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　07 月　14 日

提出者

学籍番号　２２０５９　氏名　来間　空

実験実習の概要

本実験は 3 週にわたり、MIPS アセンブリ言語のプログラミングを段階的に学習した．第 1 週では、QtSPIM シミュレータ環境で、基本的な算術演算，メモリ操作，条件分岐といった命令の動作を確認した．第 2 週では，メモリ上に配列を定義してその総和を求める演習や，jal 命令を用いたサブルーチン，引数と戻り値の規約に基づく関数の実装に取り組んだ．最終週では、スタック領域を操作して戻りアドレスやレジスタ値を退避させることで，フィボナッチ数列を計算する再帰関数を実装した ．

1. 目的

- 1 週目　プログラミング環境の構築や MIPS アーキテクチャの概要や簡単なプログラミングなど，この実験における基本的な考え方を学ぶ.
- 2 週目　繰り返し処理や配列，サブルーチンの実装など，様々な操作が可能になるように学ぶ.
- 3 週目　幅広い操作を学ぶために，関数の実装やプログラムの応用を学ぶ.

2. 実験実習の内容

- 1 週目　プログラミング環境の構築や MIPS アーキテクチャの概要や簡単なプログラミング
- 2 週目　繰り返し処理や配列，サブルーチンの実装
- 3 週目　幅広い操作を学ぶために，関数の実装やプログラムの応用

3. 環境構築の手順

本実習の作業環境として，MIPS の命令を仮想環境で実行するエミュレータである QtSPIM を導入した ．エディタ（Visual Studio Code）で入力したプログラムを QtSPIM で実行した．資料より確認できるバージョンは「SPIM Version 9.1.24 of August 1, 2023」であり，Windows や macOS，Linux で動作するオープンソースソフトウェアである.

この実行環境の簡単なプログラムによる動作確認として，addi $t0, $zero, 1 と addi $t0, $t0, 1 を順に実行するテストプログラムを用いた ．このプログラムを QtSPIM で 1 命令ずつ実行した結果，プログラムカウンタの値が 1 命令実行するごとに 4 ずつ増加し，命令が正しく順次読み込まれていることを確認した ．これにより，基本的な加算命令が意図通りに機能し，後続の演習課題を進めるためのプログラミング環境が正常に動作することが検証された.

4. アセンブリ言語プログラミングの演習

4.1 演習 1

```
1       .globl  main
2   main:
3       addi $t0, $zero, 100
4       addi $t1, $zero, 200
5       add  $t3, $t1, $t0
6       jr $ra
7
8
```

図 1　演習 1 のソースリスト

表 1　命令の実行と関係するレジスタの値

PC	命令	レジスタ $t1	レジスタ $t2	レジスタ $t3
4194344	addi $t0, $zero, 100	100	0	0
4194348	addi $t1, $zero, 200	100	200	0
4194352	addi $t3, $t1, $t0	100	200	300
	jr $ra			

このプログラムの主な目的は，2 つの数値をレジスタと呼ばれるプロセッサ内の高速な記憶領域に設定し，それらを加算して結果を別のレジスタに保存することにある．具体的には，まず`addi`命令を用いてレジスタ`$t0`に 100 を，`$t1`に 200 をそれぞれ代入する．その後，`add`命令でこれら 2 つのレジスタの値を足し合わせ，その和（100＋200）をレジスタ`$t3`に格納する．

表 1 に示された実行結果は，このプログラムの目的が正しく達成されていることを明確に示しており，その内容は完全に妥当である．まず，最初の`addi`命令によって`$t0`の値が期待通り`100`になっている．続く 2 番目の`addi`命令では，`$t0`の値を保持したまま`$t1`の値が`200`となり，これも正しい動作である．最後に，`add`命令が実行されると，`$t0`の`100`と`$t1`の`200`が加算され，その結果である`300`が`$t3`に正しく格納されている．プログラム中で使用されていない`$t2`レジスタの値が終始`0`である点も含め，すべてのレジスタの値の遷移がソースコードの意図と完全に一致している．したがって，この実行結果はプログラムの動作を忠実に反映したものであり，その正しさと妥当性を証明し

ていると言える．

4.2 演習 2

```
1   .globl    main
2   main:
3             addi $t0, $zero, 100
4             addi $t1, $zero, -200
5             sub   $t3, $t1, $t0
6             jr $ra
7
```

図 2　演習 2 のソースリスト

表 2　命令と実行と関係するレジスタの値

PC	命令	レジスタ $t1	レジスタ $t2	レジスタ $t3
4194344	addi $t0, $zero, 100	100	0	0
4194348	addi $t1, $zero, -200	100	-200	0
4194352	sub $t3, $t1, $t0	100	-200	-300
	jr $ra			

このプログラムの具体的な目的は，まず`addi`命令を用いてレジスタ`$t0`に正の数`100`を，レジスタ`$t1`に負の数`-200`をそれぞれ格納することにある．続いて，`sub`命令を用いて`$t1`の値から`$t0`の値を引き算し，その演算結果をレジスタ`$t3`に保存する．

表 2 に示された実行結果の妥当性を検証すると，その値の遷移はソースコードの意図を正確に反映しており，内容は完全に妥当であると結論付けられる．ただし，表のヘッダには「レジスタ $t1」,「レジスタ $t2」と記されているが，プログラムでは`$t0`と`$t1`が使われていること，また値の変動パターンから，これらは「レジスタ $t0」,「レジスタ $t1」の誤記であると解釈するのが合理的である．この前提に立つと，まず`$t0`に`100`が，次に`$t1`に`-200`が正しく格納されていることが確認できる．最終行の`sub`命令では，`-200 - 100`という計算が実行され，その結果である`-300`が`$t3`に正しく格納されており，表の値と完全に一致する．このように，ヘッダの誤記を訂正すれば，この実行結果はプログラムの動作を忠実に反映した，妥当なものであると言える．

4.3 演習 3

```
1   .data
2
3           .globl v
4   v:      .word   1111
5           .word   2222
6           .word   3333
7           .word   4444

            .text

            .globl  main
    main:

            la $t3, v
            lw $t1, 0($t3)
            sw $t1, 12($t3)
            lw $t1, 8($t3)
            sw $t1, 4($t3)

            jr $ra
```

図 3　演習 3 のソースリスト

表 3　命令の実行と関係するレジスタの値

PC	命令	レジスタ $t1	レジスタ $t3
4194340	la $t3, v	0	0
4194344	lw $t1, 0($t3)	0	268500992
4194348	sw $t1, 12($t3)	1111	268500992
4194352	lw $t1, 8($t3)	1111	268500992
4194356	sw $t1, 4($t3)	3333	268500992
4194360	jr $ra	3333	268500992

