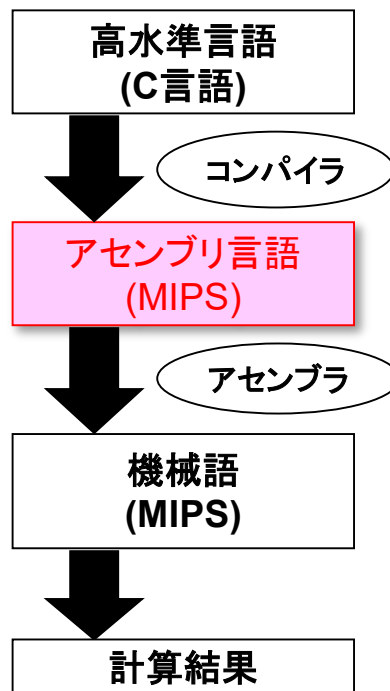


2019年度 計算機システム(演習)  
第3回  
2019.12.13

遠藤 敏夫(学術国際情報センター/数理・計算科学系 教授)  
野村 哲弘(学術国際情報センター/数理・計算科学系 助教)



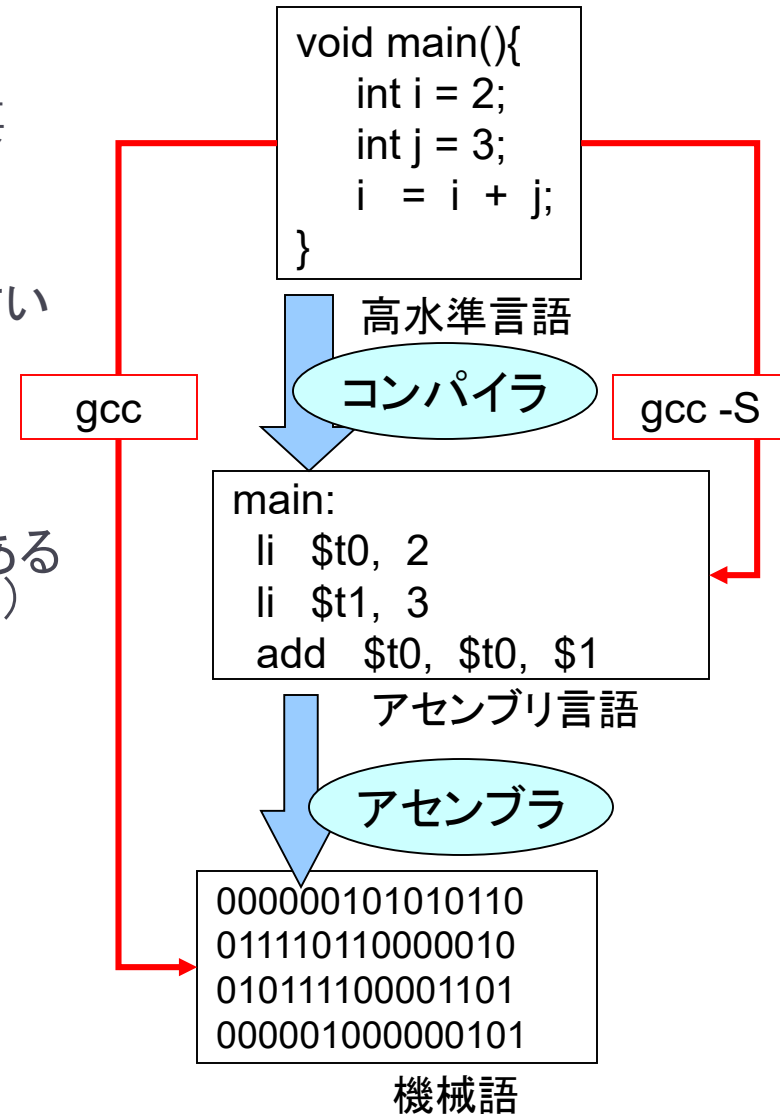
# 今日の内容

アセンブラ命令、配列の操作方法

# アセンブリ言語 (MIPS)

# プログラム実行までの流れ

- ▶ プログラムが実行されるまで
  - ▶ コンパイラ、アセンブラ、実行ファイル
  - ▶ プロセッサが処理可能な形式まで変換する必要
- ▶ 高水準(高級)言語 ←前回までの内容
  - ▶ 自然言語に近い構文であり、人間が記述しやすい
  - ▶ Java, Cなど
- ▶ アセンブリ言語(低級言語) ←次の内容
  - ▶ コンピュータ用に2進数で符号化した命令である機械語(machine language)を, 記号(シンボル)表記したものである.
  - ▶ 機械語を人間が理解できるように記述
- ▶ 機械語
  - ▶ CPUが直接理解できる言語
  - ▶ 0,1であらわされる命令の集まり
    - ▶ 命令セット



# MIPSアーキテクチャ

---

## ▶ Microprocessor without Interlocked Pipeline Stages

m1.s

```
        .data
str:    .asciiz "HelloWorld\n"
        .text
main:
        li $v0, 4
        la $a0, str
        syscall
        jr $ra
```

- Hello World プログラム
  - “HelloWorld” という文字列を画面に表示

# Hello World プログラム

---

- MIPSは2つのセグメントから成る

```
        .data  
str:  
        .asciiz "HelloWorld¥n"
```

## ▶ データセグメント

- ▶ .data 以下
- ▶ データ部分

```
        .text  
main:  
        li $v0, 4  
        la $a0, str  
        syscall  
        jr $ra
```

## ▶ コードセグメント

- ▶ .text 以下
- ▶ 命令列

