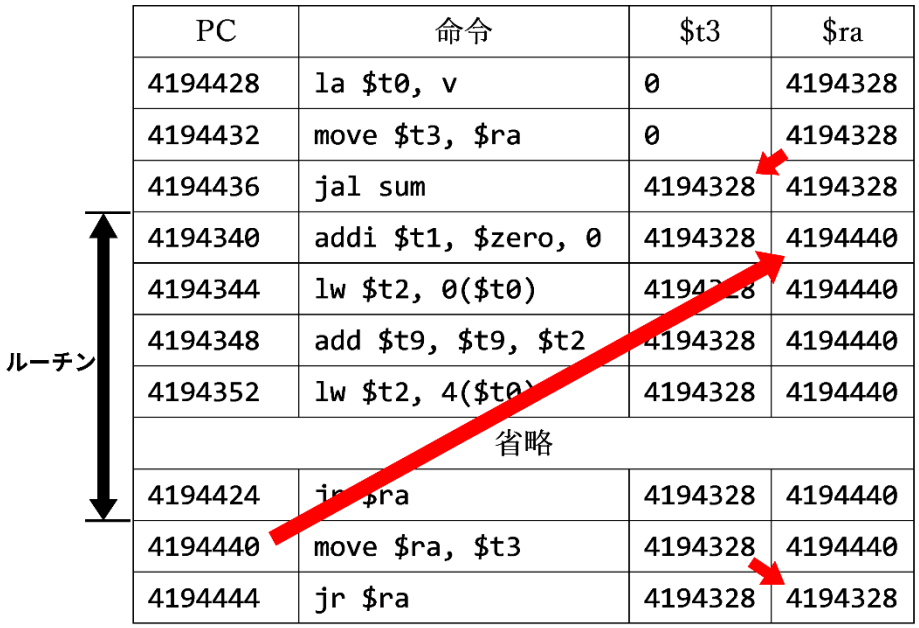
ここで意識しておくべきものが終了用の PC を示す$ra の扱い方である．

ソースコード 47 行目の jal 命令は，ジャンプ先に飛ぶと同時に$ra に jal 命令の次の PC を自動で上書きしてくれる．しかし呼び出し元ではすでにプログラム終了用の$ra が入っているためルーチン終了後プログラムを正常に終了できなくなる．

そこで jal 命令の前に$ra を move 命令で退避させておく必要があり，ここでは$t3 を使用している．そして，ルーチンから戻ってきた後，$t3 から$ra へ再び戻してプログラムを正常に終了させている．

また，$t9 にはメモリの数値の総和である 67 が格納されており，結果は妥当である．

図 21 は PC，PC の移動に関するレジスタ，命令のみのメモリの値の動きのみを示したものである．ルーチンにジャンプする前後にメモリの退避と復元を行っていることが分かる．



PC	命令	$t3	$ra
4194428	la $t0, v	0	4194328
4194432	move $t3, $ra	0	4194328
4194436	jal sum	4194328	4194328
4194340	addi $t1, $zero, 0	4194328	4194440
4194344	lw $t2, 0($t0)	4194328	4194440
4194348	add $t9, $t9, $t2	4194328	4194440
4194352	lw $t2, 4($t0)	4194328	4194440
省略			
4194424	jr $ra	4194328	4194440
4194440	move $ra, $t3	4194328	4194440
4194444	jr $ra	4194328	4194328