このプログラムの主な目的は，メモリの特定領域に予め確保されたデータ配列のアドレスを取得し，そのアドレスを基準として，配列の特定要素をレジスタに読み込み（ロード），またレジスタの値を配列の別の要素へ書き込む（ストア）という，一連のメモリアクセス操作を示すことにある．実行結果を示した表は，このプログラムの意図した動作が正しく実行されたことを示しており，その内容は完全に妥当である．まず，ソースコードの`.data`セクションで`v`というラベルのついたメモリ領域に 4 つの整数{1111, 2222, 3333, 4444}が定義されている．プログラムは`la`命令でこの`v`の先頭アドレスをレジスタ`$t3`にロードする．表を見ると`$t3`に具体的なアドレス値`268500992`が格納されており，これが以降のメモリアクセスの基準（ベースアドレス）となる．次に`lw $t1, 0($t3)`命令で，ベースアドレスからオフセット 0 の位置，つまり配列の先頭要素`1111`が`$t1`にロードされる．表で`$t1`が`1111`に変化しており，これは正しい．続く`sw $t1, 12($t3)`では，`$t1`の値`1111`がオフセット 12 の位置，つまり配列の 4 番目の要素に書き込まれる．この時点でレジスタの値は変わらないが，メモリの内容は{1111, 2222, 3333, **1111**}へと変更される．さらに，`lw $t1, 8($t3)`でオフセット 8 の位置から`3333`が`$t1`にロードされ，最後の`sw $t1, 4($t3)`でその`3333`がオフセット 4 の位置に書き込まれる．表に示されたレジスタ`$t1`, `$t3`の値の遷移は，これら一連のロード・ストア命令の実行結果と完全に一致している．したがって，この実行結果はプログラムの動作を忠実に反映した，妥当なものであると言える．

4.4 演習 4

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

図 4　演習 4 のソースリスト

表 4　命令の実行と関係するレジスタの値

PC	命令	レジスタ $t0	レジスタ $t1
4194340	addi $t0, $zero, 5	0	0
4194344	nor　$t1, $t0, $zero	5	0
4194348	jr $ra	5	-6

このプログラムの主な目的は，`NOR`（否定論理和）命令の具体的な動作を示し，特にこの命令を用いてレジスタ内の数値の全ビットを反転させる`NOT`（否定）演算を擬似的に実現する手法を学ぶことにある．実行結果を示した表 4 の内容は，論理演算の定義に基づくと完全に妥当である．プログラムはまず`addi`命令でレジスタ`$t0`に`5`を格納する．次に中心となる`nor $t1, $t0, $zero`命令が実行される．`NOR`は`NOT (A OR B)`という演算であり，この場合`$t1 = NOT ($t0 OR $zero)`となる．`$zero`レジスタの全ビットは常に`0`であるため，`$t0 OR $zero`は`$t0`そのものと等しくなる．結果として，この命令は`$t0`の各ビットを反転して`$t1`に格納する`NOT`演算として機能する．`$t0`の値`5`を 32 ビットの 2 進数で表現すると`0000...0101`となる．この全ビットを反転させると`1111...1010`が得られる．コンピュータの数値表現で広く用いられる 2 の補数表現では，このビットパターンは負の数を意味し，その値は`-6`に相当する．表を見ると，`nor`命令の実行後に`$t1`の値が`-6`となっており，これは理論的な計算結果と完全に一致している．以上の分析から，表 4 に示されたレジスタの値の遷移は，ソースコードの論理演算命令を忠実に実行した結果であり，その動作は妥当である．

4.5 演習 5

```
1  .globl      main
2  main:
3          li $t0 5
4          sll $t2, $t0, 4
5          srl    $t3, $t0, 2
6          jr $ra
7
```

図 5　演習 5 のソースリスト

表 5　命令の実行と関係するレジスタの値

PC	命令	レジスタ $t0	レジスタ $t2	レジスタ $t3
4194340	li $t0 5	0	0	0
4194344	sll $t2, $t0, 4	5	0	0
4194348	srl $t3, $t0, 2	5	80	0
4194352	jr $ra	5	80	1

プログラムの構成を見ると，まず li 命令で基準値 5 をレジスタ$t0 に格納する．次に sll 命令で$t0 を 16 倍した値 80 を$t2 に計算しているが，この$t2 レジスタは後続のどの命令からも参照されないため，この処理は最終結果に影響を与えない「デッドコード」となっている．プログラムの核心は続く srl $t3, $t0, 2 命令である．この命令は$t0 の値 5（2 進数で 0101）を直接参照し，2 ビット右に論理シフトする．これにより下 2 桁が切り捨てられ，結果として 1（2 進数で 0001）が得られる．これは 5÷4 の整数除算の結果に相当する．実行結果を示した表は，この一連の動作を完全に正しく反映しており，その妥当性は非常に高い．sll で$t2 が 80 になる点，そして srl で$t3 が 1 になる点，いずれも計算通りの値である．以上の分析から，このプログラムは途中に結果へ寄与しない命令を含みつつも，演習タイトルに掲げられた「1/á4 倍の実現」という目的を srl 命令によって正しく達成しており，その実行結果も完全に妥当である．

コンピュータのプロセッサレベルで「1/4 倍」の演算を実現するには，除算命令を直接用いるのではなく，より高速で効率的なビット単位のシフト演算を利用するのが一般的な考え方である．これは，コンピュータ内部の数値が 2 進数で表現されているという特性を最大限に活かした手法である．その基本的な原理は，「N ビットの右シフトは 2 N による除算に等しい」というものである．2 進数では，ビットが一つ右にずれると位の重みが半分になるため，1 ビットの右シフトは数値を 1/2 倍する操作に相当する．したがって，「1/4 倍」は 2 2 での除算と同じであるため，対象の数値を 2 ビット右にシフトすることで実現できる．この際，シフトによって押し出された下位ビットは単純に破棄され，これは整数除算における小数点以下の切り捨てに自然に対応する．

MIPS アセンブリ言語における具体的な処理方法としては，右シフト命令である srl または sra を用いる．例えば，レジスタ$t0 の値を 1/4 倍して$t1 に格納する場合，srl $t1, $t0, 2 と記述する．srl は符号を考慮しない論理シフトであり，正の数や符号なし整数の場合に適している．負の数も扱う場合は，符号ビットを維持してシフトする sra（算術右シフト）を用いるのがより正確である．シフト命令は，複雑な除算器を必要とする div 命令に比べて遥かに少ないクロックサイクルで実行できるため，パフォーマンスが大幅に向上する．この最適化は，現代のコンパイラによっても自動的に行われるほど基本的かつ重要な

テクニックであり，コンピュータの効率的な動作を支える根幹的な処理方法の一つであると言える．

4.5 演習 6

```
1   .text
2   .globl main
3   main:
4       # 比較対象の値を設定
5       li $t0, 10        # 比較対象1
6       li $t1, 5         # 比較対象2
7       li $t2, 3         # 比較対象3
8
9       # 比較: $t0 > $t1
10      slt $t4, $t1, $t0       # $t4 = ($t0 > $t1) ? 1 : 0
11
12      # 比較: $t0 > $t2
13      slt $t5, $t2, $t0       # $t5 = ($t0 > $t2) ? 1 : 0
14
15      # 両方満たされていれば AND = 1
16      and $t3, $t4, $t5       # $t3 = ($t4 & $t5)
17
18      # 終了
19      li $v0, 10
20      syscall
```