# データセグメント

データセグメントの開始

```
str:
.data
.asciiz "HelloWorld¥n"
```

ラベル

文字列

データが文字列であることを指定

## ▶ 定数の記述

- ▶ 実行中に変わらない値

## ▶ ラベル

- ▶ データのある場所(アドレス)に名前をつける

.asciizを使わないと...

```
.byte 72, 101, 108, 108 ...
```

# テキストセグメント

テキストセグメントの開始

## ▶ テキストセグメント

▶ プログラムの処理を記述

.text

main:

li **\$v0**, **4**

la \$a0, str

syscall

jr \$ra

個々を「オペランド」と呼ぶ

→ レジスタ \$v0 に 4 を代入 (\$v0 = 4)

→ レジスタ \$a0 に str を代入 (\$a0 = str)

→ システムコールを呼ぶ

→ メインルーチンの終了

メインルーチン  
を表すラベル



# ロード命令

---

```
.text  
main:  
    li $v0, 4  
    la $a0, str  
    syscall  
    jr $ra
```

- ▶ **li レジスタ, 数値**
  - ▶ load immediate
  - ▶ 数値をレジスタに代入
  - ▶ 例: li \$v0, 4
- ▶ **la レジスタ, ラベル**
  - ▶ load address
  - ▶ ラベルの指すアドレスをレジスタに代入
  - ▶ 例: la \$a0, str

# 使用できるレジスタ

---

- ▶ **レジスタ:** CPU内部に存在し値を保持する少量で高速な記憶素子
  - ▶ CPUはレジスタに対して計算を行う

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

# syscall 命令

---

```
.text
main:
    li $v0, 4
    la $a0, str
    syscall
    jr $ra
```

- ▶ システムコールを呼ぶ
  - ▶ OS が提供するサービス
    - ▶ 入出力など
  - ▶ 一種のサブルーチン
- ▶ 使い方
  - ▶ レジスタ \$v0 にサービス番号を設定
    - ▶ 例) \$v0=4: 文字列表示
  - ▶ レジスタ \$a0 等に引数を設定
  - ▶ syscall 命令を実行
  - ▶ (戻り値があれば)レジスタ \$v0 に入る