図 21　ルーチン前後のアドレスの移動

4.13　演習 13

- メモリ上に配列に相当するデータを準備し、配列の総和を求める関数 sum を実装する。関数の引数は配列の先頭アドレスと個数とする。総和の結果は関数の呼び出し側で$t9 レジスタに保存する。
- 引数
  - $a0 配列のアドレス
  - $a1 配列の大きさ

```
1     .data
2
3     .globl v
4   v:
5     .word   3
6     .word   2
7     .word   1
8     .word   4
9     .word   5
10    .word   8
11    .word   17
12    .word   12
13    .word   10
14    .word   5
15
16    .text
17    .globl      sum
18  sum:
19    addi    $sp, $sp, -12
20    sw      $t0, 0($sp)
21    sw      $t9, 4($sp)
22    sw      $t1, 8($sp)
23
24    move    $t0, $a0
25    add     $t9, $zero, $zero
26
27    lw      $t1, 0($t0)
28    add     $t9, $t9, $t1
```

```
29      lw      $t1, 4($t0)
30      add     $t9, $t9, $t1
31      lw      $t1, 8($t0)
32      add     $t9, $t9, $t1
33      lw      $t1, 12($t0)
34      add     $t9, $t9, $t1
35      lw      $t1, 16($t0)
36      add     $t9, $t9, $t1
37      lw      $t1, 20($t0)
38      add     $t9, $t9, $t1
39      lw      $t1, 24($t0)
40      add     $t9, $t9, $t1
41      lw      $t1, 28($t0)
42      add     $t9, $t9, $t1
43      lw      $t1, 32($t0)
44      add     $t9, $t9, $t1
45      lw      $t1, 36($t0)
46      add     $t9, $t9, $t1
47
48      move    $v0, $t9
49      lw      $t0, 0($sp)
50      lw      $t9, 4($sp)
51      lw      $t1, 8($sp)
52
53      addi    $sp, $sp, 12
54      jr      $ra
55
56      .globl main
57  main:
58      la      $t0, v
59      addi    $t1, $t1, 10
60      move    $a0, $t0
61      move    $a1, $t1
62
63      move    $t5, $ra
64      jal     sum
```

```
65      move    $ra, $t5
66
67      move    $t9, $v0
68      jr      $ra
```

図 22　演習 13 のソースリスト

この課題は課題 12 のルーチンを関数に変更したものといえる．$ra の処理はもちろん，関数にするためにはさらに引数と戻り値を考慮する必要がある．また，関数内で使用されるレジスタ($t0~$t9)を元のブロックで使用されていた値を変更しないように別に退避させておく必要もある．

関数を実装するときに使用するレジスタとして，$a0~$a3 は引数，$v0~$v1 は戻り値，$sp はスタックポインタとして使用する必要がある．中でも$sp はメモリ上にあるスタック領域のアドレスであり，FIFO で保存する．これは関数内で使用するレジスタを保存する際に使用する．

この関数では$t0,$t9,$t1 を使用するため，それらの値をスタック領域へ事前にストア(退避)する必要がある．sum ブロックの 19~22 行目はその処理である．初めに，スタックポインタを保存する要素×4 を引く．これはメモリ空間の使用法によるものである．次の 3 行で確保したメモリ領域にそれぞれの値を保存している．図 23 はそれらストアの様子の模式図である．左側が初めのメモリ空間，右側がストア後のメモリ空間を示している．また，図中のアドレスは一例である．

49~53 行目は先にストアしたレジスタを再びロード(復元)する処理である．この時注意するのはストアしたオフセットとロードするオフセットが一致している必要があることだ．また，押し下げたスタック領域を同じだけ再び戻し，メモリを解放する必要がある．