図 6　演習 6 のソースリスト

表 6　命令の実行と関係するレジスタの値

PC	命令	レジスタ $t0	レジスタ$t1	レジスタ $t2	レジスタ $t3	レジスタ$t4	レジスタ$t5
4194340	li $t0, 10	0	0	0	0	0	0
4194344	li $t1, 5	10	0	0	0	0	0
4194348	li $t2, 3	10	5	0	0	0	0
4194352	slt $t4, $t1, $t0	10	5	3	0	0	0
4194356	slt $t5, $t2, $t0	10	5	3	0	1	0
4194360	and $t3, $t4, $t5	10	5	3	0	1	1
4194364	li $v0, 10	10	5	3	1	1	1
4194368	syscall	10	5	3	1	1	1

具体的な目的は，「ある値（10）が，他の二つの値（5 と 3）の両方よりも大きいか」という，高水準言語における if (A > B && A > C)のような複合条件の真偽を，アセンブリ言語レベルで判定することにある．この目的を達成するため，プログラムは段階的なアプローチを取る．まず，MIPS の slt（set on less than）命令を用いて，個別の大小比較（10 > 5 と 10 > 3）を行う．slt は比較結果を，真であれば 1，偽であれば 0 という数値に変換する重要な役割を担う．次に，これらの数値化された比較結果を and 命令で統合し，全ての条件が満たされているか（全ての比較結果が 1 であるか）を判定する．実行結果の表はこのプロセスを正確に反映しており，その内容は完全に妥当である．$t0 に 10，$t1 に 5，$t2 に 3 が設定された後，slt $t4, $t1, $t0 は 5 < 10 が真であるため$t4 を 1 にし，slt $t5, $t2, $t0 も 3 < 10 が真であるため$t5 を 1 にする．最終的に and $t3, $t4, $t5 は，二つの真（1 と 1）の論理積として$t3 に 1 を格納する．この最終結果 1 は，複合条件全体が真であることを正しく示している．この一連の動作は，より複雑な条件分岐を構成する際の基本的な要素であり，その実行結果は論理的に一貫している．

4.6 演習 7

```
1   .text
2   .globl main
3   main:
4       li $t0, 2
5       li $t1, 4
6       slt $t2, $t0, $t1
7       beq $t2, $zero, skip
8       addi $t1, $t1, 1000
9
10
11
12
13  skip:
14      li $v0, 10
15      syscall
16
```

図 7　演習 7 のソースリスト

表 7　命令の実行と関係するレジスタの値

PC	命令	レジスタ $t0	レジスタ$t1	レジスタ $t2
4194340	li $t0, 2	0	0	0
4194344	li $t1, 4	2	0	0
4194348	slt $t2, $t0, $t1	2	4	0
4194352	beq $t2, $zero, skip	2	4	1
4194356	addi $t1, $t1, 1000	2	1004	1
4194360	li $v0, 10	2	1004	1
4194364	syscall	2	1004	1

このプログラムの目的は，高水準言語における if 文のように,「$t0 が$t1 の値より小さい場合に限り，$t1 の値に 1000 を加算する」という条件付きの処理を実行することにある．この目的を達成するため，プログラムは比較命令 slt と分岐命令 beq を巧みに連携させている．

実行結果の表は，このソースコードの正しい動作を，各ステップにおいて完全に正確に反映しており，その内容は完全に妥当である．まず，$t0 に 2，$t1 に 4 が設定され，slt 命令によって 2＜4 が真と判定されるため，$t2 に 1 が格納される．ここまでの流れは表の通りである．次に，このプログラムの核心である beq $t2, $zero, skip 命令が評価される．$t2 は 1 であるため$zero（0）とは等しくなく，分岐（ジャンプ）は発生しない．したがって，プログラムは直後の addi $t1, $t1, 1000 命令を実行する．実行結果の表では，この addi 命令が実行された直後（PC 4194356）の行で，$t1 の値が 4 から正しい計算結果である 1004 へと更新されていることが確認できる．これはソースコードのロジックが意図通りに動作したことを明確に示している．以上の分析から，このソースコードは指定された目的を達成するための正しいロジックで構成されており，実行結果の表もまた，その正しい動作を忠実にトレースした，完全に妥当なものであると言える．

4.7 演習 8

```
1   .data
2           .text
3
4           .globl      main
5   main:
6           li $t0, 0 #カウンタ
7           li $t1, 5 #終了値
8
9   loop:
10
11          beq    $t0, $t1, proc
12          addi $t3, $t3, 1
13          addi $t0, $t0, 1
14          j loop
15
16          .globl proc
17  proc:   jr $ra
```

図 8　演習 8 のソースリスト

表 8　命令の実行と関係するレジスタの値

PC	命令	レジスタ $t0	レジスタ$t1	レジスタ $t3
4194340	li $t0, 0	0	0	0
4194344	li $t1, 5	0	0	0
4194348	beq $t0, $t1, proc	0	5	0
4194352	addi $t3, $t3, 1	0	5	0
4194356	addi $t0, $t0, 1	0	5	1
4194360	j loop	1	5	1
4194348	beq $t0, $t1, proc	1	5	1
4194352	addi $t3, $t3, 1	1	5	1
4194356	addi $t0, $t0, 1	1	5	2
4194360	j loop	2	5	2
4194348	beq $t0, $t1, proc	2	5	2
4194352	addi $t3, $t3, 1	2	5	2
4194356	addi $t0, $t0, 1	2	5	3
4194360	j loop	3	5	3
4194348	beq $t0, $t1, proc	3	5	3
4194352	addi $t3, $t3, 1	3	5	3
4194356	addi $t0,	3	5	4

	$t0, 1			
4194360	j loop	4	5	4
4194348	beq $t0, $t1, proc	4	5	4
4194352	addi $t3, $t3, 1	4	5	4
4194356	addi $t0, $t0, 1	4	5	5
4194360	j loop	5	5	5
4194348	beq $t0, $t1, proc	5	5	5
4194364	jr $ra	5	5	5