# syscall サービス

---

サービス	番号 (\$v0)	引数	返回值	意味
print_int	1	\$a0(整数)		整数値を表示
print_string	4	\$a0(文字列のアドレス)		文字列を表示
read_int	5		\$v0(整数)	整数値を読み込む
read_string	8	\$a0(バッファ) \$a1(長さ)		文字列を読み込む
sbrk	9	\$a0(メモリサイズ)	\$v0(アドレス)	メモリを割り当て
exit	10			プログラム終了

# syscall 使用例

---

- ▶ 整数値の出力

- ▶ 例: 128 を出力

```
li $v0, 1  
li $a0, 128  
syscall
```

- ▶ 整数値の入力

- ▶ \$v0 に入力値が入る

```
li $v0, 5  
syscall
```

- ▶ 文字列の出力

- ▶ \$a0に代入された文字列を表示

```
li $v0, 4  
li $a0, str  
syscall
```

# SPIM

---

## ▶ MIPSシミュレータ

- ▶ <http://spimsimulator.sourceforge.net/>

- ▶ Windows, Mac OS X, Linux 版

## ▶ インストール & 利用方法

- ▶ 選択肢 1: 西7の Mac

- ▶ App フォルダに QtSpim がインストールされている

- ▶ 選択肢 2: 自宅 PC

- ▶ <https://sourceforge.net/projects/spimsimulator/files/>

- ▶ QtSpim\_(version)\_(os).(ext) をダウンロード

- QtSpim\_9.1.20\_Windows.msi など

- ▶ 展開もしくはインストーラを実行

基本的に西7のMacを用いる。選択肢2は、家や自分のラップトップで課題をやりたい学生向け

## 制御ボタン



## エラー出力など

```
spim: (parser) Label is defined for the second time on line 8 of file /Users/shirahata/Documents/ææ¥/ǎ/é"ǎ°éç§ç®/è"ç®æ©ǎ-ǎ'ǎǎ /2012/MIPS/m2.s
```

```
main:
```

A

# Hello World (1/3)

---

- ▶ Hello World プログラムを作成
  - ▶ ファイル名: hello.s

```
                .data
str:            .ascii "HelloWorld\n"

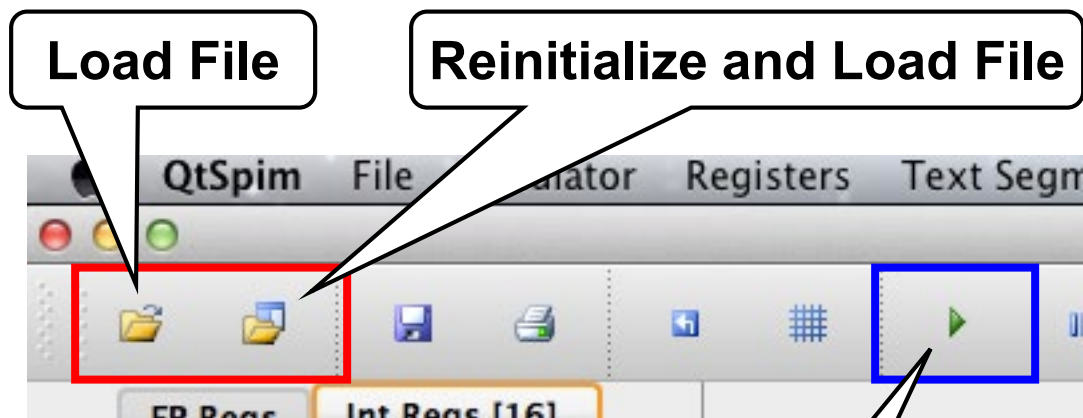
                .text
main:
    li $v0, 4
    la $a0, str
    syscall
    jr $ra
```



