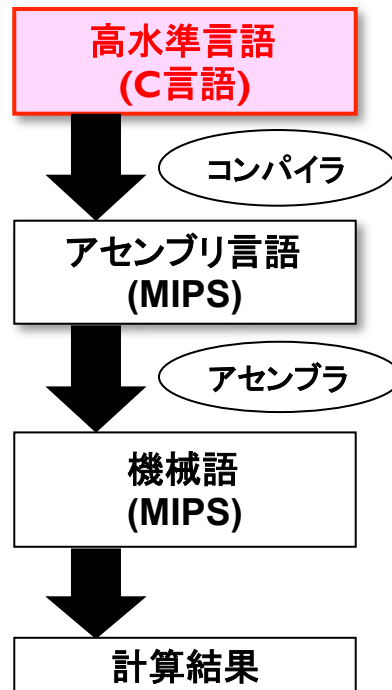


2012年度 計算機システム演習
第3回

2012.04.27

白幡 晃一



今日の内容

続・C言語(関数ポインタ)
アセンブラ言語

九九の掛け算表

sample24.c

```
#include <stdio.h>

int mul(int x, int y){
    return x * y;
}

void kuku_mul() {
    int i, j;
    for (i = 1; i <=9; i++) {
        for (j = 1; j <= 9; j++) {
            printf("%3d", mul(i, j));
        }
        printf("\n");
    }
}

int main(){
    kuku();
}
```

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

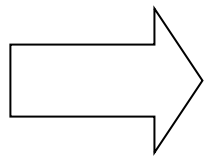
- ▶ このプログラムに足し算 (`add(i,j)`) 表を追加したい場合どうすればよいか？
-

九九の掛け算 / 足し算表

- ▶ `void kuku_sum()` 関数を追加し内部で `sum(x, y)` 関数を呼び出す
 - ▶ 引き算表、割り算表を追加したい場合どうすべきか？
⇒ さらに `kuku_sub`, `kuku_div` を作る？

```
int sum(int x, int y){
    return x + y;
}

void kuku_sum() {
    int i, j;
    for (i = 1; i <= 9; i++) {
        for (j = 1; j <= 9; j++) {
            printf("%3d", sum(i, j));
        }
        printf("\n");
    }
}
```



ここでは、関数ポインタを用いてより汎用性のある関数を定義する方法を紹介



関数ポインタ

- ▶ 関数のアドレスを保持するデータ型のことを**関数ポインタ**と言う
 - ▶ 関数定義も実行時にメモリ上に置かれる。そのため関数にもアドレスが存在するのは自然

- ▶ **関数(への)ポインタ**

- ▶ 関数のアドレスを保持し、間接的にその関数を呼び出す
 - ▶ ポインタ型なので勿論、格納する値はアドレス
 - ▶ c.f.) `int *p` ⇒ 間接的に変数へアクセス
- ▶ 宣言方法

```
int (*fp)(int, int);
```

fp: 関数ポインタ

- ▶ `fp`は`int`型の引数を2つ取り、`int`型を返す関数へのポインタ
⇒ `fp`は`int`型の引数を2つ取り、`int`型を返す**任意**の関数のアドレスを保持できる
-



関数ポインタの例

sample25.c

```
#include <stdio.h>

int sum(int x, int y){ return x + y; }
int sub(int x, int y){ return x - y; }
int mul(int x, int y){ return x * y; }
int div(int x, int y){ return x / y; }

int main() {
    int x = 10, y = 2;
    int (*calc)(int , int);

    calc = &sum;
    printf("calc(%d, %d) = %d\n", x, y, (*calc)(x, y));
    calc = &sub;
    printf("calc(%d, %d) = %d\n", x, y, (*calc)(x, y));
}
```

```
calc(10, 2) = 12
calc(10, 2) = 8
```

```
int v[] = {1,2,3};
int *pv;
pv = v; //pv = &v
のようなイメージ
```

関数ポインタの宣言

calc: 関数sumへの関数ポインタ
※ calc = sumでも可

calc: 関数subへの関数ポインタ
※ calc = subでも可

関数ポインタを引数にとる関数

- ▶ C言語では関数ポインタを用いることにより関数を引数とする関数を定義できる

sample26.c

```
#include <stdio.h>

int sum(int x, int y){ return x + y; }
int sub(int x, int y){ return x - y; }
int mul(int x, int y){ return x * y; }
int div(int x, int y){ return x / y; }

int calc_upto3(int (*calc)(int, int)) {
    int i;
    for (i = 1; i <= 3; i++) {
        printf("calc(%d, %d) = %d\n", i, i, (*calc)(i, i));
    }
}

int main() {
    calc_upto3(&sum); // sumでも可能
    calc_upto3(&mul); // mulでも可能
}
```

```
calc(1, 1) = 2
calc(2, 2) = 4
calc(3, 3) = 6
calc(1, 1) = 1
calc(2, 2) = 4
calc(3, 3) = 9
```

関数ポインタの省略記法

関数の仮引数として宣言する場合(*)を省ける

- ▶ ローカル変数として宣言する場合はダメ

```
void calc_upto10(int calc(int, int)) {  
    int i, result;  
    for (i = 1; i <= 10; i++) {  
        printf("calc(%d, %d) = %d\n", i, i, calc(i, i));  
    }  
}
```

通常の間数と同じように呼び出せる。直感的でgood !