このプログラムの目的は，カウンタとして用いるレジスタ$t0 が，終了値である 5 に達するまで，ループ内の処理を反復実行する．このプログラムのロジックは，ループ構造の基本要素である初期化，条件判定，処理本体，そして更新とジャンプから構成されている．まずカウンタ$t0 を 0 に初期化し，ループの先頭で beq 命令を用いて$t0 が終了値 5 と等しいかを判定する．等しくなければループ本体の処理，すなわち$t3 とカウンタ$t0 の値をそれぞれ 1 ずつ加算する命令が実行され，j（無条件ジャンプ）命令でループの先頭に戻る．このプロセスを$t0 が 5 になるまで繰り返す．実行結果の表は，このループ処理の全過程をステップ・バイ・ステップで完全に正確にトレースしており，その内容は妥当である．表を追うと，$t0 が 0 から 4 の間，ループが計 5 回実行され，それに伴い$t3 の値も（初期値 0 から）5 へと正しく増加していることが確認できる．そして$t0 が 5 に達した最終ループの判定では，beq の条件が成立し，プログラムカウンタがループ内の処理をスキップして proc ラベルへ正しくジャンプしている．この一連のレジスタ値とプログラムフローの遷移は，ソースコードの意図と完全に一致している．したがって，このプログラムとその実行結果は，アセンブリ言語におけるループ実装の正しい例として完全に妥当であると言える．

コンピュータプログラミングにおける繰り返し処理（ループ）は，定型的な作業を自動化するための根幹的な技術である．高水準言語では for や while といった専用構文で記述されるが，アセンブリ言語レベルでは，複数の命令を組み合わせてこの構造を自ら構築する必要がある．その実現には，大きく分けて 4 つの基本要素が不可欠である．

第一に「初期化」である．ループを開始する前に，カウンタとして用いるレジスタ（例：$t0）を li 命令などで特定の値（通常は 0）に設定する．同時に，ループの終了条件となる

値（例：5）も別のレジスタ（例：$t1）に設定しておく．これにより，ループが常に予測可能な状態から始まることが保証される．
第二に「条件判定と分岐」である．ループの先頭で，beq や bne といった分岐命令を用いて，カウンタレジスタが終了条件に達したかを比較する．もし条件が成立していれば，ループ構造の外（例えば proc ラベル）へジャンプし，繰り返しを終了させる．
第三に「処理本体」である．これはループ内で繰り返し実行したい本来の処理であり，算術演算，メモリアクセスなど様々な命令が含まれる．
第四に「更新とジャンプ」である．処理本体の実行後，addi 命令などでカウンタレジスタの値を 1 増やすなどして，終了条件に一歩近づける．そして最後に，j（無条件ジャンプ）命令を用いてループの先頭（条件判定の前）に戻る．このジャンプによって，一連の処理が再び実行され，繰り返しが形成される．
このように，状態を保持するレジスタと，プログラムフローを制御する比較・分岐・ジャンプ命令を巧みに組み合わせることによって，アセンブリ言語においても強力で柔軟な繰り返し処理を実現できるのである．

4.8 演習 9

```
1   ..text
2   .globl main
3   main:
4       li   $t0, 1000
5       li   $t1, 3                 # シフトするビット数
6   （例: 3ビット）
7
8       srlv $t2, $t0, $t1      # $t2 = $t0 >> $t1（論理
9   右シフト）
10
11      li $v0, 10                  # プログラム終了
12      syscall
```

図 9　演習 9 のソースリスト

表 9　命令の実行と関係するレジスタの値

PC	命令	レジスタ $t0	レジスタ$t1	レジスタ $t2
4194340	li $t0, 1000	0	0	0
4194344	li $t1, 3	1000	0	0
4194348	srlv $t2, $t0, $t1	1000	3	0
4194352	li $v0, 10	1000	3	125
4194364	syscall	1000	3	125

このプログラムの目的は、srlv (Shift Right Logical Variable) 命令の動作をデモンストレーションすることにある。まず li 命令を用いて、シフト対象の数値 1000 をレジスタ$t0 に、シフトするビット数 3 をレジスタ$t1 にそれぞれ格納する。そして、核心となる srlv $t2, $t0, $t1 命令で、$t0 の値を$t1 の値だけ右に論理シフトし、その結果を$t2 に保存する。実行結果を示した表は、この一連の処理を完全に正確に反映しており、その内容は完全に妥当である。srlv 命令は、2 N による除算に相当するため、3 ビットの右シフトは 2 3 =8 による整数除算と等価である。したがって、プロセッサが実行する計算は 1000 ÷ 8 となり、その結果は 125 となる。表を見ると、srlv 命令の実行直後（PC 4194348）において、$t2 の値がまさに 125 となっており、これは理論的な計算結果と完全に一致している。以上の分析から、このソースコードは可変シフト命令 srlv の正しい使い方を示しており、実行結果の表もまた、その動作を忠実にトレースした、完全に妥当なものであると言える。この srlv のような命令は、プログラムの実行状況に応じて操作対象のビット位置を変えたい場合など、実用的な場面で極めて有効な手段となる。

4.9 演習 10

```
1    .globl main
2    main:
3        add $t0, $zero, $zero        #i
4        add $t8, $zero, $zero        #v
5        addi $t3, $zero, 2           #m
6        addi $t4, $zero, 2           #n
```

```
7
8   loop_1:
9       beq $t0, $t3, proc
10      add $t1, $zero, $zero
11
12      loop_2:
13          beq $t1, $t4, next
14          addi $t8, $t8, 1
15          addi $t1, $t1, 1          #j
16          j loop_2
17
18  next:
19      addi $t0, $t0, 1
20      jr loop_1
21
22  .globl proc
23  proc:
24      jr $ra
```

図 10　演習１０のコードリスト

表 10　命令の実行と関係するレジスタの値

PC	命令	$t0	$t1	$t2	$t3	$t4	$t8
Loop_2	i=0.j=0の判定時	0	0	0	2	2	0
4194364	beq $t1, $t4, next	0	0	0	2	2	0
4194368	addi $t8, $t8, 1	0	0	0	2	2	1
4194372	addi $t1, $t1, 1	0	1	0	2	2	1
4194376	j loop_2	0	1	0	2	2	1
Loop_2	i=0,j=1の判定時	0	1	0	2	2	1
4194364	beq $t1, $t4, next	0	1	0	2	2	1

4194368	addi $t8, $t8, 1	0	1	0	2	2	2
4194372	addi $t1, $t1, 1	0	2	0	2	2	2
中略	中略						
4194384	jr loop_1	2	2	0	2	2	4
Loop_1	i=2,j=2の判定後	2	2	0	2	2	4
4194356	beq $t0, $t3, proc	2	2	0	2	2	4
4194388	jr $ra	2	2	0	2	2	4

このアセンブリプログラムは，二重ループ（ネストしたループ）構造を実装し，その総実行回数を計算して結果をレジスタ`$t8`に格納することを目的とする．具体的には，C 言語の `for (i=0; i<2; i++) { for (j=0; j<2; j++) { ... } }` というような処理を模倣し，2×2 の計算を行うものである．