# Hello World (2/3)

## ▶ hello.s プログラムの読み込み

- ▶ 起動後、[Load File] または [Reinitialize and Load File]
  - ▶ プログラムを選択



## ▶ hello.s の実行

- ▶ プログラムを最後まで実行してみる
  - ▶ [Run] ボタン

# Hello World (3/3)

- ▶ プログラムを修正した場合
  - ▶ [Reinitialize and Load File] → 初期化してファイルを読み込み

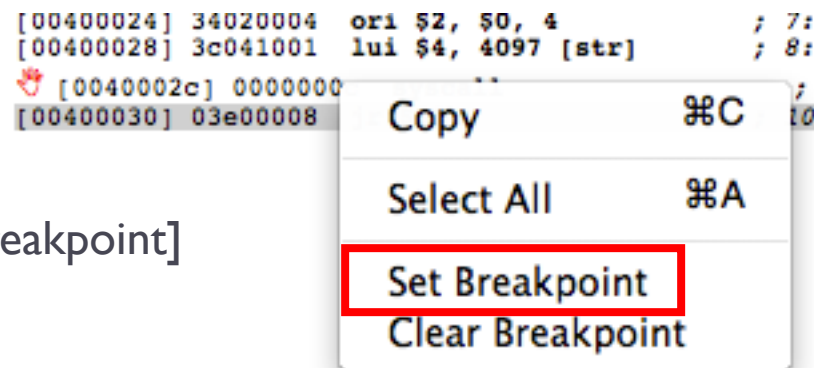
- ▶ プログラムのステップ実行

- ▶ 1命令ずつ実行する
- ▶ プログラムの読み込み後
  - ▶ [Single Step] ボタン → [Single Step] ボタンを繰り返しクリック



- ▶ ブレークポイントを設定

- ▶ 実行中に停止させたい位置を指定する
- ▶ 指定したい行の上で右クリック → [Set Breakpoint]



- ▶ 興味があれば、その他のボタンの挙動を調査

# 加減算

---

- ▶ `add $A, $B, $C`

- ▶  $\$A \leftarrow \$B + \$C$

- ▶ `addi $A, $B, 数値`

- ▶ 即値可算 (add immediate)

- ▶ レジスタが示す値に定数を加算

- ▶  $\$A \leftarrow \$B + \text{数値}$

- ▶ `sub $A, $B, $C`

- ▶  $\$A \leftarrow \$B - \$C$

- ▶ `subi`は無い

- ▶ `addi`で数値に負の値を指定

```
add $t0, $t1, $t2
addi $t0, $t1, 4
sub $t0, $t1, $t2
addi $t0, $t1, -16
```

# 加減算

---

```
.data
```

```
.text
```

```
main:
```

```
li $t0, 1
```

```
li $t1, 2
```

```
add $t0, $t0, $t1
```

```
li $v0, 1
```

```
move $a0, $t0
```

```
syscall
```

```
jr $ra
```

→ レジスタ \$t0 の値を\$a0に  
コピー

# コード例

- ▶ 右のアセンブリプログラムはどのような処理を行うプログラムか？

```
.data
AQ:
    .asciiz "A?:"
NL:
    .asciiz "¥n"

.text
main:
    li $v0, 4
    la $a0, AQ
    syscall

    li $v0, 5
    syscall

    add $a0, $v0, $v0
    li $v0, 1
    syscall

    li $v0, 4
    la $a0, NL
    syscall

    jr $ra
```

# コード例

```
#define AQ "A?:"  
#define NL "¥n"
```

```
int main() {  
    int a0, v0;
```

```
    printf("%s", AQ);
```

```
    scanf("%d", &v0);
```

```
    a0 = v0 + v0;  
    printf("%d", a0);
```

```
    printf("%s", NL);
```

```
    return 0;
```

```
}
```

AQ:

.data

.asciiz "A?:"

NL:

.asciiz "¥n"

.text

main:

```
li $v0, 4  
la $a0, AQ  
syscall
```

```
li $v0, 5  
syscall
```

```
add $a0, $v0, $v0  
li $v0, 1  
syscall
```

```
li $v0, 4  
la $a0, NL  
syscall
```

```
jr $ra
```