最終的な結果として，$t9 にメモリの総和である 67 が保存してあるため，結果は妥当である．

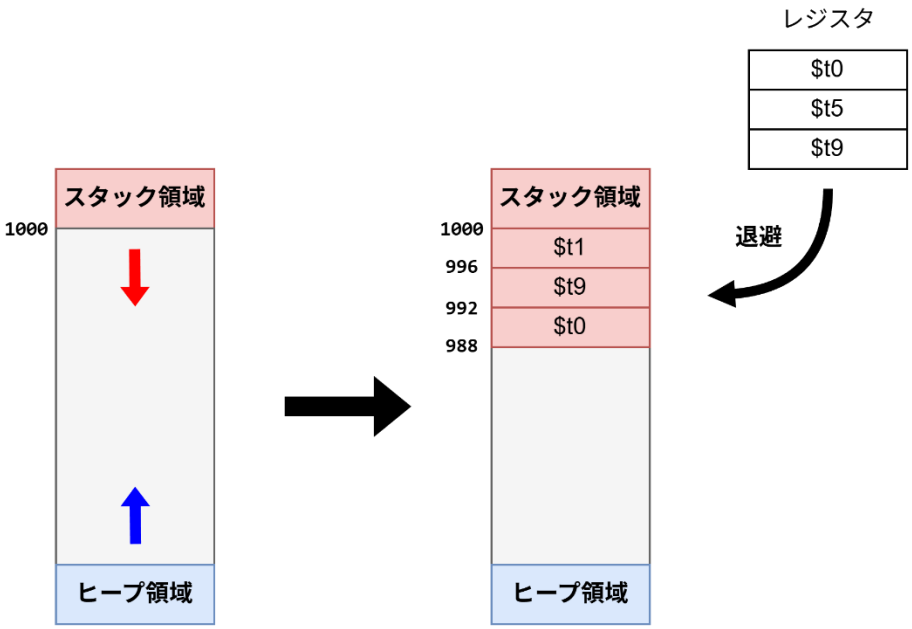


図 23　スタック領域の拡大とレジスタの退避

表 15 演習 13 の命令の実行と関係するレジスタの値

PC	命令	$v0	$a0	$a1	$t0	$t1	$t5	$t9	$sp	$ra
4194468	la $t0, v	4	1	2147480632	0	0	0	0	2147480628	4194328
4194472	addi $t1, $t1, 10	4	1	2147480632	268500992	0	0	0	2147480628	4194328
4194476	move $a0, $t0	4	1	2147480632	268500992	10	0	0	2147480628	4194328
4194480	move $a1, $t1	4	268500992	2147480632	268500992	10	0	0	2147480628	4194328
4194484	move $t5, $ra	4	268500992	10	268500992	10	0	0	2147480628	4194328
4194488	jal sum	4	268500992	10	268500992	10	4194328	0	2147480628	4194328
4194340	addi $sp, $sp, -12	4	268500992	10	268500992	10	4194328	0	2147480628	4194492
4194344	sw $t0, 0($sp)	4	268500992	10	268500992	10	4194328	0	2147480616	4194492
4194348	sw $t9, 4($sp)	4	268500992	10	268500992	10	4194328	0	2147480616	4194492
4194352	sw $t1, 8($sp)	4	268500992	10	268500992	10	4194328	0	2147480616	4194492
4194356	move $t0, $a0	4	268500992	10	268500992	10	4194328	0	2147480616	4194492
4194360	add $t9, $zero, $zero	4	268500992	10	268500992	10	4194328	0	2147480616	4194492
4194364	lw $t1, 0($t0)	4	268500992	10	268500992	10	4194328	0	2147480616	4194492
4194368	add $t9, $t9, $t1	4	268500992	10	268500992	3	4194328	0	2147480616	4194492
省略										
4194444	move $v0, $t9	4	268500992	10	268500992	5	4194328	67	2147480616	4194492
4194448	lw $t0, 0($sp)	67	268500992	10	268500992	5	4194328	67	2147480616	4194492
省略										
4194460	addi $sp, $sp, 8	67	268500992	10	268500992	10	4194328	0	2147480616	4194492
4194464	jr $ra	67	268500992	10	268500992	10	4194328	0	2147480628	4194492

PC	命令	$v0	$a0	$a1	$t0	$t1	$t5	$t9	$sp	$ra
4194492	move $ra, $t5	67	268500992	10	268500992	10	4194328	0	2147480628	4194492
4194496	move $t9, $v0	67	268500992	10	268500992	10	4194328	0	2147480628	4194328
4194500	jr $ra	67	268500992	10	268500992	10	4194328	67	2147480628	4194328