プログラムはまず，外側ループのカウンタとして用いる`$t0`と，最終結果を格納する`$t8`を 0 に初期化する．同時に，ループの終了条件を判定するための値「2」を，外側ループ用に`$t3`，内側ループ用に`$t4`へ設定する．外側のループ`loop_1`は，`beq`命令によってカウンタ`$t0`が 2 に達した時点で終了する．このループ構造で特に重要なのは，外側ループが一周するたびに，内側ループのカウンタである`$t1`を`add $t1, $zero, $zero`命令で 0 にリセットしている点である．これにより，内側のループ`loop_2`が常に初回から正しく実行されることが保証される．内側ループでは，カウンタ`$t1`が 2 に達するまで，中心的な処理である`addi $t8, $t8, 1`が繰り返される．

最終的に，外側のループが 2 回（`$t0`が 0 と 1 の時），そしてその各回で内側のループが 2 回（`$t1`が 0 と 1 の時）ずつ実行されるため，結果を保持する`$t8`は合計で 4 回インクリメントされる．したがって，プログラム終了時に`$t8`の値が「4」となるのは，この二重ループが設計意図通りに動作したことを示す，完全に妥当な結果である．

4.10　演習 11

```
1   .data
2
3       .globl v
4
5   v:  .word 3, 2, 1, 4, 5, 8, 17, 12, 10, 5
6
7
8   Size: .word 10
9
10  .text
11  .globl main
12      main:       la      $t0, v
13          addi    $t1, $zero, 0
14          lw      $t2, 0($t0)
15          add     $t1, $t1, $t2
16          lw      $t2, 4($t0)
17          add     $t1, $t1, $t2
18          lw      $t2, 8($t0)
19          add     $t1, $t1, $t2
20          lw      $t2, 12($t0)
21          add     $t1, $t1, $t2
22          lw      $t2, 16($t0)
23          add     $t1, $t1, $t2
24          lw      $t2, 20($t0)
25          add     $t1, $t1, $t2
26          lw      $t2, 24($t0)
27          add     $t1, $t1, $t2
28          lw      $t2, 28($t0)
29          add     $t1, $t1, $t2
30          lw      $t2, 32($t0)
31          add     $t1, $t1, $t2
32          lw      $t2, 36($t0)
33          add     $t1, $t1, $t2
34          move    $t9, $t1
```

```
35
36              jr          $ra
```

図 11　演習１１のコードリスト

表 11　命令の実行と関係するレジスタの値

PC	命令	$t0	$t1	$t2	$t9
4194340	la $t0, v	268500992	0	0	0
4194344	addi $t1, $zero, 0	268500992	0	0	0
4194348	lw $t2, 0($t0)	268500992	0	3	0
4194352	add $t1, $t1, $t2	268500992	3	3	0
4194356	lw $t2, 4($t0)	268500992	3	2	0
中略	中略				
4194420	lw $t2, 36($t0)	268500992	62	5	0
4194424	add $t1, $t1, $t2	268500992	67	5	0

このプログラムの主な目的は，メモリ上に定義された 10 個の整数からなる配列`v`の，全要素の合計値を計算することである．

高水準言語で用いられる「配列」は，アセンブリ言語レベルでは，`.data`セクションに`.word`ディレクティブなどを用いて確保された連続したメモリ領域として実現される．各要素にアクセスするには，まず配列の先頭アドレス（ベースアドレス）を特定し，そこからの相対的な位置（オフセット）を指定する必要がある．このプログラムでは，`la $t0, v`命令によって配列`v`の先頭アドレスをレジスタ`$t0`に一度ロードしている．これがアドレスをレジスタに退避させることに相当する．

アドレスをレジスタに退避させる理由は，CPU の高速なアドレッシングモードを活用し，メモリアクセスを効率化するためである．一度レジスタにベースアドレスを格納すれば，`lw $t2, 4($t0)`のように「ベースアドレス（`$t0`）＋ オフセット（`4`）」という形式で，各要素へ高速にアクセスできる．もし毎回メモリ上のラベル`v`からアドレスを計算していては，処理が著しく非効率になる．

プログラムの実行結果の妥当性については，そのロジックは明確であり，正しいと言える．まず合計値を累積するレジスタ`$t1`を 0 に初期化し，その後`lw`（ロード）と`add`（加算）の命令ペアを 10 回繰り返している．各`lw`命令は，ベースアドレス`$t0`に対して正しいオフセット（0, 4, 8, ..., 36）を用いて配列の全要素を順に読み出し，`add`命令がそれらを`$t1`へ正確に累積していく．最終的に，全要素の合計値である`67`が`$t1`に計算され，その値が`$t9`へコピーされてプログラムは終了する．ループを用いず命令を直接

記述する「ループ展開」という手法が取られているが，配列の合計を計算するという目的は正しく達成されており，その結果は完全に妥当である．

4.11　演習 12

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

```
32              lw      $t2, 32($t0)
33              add     $t1, $t1, $t2
34              lw      $t2, 36($t0)
35          add         $t1, $t1, $t2
36              move    $t9, $t1
37
38  .globl main
39
40  main:   la $t0, v
41              move $t3, $ra
42              jal sum
43              move $ra, $t3
44              jr $ra
```

図 12　演習１２のソースリスト

表 12　命令の実行と関係するレジスタの値

PC	命令	$t0	$t1	$t2	$t3
4194432	la $t0, v	268500992	0	0	0
4194436	move $t3, $ra	268500992	0	0	4194328
4194440	jal sum	268500992	0	0	4194328
省略					
4194436	move $ra, $t3	268500992	67	5	4194328
4194444	jr $ra	268500992	67	5	4194328

このプログラムは，単一の処理フローではなく，サブルーチン（関数）の呼び出しという，より構造化されたプログラミングの基本概念を扱うものである．その主な目的は，主処理を行う main 関数から，配列の合計値を計算する sum サブルーチンを呼び出し，その結果を受け取るという一連のプロセス，特にサブルーチン呼び出しに伴うリターンアドレスの管理方法を具体的に示すことにある．MIPS アーキテクチャにおいて，jal（Jump And Link）命令はサブルーチンへジャンプすると同時に，戻ってくるべきアドレスを自動的に $ra レジスタへ格納する．しかし，main 関数自体も OS などから呼び出されたサブルーチンであり，その呼び出し元へ戻るための重要なアドレスが$ra に保持されている．もし main 関数がそのまま jal sum を実行すると，この$ra の値は sum からの戻り先アドレスで上書きされ，main は元の場所へ戻れなくなってしまう．このプログラムは，その問題を解決するため，sum を呼び出す前に main の戻り先である$ra の値を別のレジスタ$t3 に一時