# よく使う命令

---

- ▶ 分岐命令 (ジャンプ命令)
  - ▶ j, jr
- ▶ 条件分岐命令
  - ▶ beq, bne, blt, ble, bgt, bge
- ▶ 比較命令
  - ▶ slt, slti

# 分岐命令(1/2)

## ▶ j label

- ▶ ラベルの命令へジャンプ

```
        j next
        :
next:    :
        :
```

m3.s

```
J:      .data
        .asciiz "Jump¥n"
NJ:     .asciiz "Not Jump¥n"

        .text
main:   j jump

        li $v0, 4
        la $a0, NJ
        syscall
        jr $ra

jump:   li $v0, 4
        la $a0, J
        syscall
        jr $ra
```

※ labelの有無  
に関係なく、  
jump命令が呼び  
出されるまで、次  
の命令が実行さ  
れ続ける

Jump



# 分岐命令(2/2)

## ▶ jr \$A

- ▶ レジスタ \$A の値の指すアドレスにジャンプ
- ▶ 例: jr \$ra

```

        la    $t0, next
        jr    $t0
        :
next:
```

```

J:      .data
        .asciiz "Jump¥n"
NJ:     .asciiz "Not Jump¥n"

        .text
main:
        la $t0, jump
        jr $t0

        li $v0, 4
        la $a0, NJ
        syscall
        jr $ra

jump:   li $v0, 4
        la $a0, J
        syscall
        jr $ra
```

Jump

# 条件分岐命令

---


- ▶ **beq \$A, \$B, label**
  - ▶ branch on equal
  - ▶ \$A == \$B ならラベルにジャンプ
- ▶ **bne \$A, \$B, label**
  - ▶ branch on not equal
  - ▶ \$A != \$B ならラベルにジャンプ

```
        beq    $t0, $t1, Label
        :
Label:
        :
```

# if 文の実現

---

```
if (x != 0)
    y = 1;
else
    y = 2;
```



```
        bne    $t0, $zero, then    # if (x!=0) goto then
        li     $t1, 2              # y=2
        j      end
then:
        li     $t1, 1              # y=1
end:
```

( $\$t0$ にx,  $\$t1$ にyが該当する)

# while 文の実現

---

```
while (x != y) {  
    y++;  
}
```



```
while (true) {  
    if (x==y) break;  
    y++;  
}
```



```
while:  
    beq    $t0, $t1, end    # if (x==y) goto end  
    addi   $t1, $t1, 1      # y++  
    j      while  
end:
```

(\$t0にx, \$t1にyが該当する)

# 比較命令

---

## ▶ **slt \$A, \$B, \$C**

- ▶ set less than

- ▶  $\$B < \$C$  なら  $\$A = 1$ ; そうでなければ  $\$A = 0$

## ▶ **slti \$A, \$B, 数値**

- ▶ set less than immediate

- ▶  $\$B < \text{数値}$  なら  $\$A = 1$ ; そうでなければ  $\$A = 0$

# その他の条件分岐命令

---

<code>blt \$A, \$B, label</code>	$\$A < \$B$ なら分岐
<code>ble \$A, \$B, label</code>	$\$A \leq \$B$ なら分岐
<code>bgt \$A, \$B, label</code>	$\$A > \$B$ なら分岐
<code>bge \$A, \$B, label</code>	$\$A \geq \$B$ なら分岐

**(less than, less than equal, greater than, greater than equal)**

- ▶ これらは疑似命令
  - ▶ `slt, beq, bne` の組み合わせで実現できる (⇒課題)
  - ▶ 「`move $A, $B`」も疑似命令
    - ▶ `add A, B, $zero`
- ▶ その他の命令
  - ▶ [http://www.cs.wisc.edu/~larus/HP\\_AppA.pdf](http://www.cs.wisc.edu/~larus/HP_AppA.pdf) の A.10 (A-51) 以降にその他の命令が載っている

