

言い訳

このスライドはAIが作成したなんかないいろいろ適当です。 [github](#) にいろいろ置いといたからそっち読んで!

コンピュータの構成と設計

第2章 演習問題解説

パタヘネ2章の演習問題1-22を通して学ぶ

MIPS アセンブリ言語とコンピュータアーキテクチャ

MIPSとは？

MIPS (Microprocessor without Interlocked Pipeline Stages)

- 1980年代にスタンフォード大学で開発されたRISCアーキテクチャ
- **RISC**: Reduced Instruction Set Computer
- シンプルで規則的な命令セット
- 教育用プロセッサとして広く使用

特徴:

- 32ビットアーキテクチャ
- 32個の汎用レジスタ
- 固定長命令（32ビット）
- ロード・ストア型アーキテクチャ

MIPS vs x86の違い

項目	MIPS	x86
設計思想	RISC	CISC
命令長	固定（32ビット）	可変（1-15バイト）
レジスタ数	32個	少数（歴史的制約）
メモリアクセス	ロード・ストア	レジスタ・メモリ
学習コスト	低い	高い

なぜMIPSを学ぶ？

- アーキテクチャの基本概念を理解しやすい
- 命令セットがシンプルで規則的
- 他のアーキテクチャへの応用が効く

MIPSレジスタ構成

32個の汎用レジスタ (32ビット)

レジスタ	名前	用途
\$0	\$zero	常に0
\$1	\$at	アセンブラー時レジスタ
\$2-\$3	\$v0-\$v1	戻り値
\$4-\$7	\$a0-\$a3	引数
\$8-\$15	\$t0-\$t7	一時レジスタ
\$16-\$23	\$s0-\$s7	保存レジスタ
\$24-\$25	\$t8-\$t9	一時レジスタ

MIPSレジスタ構成(続き)

レジスタ	名前	用途
\$26-\$27	\$k0-\$k1	カーネルレジスタ
\$28	\$gp	グローバルポインタ
\$29	\$sp	スタックポインタ
\$31	\$ra	リターンアドレス

MIPS命令形式

3つの命令形式

R形式 (Register)

[opcode:6][rs:5][rt:5][rd:5][shamt:5][funct:6]

例: add \$t0, \$s1, \$s2

I形式 (Immediate)

[opcode:6][rs:5][rt:5][immediate:16]

例: addi \$t0, \$s1, 100

J形式 (Jump)

[opcode:6][target:26]

例: j 0x400000

MIPS基本命令

算術演算

```
add  $t0, $s1, $s2    # $t0 = $s1 + $s2
addi $t0, $s1, 100     # $t0 = $s1 + 100
sub   $t0, $s1, $s2    # $t0 = $s1 - $s2
```

メモリアクセス

```
lw  $t0, 0($s1)        # $t0 = Memory[$s1]
sw  $t0, 4($s1)        # Memory[$s1+4] = $t0
la  $t0, array         # $t0 = &array
```

制御フロー

```
beq $t0, $t1, label    # if ($t0 == $t1) goto label
j   label              # goto label
jal function            # call function
```


SPIMシミュレータ

SPIM (MIPS Simulator)

- MIPSアセンブリプログラムを実行できるシミュレータ
- 実際のMIPSプロセッサの動作を模擬
- デバッグ機能付き

使用方法:

```
# ファイルを実行
spim -file program.s

# 対話モード
spim
(spim) load "program.s"
(spim) run
```

演習問題の全体像

範囲	内容	ファイル例
1-4	基本演算・配列操作	enshuu1.s , enshuu3.s
5-6	数値表現・エンディアン	enshuu5.md , enshuu6.md
7-8	64ビット演算・アドレス	enshuu7.s , enshuu8.c
9-15	機械語・命令形式	enshuu9.s , enshuu12.s
16-22	アーキテクチャ拡張	enshuu17_1.s , enshuu22.md

演習1: MIPS基本演算

問題: $f = g + (h - 5)$

```
li    $s1, 100      # g = 100
li    $s2, 5         # h = 5
addi  $s2, $s2, -5   # (h - 5)
add   $s0, $s1, $s2  # f = g + (h - 5)
```

ポイント:

- `addi` で即値減算
- レジスタ割り当て: `$s0=f` , `$s1=g` , `$s2=h`

演習2: C言語との比較

```
// C言語版
int hoge(int f, int g, int h, int i) {
    f = g + h + i;
}
```

学習効果:

- 高級言語と低級言語の対応関係
- コンパイラの最適化理解

演習3: 配列操作と境界チェック

問題: `arrayB[i] = arrayA[i-j]` (境界条件付き)

```
sub    $s3, $s3, $s4      # i - j
sltu   $t1, $s3, $t0      # 境界チェック (0 < i-j < 配列長)
beq     $t1, $zero, error  # 条件違反時はエラー
sll     $s3, $s3, 2        # インデックス×4 (バイト単位)
add     $s3, $s3, $s0      # 配列ベースアドレス + オフセット
lw      $t2, 0($s3)        # arrayA[i-j]を読み込み
sw      $t2, 32($s1)       # arrayB[i]に格納
```

重要概念:

- 配列インデックス計算 (`sll $s3, $s3, 2`)
- 境界チェック (`sltu`)
- エラーハンドリング

演習4: ポインタ操作

```
int hoge2(int A[], int B[], int f, int g) {  
    B[g] = A[f] + A[f + 1];  
}
```

アセンブリだとやばそうなことでもできるね。

演習5: エンディアン形式

0xABCDEF12の格納方法

	Little Endian	Big Endian
アドレス	値	値
0x00	0x12	0xAB
0x01	0xEF	0xCD
0x02	0xCD	0xEF
0x03	0xAB	0x12

余談

- 元ネタはガリバー旅行記

演習6: 16進数→10進数変換

0xABCDEF12を10進数に

$$10 \times 16^7 + 11 \times 16^6 + 12 \times 16^5 + 13 \times 16^4 + 14 \times 16^3 + 15 \times 16^2 + 1 \times 16^1 + 2 \times 16^0 \\ = 2,882,400,018_{10}$$

計算過程:

- 各桁の16進数字を10進数に変換
- 位の重みを掛けて合計

演習7: 64ビット加算

arrayA[i] + arrayB[j] (64ビット)

```
lw $t0, 0($s6)      # arrayA下位32ビット
lw $t1, 4($s6)      # arrayA上位32ビット
lw $t2, 0($s8)      # arrayB下位32ビット
lw $t3, 4($s8)      # arrayB上位32ビット
addu $t2, $t0, $t2   # 下位32ビット加算
sltu $t0, $t0, $t2   # キャリー検出
addu $t3, $t1, $t3   # 上位32ビット加算
addu $t3, $t3, $t0   # キャリーを加算
```

重要概念:

- 64ビット値の分割格納
- キャリー処理
- `sltu` でのオーバーフロー検出

演習8: C言語でのアドレス操作

```
int hoge(int A[], int f) {  
    A[1] = (int)&A[0];    // アドレスを整数として格納  
    f = &A[0] + A[1];    // アドレス演算  
    return f;  
}
```

学習効果:

- C言語でのアドレス操作
- ポインタと配列の関係

演習9: 機械語形式

MIPSアセンブリ→機械語変換

```
addi $t0, $s6, 4    # 001000 10110 01000 00000000000000100
add  $t1, $s6, $0    # 000000 10110 00000 01001 00000 100000
sw   $t1, 0($t0)     # 101011 01000 01001 00000000000000000
```

命令形式:

- I形式: opcode(6) + rs(5) + rt(5) + immediate(16)
- R形式: opcode(6) + rs(5) + rt(5) + rd(5) + shamt(5) + funct(6)

演習10: オーバーフロー処理

```
# 10.1: 加算
addu $t0, $s0, $s1  # オーバーフロー無視
# 結果: 0x50000000
# 10.2: add命令だとArithmetic Overflow例外

# 10.3: 減算
sub $t0, $s0, $s1    # 0xB0000000
# 結果: 0xB0000000
# 10.4: これは期待通りの結果

# 10.5: さらに加算
addu $t0, $s0, $s1    # Add A and B
addu $t0, $t0, $s0    # Add A again to the result
# 結果: 0xC0000000

# 10.6: 10.2時点でオーバーフローしてるんだからここもオーバーフローする
```

演習11: オーバーフロー範囲計算

\$s0=128の場合

add命令でオーバーフロー:

```
2,147,483,519 < $s1 < 2,147,483,648
```

sub命令でオーバーフロー:

```
-2,147,483,649 < $s1 < -2,147,483,519 #引き算の順序に依存せず
```

理由: 32ビット符号付き整数の範囲制限

演習12-13: 機械語解読

演習12: 00000010000100001000000000100000

```
opcode=000000 (R形式)
funct=100000 (add命令)
rs, rt, rd = 10000, 10000, 10000 ($s0)
→ add $s0, $s0, $s0
```