的に退避（保存）し，sum から戻った後に$t3 から復元するという，サブルーチン呼び出しにおける基本的な作法をデモンストレーションしている．
提供された実行結果の表は，この一連のプロセスを正確に反映しており，その内容は完全に妥当である．move $t3, $ra 命令によって$ra の値が$t3 に保存され，jal sum が実行される．sum 関数の処理は省略されているが，関数から戻ってきた時点で$t1 の値が配列の合計値である 67 になっていることから，sum が意図通りに動作したことがわかる．そして，move $ra, $t3 命令によって，退避させていた main の元のリターンアドレスが$ra に正しく復元されている．
以上の分析から，このプログラムはサブルーチン呼び出しにおけるリターンアドレス管理という目的を正しく実装しており，実行結果の表もその動作を忠実にトレースした，完全に妥当なものであると言える．（注：提示された sum 関数のソースコードには jr $ra がありませんが，main 関数に処理が戻っている実行結果から，実際には存在すると解釈するのが妥当である．）

4.12　演習 13

```
1   .data
2           .globl    v
3   v:      .word 3, 2, 1, 4, 5, 8, 17, 12, 10, 5
4
5           .text
6
7           .globl    main
8   main:   la $a0, v
9           li $a1, 10
10          jal sum
11
12          move $t9, $v0
13
14          li $v0, 10
15          syscall
16
17  sum:    li $v0, 0
18          li $t0, 0
19
20  sumloop:
```

```
21              bge $t0, $a1, sumend
22
23              sll $t1, $t0, 2
24              add $t2, $a0, $t1
25              lw   $t3, 0($t2)
26
27              add $v0, $v0, $t3
28
29              addi $t0, $t0, 1
30              j sumloop
31
32      sumend:
33              jr $ra
```

図 13　演習１３のソースリスト

表 13　命令の実行と関係するレジスタの値

PC	命令	$v0	$a0	$a1	$t0	$t1	$t2	$t3	$t9
main									0
4194340	la $a0, v	0	268500992	0	0	0	0	0	0
4194344	li $a1, 10	0	268500992	10	0	0	0	0	0
4194358	jal sum	0	268500992	10	0	0	0	0	0
中略	中略								
Sumloop									
4194380	sll $t1, $t0, 2	62	268500992	10	9	36	268501024	10	0
4194384	add $t2, $a0, $t1	62	268500992	10	9	36	268501028	10	0
4194388	lw $t3, 0($t2)	62	268500992	10	9	36	268501028	5	0
4194392	add $v0, $v0, $t3	67	268500992	10	9	36	268501028	5	0
4194396	addi $t0, $t0, 1	67	268500992	10	10	36	268501028	5	0
4194400	j sumloop	67	268500992	10	10	36	268501028	5	0
4194372	bge $t0, $a1, sumend	67	268500992	10	10	36	268501028	5	0
Sumend									
4194404	jr $ra		268500992	10	10	36	268501028	5	0
main									
4194352	move $t9, $v0	67	268500992	10	10	36	268501028	5	67
4194356	li $v0, 10	10	268500992	10	10	36	268501028	5	67

このプログラムは，より実践的かつ汎用的なサブルーチン（関数）設計を示すものである．その主な目的は，引数として与えられた「配列のメモリアドレス」と「要素数」に基づき，ループ構造を用いて配列の全要素の合計を計算し，その結果を戻り値として返す`sum`関数を実装することにある．そして`main`関数は，この`sum`関数を呼び出すための引数を設定し，実行後に戻り値を受け取るという，構造化プログラミングの基本的な役割分担をデモンストレーションしている．
アセンブリ言語における効率的な関数設計では，大量のデータを直接コピーするのではなく，そのデータが格納されている先頭アドレスと，大きさなどの関連データをレジスタに退避させて（格納して）渡すのが基本である．このプログラムでは，`main`関数が配列`v`の先頭アドレスを引数レジスタ`$a0`に，要素数`10`を`$a1`に格納し，`jal`命令で`sum`関数を呼び出す．`sum`関数は，渡されたこれらの情報を頼りに，ループを用いて配列の要素に順次アクセスし，合計を計算する．このアドレス渡しは，メモリ使用量と処理時間を削減するための極めて重要な手法である．
実行結果の表は，注釈の通り，ループ処理の最終盤（10 回目のループ）と関数から戻った後の様子を抜粋して示しているが，その内容はソースコードのロジックと完全に一致しており，完全に妥当である．表では，ループカウンタ`$t0`が`9`の時に最後の要素`5`が読み出され，それまでの合計`62`に加算されて最終的な合計値`67`が`$v0`に得られる過程が正確に示されている．その後，`bge`命令でループを正しく脱出し，`jr $ra`によって`main`関数へ戻る．`main`関数では，戻り値が格納された`$v0`の値（67）を`$t9`にコピーしており，これも表の通りである．

以上の分析から，このプログラムはループを用いた汎用的なサブルーチンの実装と，関数間の引数・戻り値の受け渡しという目的を正しく達成している．そして実行結果の表も，その重要な部分の動作を正確に反映しており，全体として目的，実装，結果のすべてが妥当であると言える．

4.13　演習 14

```
1                .data
2                .text
3
4                .globl    fib
5        fib:                #フィボナッチ関数
6                addi      $sp, $sp, -12
7                sw        $ra, 0($sp)
8                sw                $a0, 4($sp) # 引数
9
10               slti      $t0, $a0, 2
11               beq               $t0, $zero, L3
12               beq               $a0, $zero, L1
13               j                 L2
14
15
16               .globl L1
17       L1:
18               addi      $v0, $zero, 0
19               addi      $sp, $sp, 12
20               jr                $ra
21
22               .globl L2
23       L2:
24               addi      $v0, $zero, 1
25               addi      $sp, $sp, 12
26               jr                $ra
27
28               .globl    L3
29       L3:
30               addi      $a0, $a0, -1
31               jal               fib
32               sw                $v0, 8($sp)
33
34               lw                $a0, 4($sp)
35               addi      $a0, $a0, -2
36               jal       fib
```

```
37
38              lw              $t1, 8($sp)
39              add     $v0, $v0, $t1
40
41              lw              $ra, 0($sp)
42              lw              $a0, 4($sp)
43              addi    $sp, $sp, 12
44
45              jr              $ra
46
47              .globl main
48      main:
49              li              $t0, 10
50              addi    $sp, $sp, -8
51              sw              $t0   0($sp)
52              sw              $ra, 4($sp)
53
54              move    $a0, $t0
55              jal     fib
56              move    $t1, $v0
57
58              lw              $t0, 0($sp)
59              lw              $ra, 4($sp)
60              addi    $sp, $sp, 8
61
62              jr              $ra
```

図 14　演習１４のソースリスト

表 14　演習 14 の命令の実行と関係するレジスタ

PC	命令	$v0	$a0	$t0	$t1	$sp	$ra
省略							
4194460	jal fib	4	10	10	0	2147483108	4194328
4194340	addi $sp, $sp, -12	4	10	10	10	2147483108	4194464
4194344	sw $ra, 0($sp)	4	10	10	0	2147483096	4194464
省略							

4194336	addi $sp, $sp, 8	10	10	10	55	2147483116	4194328
4194480	jr $ra	55	10	10	55	2147483116	4194328