# 配列

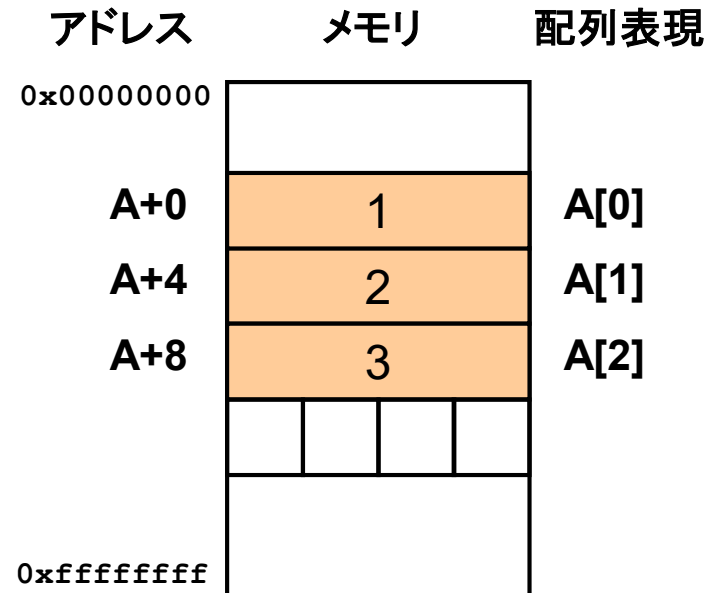
## ▶ ワードの配列

- ▶ `.word` でワード (4 byte) の数値を定義できる
  - ▶ MIPSでは、1 ワード(語) = 4バイト = 32ビット
    - 1バイト = 8ビット

```
A:      .data
        .word 1 2 3
```

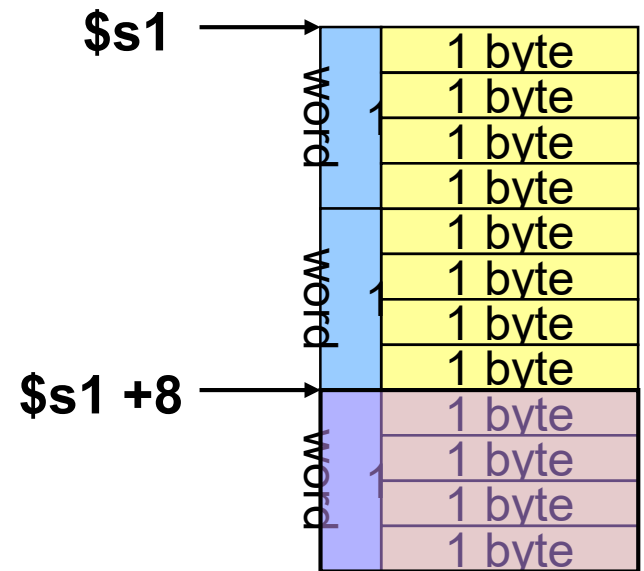
- ▶ 要素 `A[i]` にはアドレス  $A+i*4$  が対応

- ▶ ワード単位でデータは格納されるが、アドレッシングはバイト単位で表現



# メモリアクセス命令

- ▶ `lw $A, X($B)`
  - ▶ `$A = メモリ[X + $B]`
    - ▶ `$A, $B`: レジスタ、`X`: 定数 (ラベル or 数字)
    - ▶ メモリ上のアドレス「`X + $B`」から始まる1ワード(4バイト)のデータをレジスタ`$A`に転送する
  - ▶ 例: `lw $t0, 8($s1)`
    - ▶ `$s1`が示すアドレスから2ワード先のデータ(1ワード分)を`$t0`に読み込む
- ▶ `sw $A, X($B)`
  - ▶ `メモリ[X + $B] = $A`
    - ▶ `$A, $B`: レジスタ、`X`: 定数 (ラベル or 数字)
    - ▶ レジスタ`$A`の値 (1ワード)をメモリ上のアドレス「`X + $B`」に転送する
  - ▶ 例: `sw $t0, 8($s1)`
    - ▶ `$s1`が示すアドレスから2ワード先のデータ(1ワード分)に`$t0`の値を書き込む