演習 14

- フィボナッチ数列の計算を再帰関数を使って実装する。フィボナッチ数列の C 言語の処理は下記とする。フィボナッチ数列の引数は$t0 レジスタ、演算結果は$t1 レジスタに保存する。実行は n=10 で動作確認する。

```
1   int fibonacci(int n) {
2       int v = 0;
3
4       if (n == 0) {
5           v = 1;
6       } else if (n == 1) {
7           v = 1;
8       } else {
9           v = fibonacci(n - 1) + fibonacci(n - 2);
10      }
11
12      return v;
13  }
```

図 24　演習 14 の実装前の C 言語ソースリスト

```
1   .data
2       .text
3
4       .globl  fib
5   fib:
6       addi    $sp, $sp, -12
7       sw      $ra, 0($sp)
8       sw      $a0, 4($sp)
9
10      slti    $t0, $a0, 2
11      beq     $t0, $zero, L3
12      beq     $a0, $zero, L1
13      j       L2
14
15      .globl L1
16  L1:
```

```
17      addi    $v0, $zero, 0
18      addi    $sp, $sp, 12
19      jr      $ra
20
21      .globl L2
22  L2:
23      addi    $v0, $zero, 1
24      addi    $sp, $sp, 12
25      jr      $ra
26
27      .globl  L3
28  L3:
29      addi    $a0, $a0, -1
30      jal     fib
31      sw      $v0, 8($sp)
32
33      lw      $a0, 4($sp)
34      addi    $a0, $a0, -2
35      jal     fib
36
37      lw      $t1, 8($sp)
38      add     $v0, $v0, $t1
39
40      lw      $ra, 0($sp)
41      lw      $a0, 4($sp)
42      addi    $sp, $sp, 12
43
44      jr      $ra
45
46      .globl main
47  main:
48      li      $t0, 10
49      addi    $sp, $sp, -8
50      sw      $t0  0($sp)
51      sw      $ra, 4($sp)
52
```

```
53      move    $a0, $t0
54      jal     fib
55      move    $t1, $v0
56
57      lw      $t0, 0($sp)
58      lw      $ra, 4($sp)
59      addi    $sp, $sp, 8
60
61      jr      $ra
```

図 25　演習 14 の実装後のアセンブリ言語ソースリスト

表 16　演習 14 の fib(10)呼び出し時スタック領域

[7ffff42c]	0000000010
[7ffff430]	0004194328 0000000001 2147480838 0000000000

表 17　演習 14 の fib(10)呼び出し時スタック領域

[7ffff3b4]	0004194400 0000000001 0000000000
[7ffff3c0]	0004194400 0000000002 0000000001 0004194400
[7ffff3d0]	0000000003 0000000001 0004194400 0000000004
[7ffff3e0]	0000000001 0004194400 0000000005 0000000001
[7ffff3f0]	0004194400 0000000006 0000000001 0004194400
[7ffff400]	0000000007 0000000002 0004194400 0000000008
[7ffff410]	0000000005 0004194400 0000000009 0000000013
[7ffff420]	0004194464 0000000010 0000000034 0000000010
[7ffff430]	0004194328 0000000001 2147480838 0000000000

再帰関数 fib では条件分岐の fib ブロック，2 つのベースケース L1・L2 ブロック，1 つの再帰ケース L3 ブロックで構成されている．

関数 fib は一度の呼び出しで最大 3 つの変数を使用するため，最初に呼び出される fib ブロックの先頭でスタックポインタを 12 バイト分下げている．L3 ブロックでは自分自身の fib 関数を呼び出すため，逐一戻ってくる PC をスタックポインタへ退避させる必要がある．ほかには関数の引数と累計を退避させている．

表 16 は n=0，表 17 は n=10 のときのスタック領域を示している．最もスタック領域に保存されている量が少ないのが n=10 であり，最も多いのが n=0 の時である．ソースコード 7 行目の処理を考えてみる．そのコードが実行されるとき，表 18 の$sp は 16 進数で 7FFFF42C，$a0 は 10 である．これを表 16 で確認すると 10 の値が入っているため，正し

くスタックポインタへのストアが成功していることが分かる．$sp は保存されるごとにどんどん値を減少させていくので，7FFFF42C より若いメモリアドレスに新しいデータが書き込まれて行っている様子が表$で分かる．