以下に，関数 fib(10)の呼び出し直後と終了後のスタック領域を表 15 に示す．呼び出し直後を sw　$a0, 4($sp）を実行した直後とし，終了後を lw $ra,0($sp）を終え，これから addi $sp,$sp,12 → jr $ra へ進む直前とする．

表 15　fib(10)の呼び出し直後・終了後のスタック領域

[7ffffdd8]	0004194464　0000000010
[7ffffd78]	0004194400　0000000002

以下に，関数関数 fib(10)の呼び出し直後と終了後のスタック領域を表 16 にしめす．呼び出し直後を sw $a0,4($sp)とし，呼び出し終了直前を lw $ra,0($sp）実行直後とする．

表 16　fib(0)の呼び出し直後・終了後のスタック領域

[7ffffdd8]	
[7ffffdcc]	0004194400
[7ffffdd8]	0004194464　0000000010
[7ffffdcc]	0004194400
[7ffff3f0]	0004194400　0000000006　0000000005　0004194400
[7ffff400]	0000000007 0000000002 0004194400 0000000008
[7ffff410]	0000000005 0004194400 0000000009 0000000013
[7ffff420]	0000000005 0004194400 0000000009 0000000013
[7ffff430]	0004194328 0000000001 2147480838 0000000000

Fib(5)について考える．main から `jal fib` が実行されるとプロローグで `$sp` を 12 バイト減らして新しいフレームを確保し，0(¥$sp）に `$ra`，4(¥$sp）に n＝5 を保存した後に n－1＝4 を `$a0` に設定して `fib(4)` を再帰呼び出しし，戻り値を 8(¥$sp）に退避してから n－2＝3 を `$a0` にして `fib(3)` を呼び戻し，得られた値と 8(¥$sp）から取り出した `fib(4)` の値を `add` 命令で加算して `$v0` に格納し，エピローグで `lw $ra,0($sp)` で戻りアドレスを復元しつつ `addi $sp,$sp,12` でフレームを解放してスタックを 12 バイト戻し，最後に `jr $ra` を実行して呼び出し元の main に `fib(5)=fib(4)+fib(3)` の結果を返す．

このプログラムは main が n＝10 をレジスタ ¥$t0 にロードして ¥$ra とともにスタックへ退避したうえで `jal fib` を実行して再帰計算を開始し，fib プロローグでは毎回 12 バイトだけ ¥$sp を下げて新しいフレームを確保し 0(¥$sp) に戻りアドレス ¥$ra，4(¥$sp) に引数 n（¥$a0），8(¥$sp) に `fib(n-1)` の戻り値を一時保存する領域を確保しておくことで jal による ¥$ra 上書きや ¥$a0 書き換えから元の値を保護しつつ，再帰末端の fib(0)=0 と fib(1)=1 から戻るたびに 8(¥$sp) から `fib(n-1)` を取り出して直前に得た `fib(n-2)` と加算し `$v0←fib(n-1)+fib(n-2)` として戻り値を更新し，エピローグで ¥$ra をロードして `$sp+=12` でフレームをポップしながら LIFO でスタックを完全に元の高さへ戻していき，最終的に fib(10)=55 が `$v0` で main に返って ¥$t1 に移され，スタックに残留データを残さず計算結果が正しく 55 となることでプログラムの目的と実行結果の妥当性が確認できる．Fib(5)の再帰呼び出し構造を模式化したツリーを図 15 に示す．

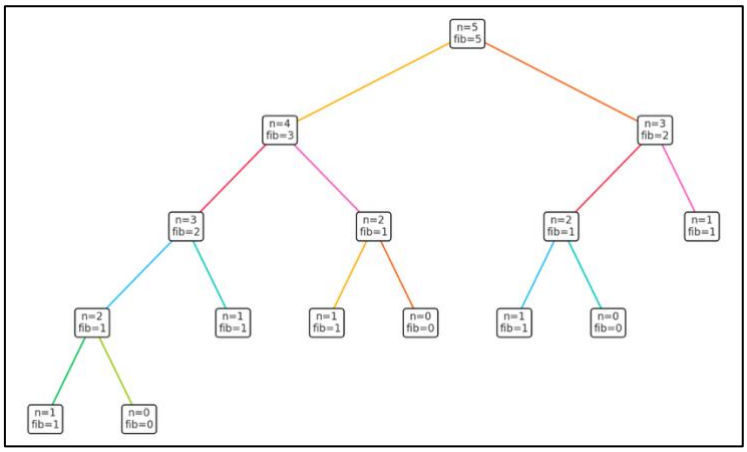

図 15　fib(5)の再帰呼び出し構造

5. 検討課題

課題 1

MIPS アーキテクチャは，単純化された命令セットで高速な処理を目指す RISC（Reduced Instruction Set Computer）思想を体現した代表的なプロセッサである．その特徴の一つに，全ての命令を 32 ビットの固定長とし，命令の形式を R，I，J の数種類に限定することで，CPU のデコード処理を簡素化・高速化している点が挙げられる．本稿では，これらの中から即値（Immediate）を伴う演算やメモリアクセスで広く用いられる I 形式命令を取り上げ，その具体的な構成と設計の妥当性を論じる．

例として，I 形式の算術演算命令である addi（Add Immediate）を用いて解説を進める．I 形式の命令フォーマットは 32 ビットであり，以下の 4 つのフィールドで構成される．まず，命令の種類を識別するための 6 ビットの opcode，次にソースレジスタを指定する 5 ビットの rs，そして主に結果の格納先となるターゲットレジスタを指定する 5 ビットの rt，最後

に演算で直接使用される 16 ビットの immediate（即値）である．I 形式のフィールド分割を図 16 に示す．

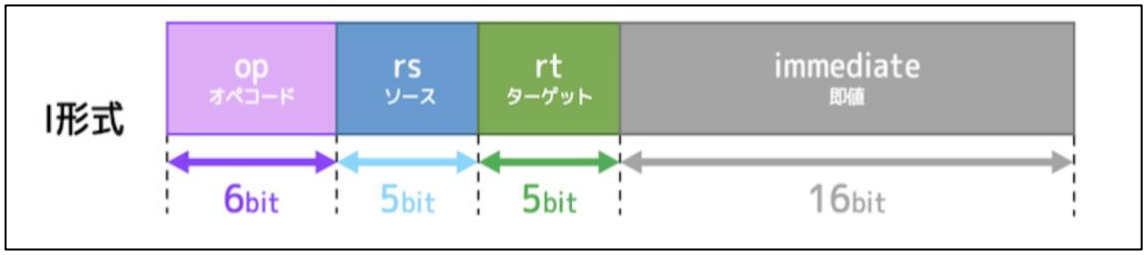

図 16　I 形式のフィールド分割[1]