演習13: Store Word命令

```
sw $t1, 32($t2) # 0x20 0x00 0x49 0xad (Little Endian)
```

演習14-15: 命令デコード

演習14: opcode 0 rs 3 rt 2 rd 3 shamt 0 funct 34

opcode=0はR形式命令, funct 34で sub命令

```
sub $v1, $v1, $v0 # funct=34はsub命令
```

演習15: opcode 0x23 rs 1 rt 2 const 0x4

opcode=0x23はlw命令, rs=1は\$atレジスタ, rt=2は\$v0レジスタ

```
lw $v0, 4($at) # opcode=0x23はlw命令
```



演習16: アーキテクチャ拡張

MIPSレジスタファイルを128個に拡張

影響:

- R命令: レジスタ指定に7ビット必要 → 追加6ビット
- I命令: レジスタ指定2つ → 追加4ビット
- 命令長: 32ビット → 64ビットに拡張

トレードオフ:

-  レジスタスピル減少
-  命令長増加、メモリ使用量増

演習17: ビット操作

シフトとマスク操作

17.1: 44ビット左シフト (そんなものはできない)

```
sll $t2, $t0, 31    # 31ビットシフト  
sll $t2, $t2, 13    # 追加13ビット (計44ビット)
```

17.2: 4ビット左シフト+マスク

```
sll $t2, $t0, 4      # 4ビット左シフト  
andi $t2, $t2, 0xFFFF # 下位16ビットマスク (-1はandiにつかえません)
```

演習18: bit操作

演習18: 条件付きビット操作

```
sll $t2, $t0, 15 # 15ビット左シフト
srl $t2, $t2, 26 # 右のbitをすべて消す
sll $t2, $t2, 26 # 31-26にセット
sll $t3, $t1, 6  # 左側bitを6ビット左シフト
srl $t3, $t3, 6  # 右側bitを6ビット右シフト
or  $t2, $t2, $t3 # $t2 = A | B
```

演習19: bitflip

演習19: bitflip

2の補数表現において、 $x + \text{bitflip}(x) + 1 = 0$ の関係があるのでそれを用いる。

```
sub $t2, $zero, $t2    # -A
addi $t1, $t2, -1      # -A - 1
```

演習20-21: 制御フロー

演習20: 配列要素のシフト演算

```
lw $t0, 0($s0)    # C[0]読み込み  
sll $t0, $t0, 4    # 16倍 (左4ビットシフト)
```

演習21: 条件分岐

```
lui $t0, 0x0101    # 上位16ビット設定  
slt $t2, $zero, $t0 # 0 < $t0 ?  
bne $t2, $zero, ELSE # 条件分岐
```

演習22: アドレッシングモード

PC = 0x2000 0000からのジャンプ範囲

JAL命令 (疑似直接アドレッシング):

範囲: 0x2000 0000 ~ 0x2FFF FFFC

- PC上位4ビット + 26ビット即値 を 2ビット左シフト

BEQ命令 (PC相対アドレッシング):

範囲: 0x1FFF 8000 ~ 0x2000 7FFF

- $PC \pm 16\text{ビット符号付きオフセット} \times 4$ (この4は語長)

まとめ

学習した主要概念

1. **基本演算**: レジスタ操作、即値演算
2. **メモリ操作**: 配列、ポインタ、エンディアン
3. **64ビット演算**: 分割処理、キャリー検出
4. **機械語**: 命令形式、エンコーディング
5. **制御フロー**: 分岐、ジャンプ、条件処理
6. **ビット操作**: シフト、マスク、論理演算
7. **アーキテクチャ**: 命令セット拡張の影響

次のステップ

- SPIM シミュレータでの実行
- パフォーマンス解析

参考資料とツール

実行環境:

```
# SPIM シミュレータ  
spim -file enshuu1.s
```

重要なMIPSレジスタ:

- `$s0-$s7` : 保存レジスタ
- `$t0-$t9` : 一時レジスタ
- `$v0-$v1` : 戻り値、システムコール
- `$a0-$a3` : 引数レジスタ

オンラインツール

Compiler Explorer (godbolt.org)

- 様々な言語・アーキテクチャのコンパイル結果を確認
- C/C++からMIPSアセンブリへの変換を試せる
- リアルタイムでコンパイル結果が見える
- URL: <https://godbolt.org/>

Online Assembler and Disassembler

- アセンブリ ↔ 機械語の相互変換
- MIPSを含む多数のアーキテクチャに対応
- 16進数ダンプの解析に便利
- URL: <https://shell-storm.org/online/Online-Assembler-and-Disassembler>