また fib(5)が入力されたときの動作を考える．

1. main ブロックで使用されているレジスタをスタック領域にストアする．
2. func 関数で使用するためにスタックポインタを下げ，$a0, $ra の値をスタック領域にストアする．
3. $a0 に 5 が格納されているとき，11 行目の beq 命令によって L3 へジャンプする．
4. 30 行目で$a0 の値を 5-1= 4 に変更したのち，fib(4)が実行される．
5. Fib(4)の結果が$v0 に入っており，スタック領域にストアする．
6. $a0 をスタック領域からロードする．
7. 35 行目で$a0 の値を 5-2=3 に変更したのち，fib(3)が実行される．
8. スタック領域へストアした fib(4)の値を$t1 へ仮保存する．
9. fib(3)の値と fib(4)の値を足す．
10. func 関数で使用したスタック領域から$a0, $ra をロードし，スタックポインタを戻す．
11. main ブロックで使用するためのレジスタをスタック領域からロードする．

最終的な結果が$t1 に 55 が入っている．フィボナッチ数の n=10 は 55 であるため，この計算結果は妥当である．

表 18　演習 14 の命令の実行と関係するレジスタの値

PC	命令	$v0	$a0	$t0	$t1	$sp	$ra
省略							
4194472	jal fib	4	10	10	0	2147480620	4194328
4194340	addi $sp, $sp, -12	4	10	10	0	2147480620	4194464
4194344	sw $ra, 0($sp)	4	10	10	0	2147480608	4194464
省略							
4194476	addi $sp, $sp, 8	55	10	10	55	2147480620	
4194480	jr $ra	55	10	10	55	2147480628	4194328

5. 検討課題

5.1 課題 1

ここではI形式の addi 命令を挙げて説明する.

表 19　I形式 addi 命令の表記表と疑似コード

命令	表記法	疑似コード
addi	addi Rt, Rs, Imm	Rt = Rs + Imm

I形式は即値やデータ転送に使われる形式である. 32bit を図 26 のようなフィールドで分割している. OP は命令, Rs はソース, Rt はターゲット, Imm は Immediate の即値を表している.

これらを組み合わせて機械語表現を考える. ここでは `addi, $v0, $t0, 1000` のコードを考えてみる.

- OP: addi の命令コードは 001000 [%]
- Rs: $v0($2)の 2 進数 00010
- Rt: $t0($8)の 2 進数 01000
- imm: 100 の 2 進数 0000001111101000

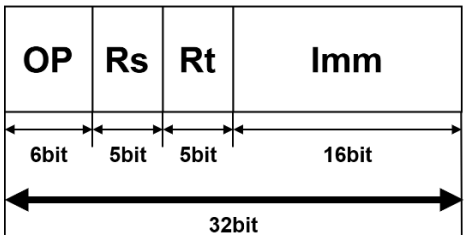


図 26　I形式のフィールド分割

これらを組み合わせると機械語表現は

```
001000 01000 00010 0000001111101000
```

となる. この時ソースコードの Rs と Rt が入れ替わることに注意をする.

imm のフィールドの長さは 16bit であり, 符号付きであるため, 表現できる数は-32,768~32,767 の範囲までということになる.

QtSpim の画面右側, テキストセグメントのアドレスの右隣には機械語の命令が 16 進数で書かれている. 作成した命令に対応する行には 210203e8 の表記があり, 2 進数にすると,

```
0010 0001 0000 0010 0000 0011 1110 1000
```

という結果になった. これは作成した機械語表現と一致するため, 正しく機械語表現が作れたと言える.