具体的なアセンブリ命令 addi $t1, $t0, 100 を機械語へ変換するプロセスを考える．この命令は，「レジスタ$t0 の値と即値 100 を加算し，その結果をレジスタ$t1 に格納せよ」という指示である．各フィールドへの割り当ては次のようになる．addi 命令の opcode は 001000 と定義されている．ソースレジスタ$t0 は 32 本ある汎用レジスタの 8 番目に相当するため，rs フィールドは 01000 となる．同様に，ターゲットレジスタ$t1 は 9 番目のため，rt フィールドは 01001 である．最後の immediate フィールドには，定数 100 が 16 ビットの符号付き 2 進数 0000000001100100 として格納される．これらを連結した 00100001000010010000000001100100 が，この addi 命令に対応する 32 ビットの機械語表現（16 進数では 0x21090064）となる．

この I 形式の構成は，CPU のパイプライン処理において極めて合理的である．CPU は命令をフェッチすると，まず先頭 6 ビットの opcode を読み，命令の種類を特定する．フィールドの位置が固定されているため，デコーダは迅速に rs と rt が指すレジスタ番号や immediate の値を特定できる．そして，rs フィールドが示すレジスタから値を読み出し，immediate フィールドの 16 ビット値は 32 ビットに符号拡張された上で ALU（算術論理演算装置）へ送られる．ALU が加算を実行した後，その結果は rt フィールドが示すレジスタに書き戻される．このように，命令の各フィールドが CPU ハードウェアの各機能ブロックの動作と整然と対応しており，一連の処理が滞りなく流れるように設計されている．この単純明快な構造こそ，RISC の設計思想に基づいた，効率性と高速性を両立させるための優れた工夫なのである[2].

5.1 課題 2

高級言語（C，C++，Java，Python など）は移植性が非常に高く，一度書いたソースコードをコンパイルまたはインタプリタ実行するだけで異なる CPU や OS 環境へ簡単に移行できるという利点がある．その一方で，アセンブリ言語は特定の CPU の命令セットに依存するため，環境が変わるとソースコード全体を大幅に書き換える必要があり，移植性は極めて低い．ここで，プログラミング言語の種類を Error! Reference source not

found.に示す.

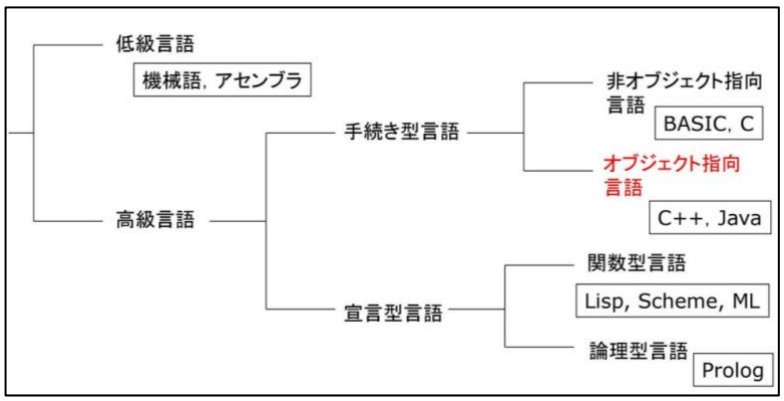

図 17　プログラミング言語の種類[3]

実行速度の観点では，高級言語の場合，コンパイラやインタプリタが汎用的な最適化を施すが，メモリアクセスやキャッシュ利用，分岐予測などの CPU 動作を細かく制御できないため，最適化に限界があり，最高速度の達成は難しい．対してアセンブリ言語では，レジスタ管理，パイプライン遅延や SIMD 命令の使用などを命令単位で精密に制御できるため，極限まで実行速度を追求することが可能である．

実装のしやすさに関しては，高級言語は豊富な抽象化機能（ポインタ，構造体，クラスなど）と強力なデバッグ環境，統合開発環境（IDE）を利用できるため，コードが書きやすく，大規模プロジェクトでも高い生産性を実現できる．一方，アセンブリ言語ではすべてを手作業で行うためコード量が多くなり，スタック管理やレジスタ割付けも手動で制御する必要があるため，開発効率は低く，コードの保守性や可読性も非常に低くなる[4].

このため現代のソフトウェア開発では，全体のコードを移植性と生産性に優れた高級言語で記述し，性能上のボトルネックとなるクリティカルな部分だけをアセンブリ言語で手作業最適化するというハイブリッドな手法が，移植性，実行速度，実装のしやすさを総合的に考えた現実的な解となっている．

6. まとめ

3 週間にわたるアセンブリ言語プログラミング実習では，MIPS アーキテクチャの基礎から応用的なプログラミング技法までを段階的に学んだ．

第 1 週では，MIPS の命令セットやレジスタの役割について学び，QtSPIM シミュレータを用いて基本的な算術演算，ロード・ストア，条件分岐などの命令を扱う演習に取り組んだ ．第 2 週は，メモリ上の配列データへのアクセス方法や，`jal`命令を用いたサブルーチンの実装，そしてスタックを利用した引数や戻り値を持つ関数の作成方法を学んだ ．最終の第 3 週では，スタック操作をさらに応用し，階乗計算やフィボナッチ数列を題材とした再帰関

数の実装に挑戦した．
この一連の実習を通じ，コンピュータが命令をどう実行するかという基本原理から，高級言語の関数や再帰処理がアセンブリ言語レベルでどのように実現されるかまで，理解を深めることができた．

7. 感想

この 3 週間のアセンブリ言語プログラミング実習を通して，コンピュータのハードウェアとソフトウェアがどのように協調して動作するのかを深く学んだ．アセンブリ言語は CPU の機械語と密接に関連しており，普段は意識しないレジスタやメモリを直接操作することで，コンピュータの動作原理を肌で感じることができた．特に，C 言語などで当たり前に使っている配列や関数の実装方法が印象的であった．配列はメモリ上のアドレス計算によって，関数の呼び出しや再帰処理は，限られたレジスタの値をスタックに退避・復元させることで実現されていることを演習を通じて学んだ．この経験から，コンパイラが高級言語をいかにしてアセンブリ言語に翻訳しているかという役割の重要性を実感し，プログラムが動く仕組みの解像度が大きく向上した．

参考文献

[1] 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」, https://www.momoyama-usagi.com/entry/info-calc-sys21, 2023 年 7 月 17 日更新.
[2] 「MIPS のまとめ」,
chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/http://ocw.kyushu-u.ac.jp/menu/faculty/09/4/10.pdf, 九州大学工学部電気情報工学科, 2025 年 7 月 4 日参照.
[3] 新田直也，「プログラミング言語論」, https://slidesplayer.net/slide/11517979/, 2025 年 7 月 4 日参照.
[4] 「マイクロプロセッサ演習」,
chromeextension://efaidnbmnnnibpcajpcglclefindmkaj/https://brain.cc.kogakuin.ac.jp/~kanamaru/lecture/MP/final/part07/part07.pdf, 2025 年 7 月 4 日参照.