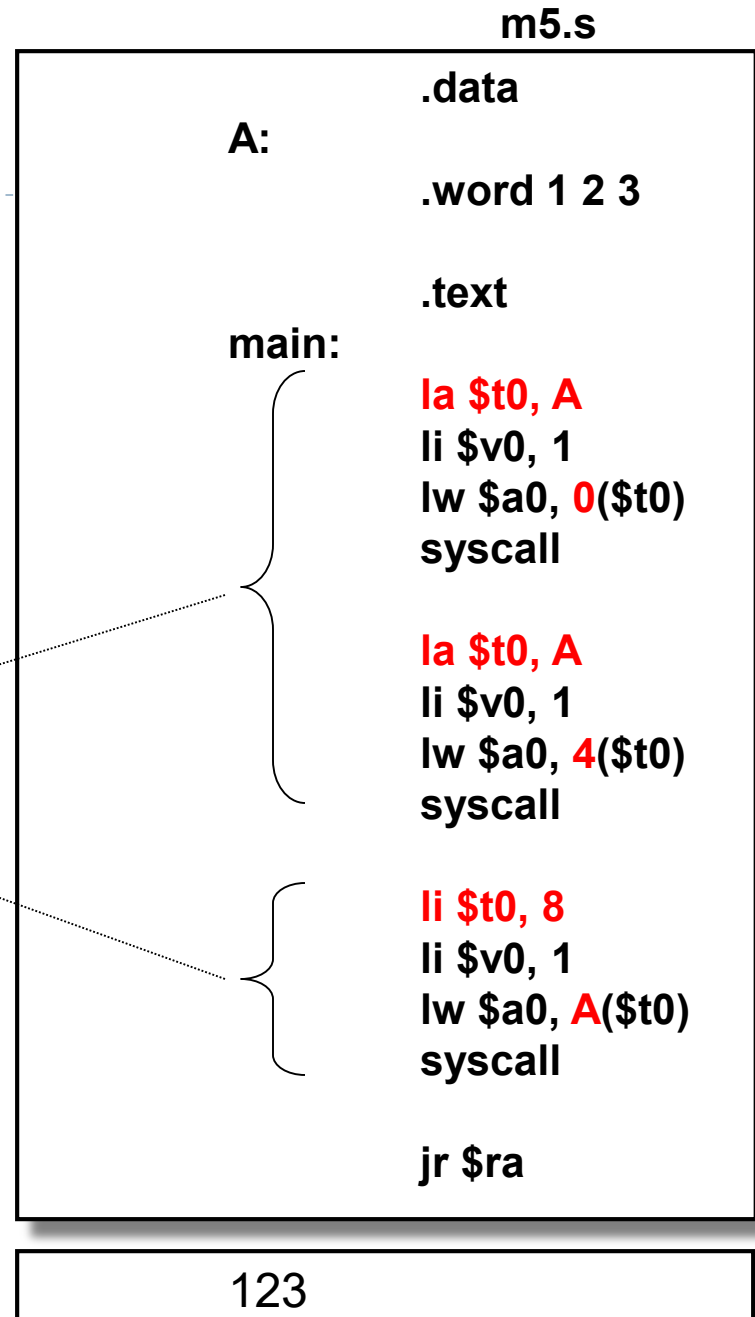




# 配列の操作 (例)

- ▶ 配列Aの値を表示するプログラム

どちらでもよい



# 今日の課題

# 課題1：bltの実装

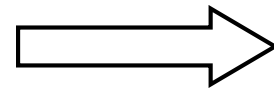
---

- ▶ sltを用いてblt (branch if less than) を実装せよ
  - ▶ 「blt \$s1, \$s2, label」を対象に、一時レジスタとして\$t0を使う
  - ▶ bltを実装したコード断片をレポートに記述し、その解説をすること
- ▶ 注意点
  - ▶ 短いコードで記述されたもの程よい。
- ▶ オプション課題 (課題1-2)
  - ▶ sltを用いて ble (branch if less than or equal)を実装せよ