5.2 課題 2

ここでは高級言語の中でもインタプリタ系の Python とコンパイルを必要とする C 言語，低級言語のアセンブリ言語の 3 つをそれぞれの観点から比較する．

表 20　プログラミング言語を各観点で比較した点数

	移植性	実行速度	実装の容易さ	保守性	ハードとの親和性
Python	4.0	2.0	4.5	4.5	2.0
C 言語	3.5	4.0	3.5	3.5	3.5
アセンブリ言語	1.0	4.9	1.5	1.0	4.5

表 20 はそれぞれの観点から 5 点満点で考えた点数である．特に特徴的なものを説明する．

実行速度について，Python はインタプリタを使用する点，動的型付けである点などが遅くなる原因であると考えられる[%]．C 言語は静的型付けである点や，CPU にあったコンパイラを使用し，機械語を生成するので実行速度が速くなる．

保守性について，Python は簡単なコードになることが多く，だれが見てもわかりやすいようになっている印象である．C 言語は Python よりかコードが複雑になりがちであり，メモリ操作などが絡むとより難しくなる．アセンブリ言語は最悪で，人間にとって直感的でなく，バグの修正は多くの時間を要す．

実装の容易さについて，これも保守性と間接的につながる．Python はオブジェクト指向言語であり，クラスの概念が使用できるので，より簡潔なロジックでコーディングをすることができる．C 言語は手続き型言語であるため，関数定義やヘッダファイルで乱雑としがちである．アセンブリ言語は関数定義をするのにも引数，戻り値レジスタを逐一意識する必要があり，サブルーチン化するのにも大変である．

ハードとの親和性について，Python はハードウェアが抽象化されており，メモリやレジスタを操作することができない．C 言語は C 言語代表的なメモリ操作が可能である．このことは実行速度の向上にもつながる．アセンブリ言語は先に行った通り，CPU レジスタの操作が基本であり，CPU やメモリと密接にかかわることができる．

6. まとめ

この実験では MIPS アーキテクチャでのアセンブリ言語について，QtSpim を使用したシミュレーションで実験をした．

1 週目は命令セットの概要や，種類，簡単な演習課題を取り組んだ．2 週目から本格的なサブルーチンや繰り返しの実装，メモリの操作など、3 週目はスタックを使用した再帰関数の実装を行った．

レポートにまとめる際には QtSpim のステップ実行でレジスタの値やメモリのアドレスの変化を追跡し，1 つずつ挙動を考え，理解することができた．

検討課題では，行ったシミュレーションでは考えることのできなかった部分まで調べ，吟味することができた．アセンブリ言語という低級な言語からさらに低い 2 進数の状態まで分解して考え，コンピュータの基礎的な部分を体系的に学んだ．

7. 感想

アセンブリ言語について，とてつもなく敷居の高いイメージがあったが，実際に取り組んでみると，意外と簡単で驚いた．おそらく，僕が以前見たアセンブリ言語は MIPS ではなく，もっと難しい構文や英単語で記述されていたものであった気がする．

課題ではやはり課題 14 が段違いで難しかった．レジスタの退避と復元が何回も層になっているため，理解するのも精いっぱいであった．普段使う C 言語も結構な低級言語だと思っていたが，いかに高級で簡単なものであるかが身をもって感じることができた．GCC コンパイラには感謝をしたい．

アセンブリという一番低級で難しい言語を一通り見ることができたため，今後のコーディングはより簡単だという気持ちでいろんな開発に挑めたらと思う．

脚注

ダウンロードサイト

[1]https://spimsimulator.sourceforge.net/

[2] https://marketplace.visualstudio.com/items?itemName=kdarkhan.mips

参考文献

[3]ももうさ，うさぎでもわかる計算機システム　Part21　MIPS アーキテクチャ・命令一覧　前編 | 工業大学生ももやまのうさぎ塾, https://www.momoyama-usagi.com/entry/info-calc-sys21, 2025/06/20 閲覧

[4] OpenCores, Opcodes :: Plasma - most MIPS I(TM) opcodes :: OpenCores, https://opencores.org/projects/plasma/opcodes, 2025/06/20 閲覧

[5] @toaru-05, C 言語と python の実行速度の違い #Python – Qiita, https://qiita.com/toaru-05/items/591e84911d08014d27e9, 2024/12/25 最終更新