## 課題2: 2つの配列の要素の和

---

- ▶ これら2つの配列の  $i$  番目と  $3-i$  番目の要素どうしを足し合わせ、表示せよ
- ▶ .wordとして、2つの配列を定義
  - ▶  $A = \{ 1, 2, 3, 4 \}$
  - ▶  $B = \{ 5, 6, 7, 8 \}$
- ▶ 注意点
  - ▶  $A[0] + B[3], A[1] + B[2], A[2] + B[1], A[3] + B[0]$
  - ▶ ループ処理で実装すること
  - ▶ syscallを使ってコンソールに表示すること
- ▶ オプション課題 (課題2-2)
  - ▶ 任意長の配列で同様の動作を実現するプログラムを書け
    - ▶ 配列長の与え方は各自で定義すること



9999

# 配列長の与え方「各自で定義する」の補足

- ▶ 本当に好きに決めてもらって構いません
  - ▶ 本人が「これってアリかよ?」と思うほどに余程変な決め方じゃなければ
  - ▶ 課題の意図としては「配列長が4である」ということに強く依存していないコードを書けますよね という点だけです
- ▶ 例
  - ▶ これを使っていたとしても、これ以外の定義方法でも可

```
                .data
A:
                .word 1 2 3
B:
                .word 4 5 6
L: # 配列長
                .word 3
```

```
                .data
A:
                .word 1 2 3 -1
B:
                .word 4 5 6 -1
# 配列の終端を-1とする
```

# 補足

---

## 課題2

```
    .data
A:    .word 1 2 3 4
B:    .word 5 6 7 8

    .text
main:
    #和を表示
```

# アセンブリプログラムの書き方の補足 (1/2)

---

- ▶ 意味の切れ目で改行を入れる
  - ▶ SPIM は空行を無視する
- ▶ コメントを書く
  - ▶ # 以降はコメントになる

```
li    $v0, 5
syscall

move  $a0, $v0
li    $v0, 1
syscall
```

```
# println "HelloWorld"
li    $v0, 4
la    $a0, str
syscall                                # print_string
```

# アセンブリプログラムの書き方の補足 (2/2)

- ▶ 行頭のスペースは無くてもよい
  - ▶ あるほうがプログラムが見やすくなる
  - ▶ 命令中には適切にスペースを入れる必要がある(数は任意)

```
.ascii□ 'HelloWorld¥n"  
li □ $v0, 4
```

- ▶ データが無いときはデータセグメントの記述は省略できる

```
.data  
.text  
main:  
li $t0, 4
```



```
.text  
main:  
li $t0, 4
```



# 課題提出

---

- ▶ 〆切: 2018/01/07 (火) 23:59
  - ▶ OCW-iから提出すること
  - ▶ 遅れても(減点しますが)受け付けます。
- ▶ 提出物: 以下のファイルを1つのファイルにzip圧縮したもの
  - ▶ ドキュメント (pdf, txt 形式)
    - ▶ 各課題の実行結果
    - ▶ プログラムソースの簡単な説明、工夫したところ等
    - ▶ プログラムの実行結果
    - ▶ 感想、質問等
  - ▶ プログラムソース
    - ▶ テスト用のmain関数も含む(課題1, 1-2)
  - ▶ 全てのファイル名は半角英数字でお願いします
    - ▶ レポートのファイルを含む、文字化け防止のため

# 課題締め切り

---

## ▶ 第01回

▶ 12/13 (金) 本日 日本時間 23:59 まで

▶ 遅れても減点しますが受け付けます

## ▶ 第02回

▶ 12/20 (金)

## ▶ 第03回

▶ 1/7 (火)

## ▶ 第04回

▶ 1/7 (火)