注意: int (* calc)(int, int)と記述する場合は()は省略できない

- ▶ int *calc (int, int) => int * (calc)(int, int)と解釈される

和 / 差 / 積 / 商表

sample27.c

```
#include <stdio.h>

int sum(int x, int y){ return x + y; }
int sub(int x, int y){ return x - y; }
int mul(int x, int y){ return x * y; }
int div(int x, int y){ return x / y; }

int kuku(int nx, int ny, int calc(int, int)) {
    int i, j;
    for (i = 1; i <= nx; i++) {
        for (j = 1; j <= ny; j++) {
            printf("%3d", calc(i, j));
        }
        printf("\n");
    }
}

int main() {
    kuku(3, 3, sum);
    kuku(5, 5, mul);
}
```

```
2 3 4
3 4 5
4 5 6
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25
```

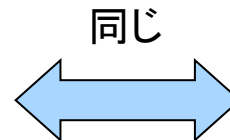
typedef (再掲)

- ▶ 既存の型に対して同義語を与える宣言

```
typedef <既存型> <新規型>;
```

- ▶ 例) `typedef int NUMBER;` `typedef unsigned long size_t`
- ▶ メリット1: 読みやすさ・書きやすさ向上
 - ▶ 毎回 `struct data` と書く必要がない

```
struct _data {  
    int key;  
    int val;  
};  
typedef struct _data data;
```



```
typedef struct {  
    int key;  
    int val;  
} data;
```

- ▶ メリット2: コードに影響を与えず、既存型を置き換えられる
 - ▶ `typedef int NUMBER;` ⇒ `typedef double NUMBER;`
-



おまけ1: typedefと関数ポインタ

sample28.c

```
#include <stdio.h>

typedef int (*Calc)(int, int);

int sum(int x, int y){ return x + y; }
int sub(int x, int y){ return x - y; }
int mul(int x, int y){ return x * y; }
int div(int x, int y){ return x / y; }

int main() {
    int x = 10, y = 2;
    Calc calc;

    calc = sum;
    printf("calc(%d, %d) = %d\n", x, y, calc(x, y));
    calc = sub;
    printf("calc(%d, %d) = %d\n", x, y, calc(x, y));
}
```

```
calc(10, 2) = 12
calc(10, 2) = 8
```

← int型の引数を2つ取り、int型を返す関数を指すポインタ型にCalcと言う名前をつける

← calcは
int型の引数を2つ取り、int型を返す関数を指すポインタ
⇒毎回 `int (*calc) (int, int)`と書く必要がない

おまけ2: typedefと関数ポインタの配列

sample29.c

```
#include <stdio.h>

typedef int (*Calc)(int, int);

int sum(int x, int y){ return x + y; }
int sub(int x, int y){ return x - y; }
int mul(int x, int y){ return x * y; }
int div(int x, int y){ return x / y; }

int main() {
    int x = 10, y=2;
    Calc calc[] = {sum, sub, mul, div};
    int i;
    for (i = 0; i < 4; i++) {
        printf("calc(%d, %d) = %d\n", x, y, calc[i](x,y));
    }
}
```

```
calc(10, 2) = 12
calc(10, 2) = 8
calc(10, 2) = 20
calc(10, 2) = 5
```

calcは
int型の引数を2つ取り、int型を返す関数を指すポインタの配列

calc[0] : sum
calc[1] : sub
calc[2] : mul
calc[3] : div

おまけ3：（int型引数を2つ取り）intを返す関数ポインタの配列を表す型Tの定義

sample30.c

```
#include <stdio.h>

typedef int (*Calc)(int, int);
typedef Calc T[];

int sum(int x, int y){ return x + y; }
int sub(int x, int y){ return x - y; }
int mul(int x, int y){ return x * y; }
int div(int x, int y){ return x / y; }

int main() {
    int x = 10, y=2;
    T calc = {sum, sub, mul, div};
    int i;
    for (i = 0; i < 4; i++) {
        printf("calc(%d, %d) = %d\n", x, y, calc[i](x,y));
    }
}
```

Tはint型の引数を2つ取り、int型を返す関数を指すポインタの配列を表す型

calc[]とは書かない
※T自体が配列を意味する

関数ポインタの応用例

- ▶ 汎用的な関数の定義

- ▶ 例)


```
qsort(void *base, size_t num, size_t size,  
      int (*compar)(const void *a, const void *b));
```

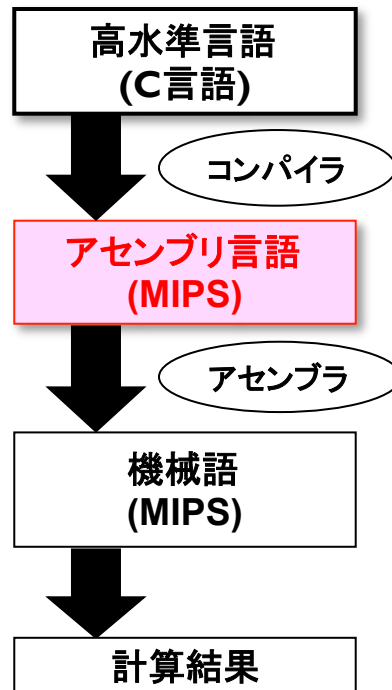
- ▶ 関数ポインタ comparは配列base中の2つのデータの大小比較を行なう関数へのポインタ
 - ▶ aを先⇒負の値、どちらでもよい⇒0、bが先⇒正の値

- ▶ あらかじめ登録した任意の関数をまとめて実行

- ▶ 例)

```
int atexit(void (*func)(void));
```

- ▶ 関数ポインタfuncが指す関数を登録し、プログラム終了時にそれらをまとめて実行する
-
- 

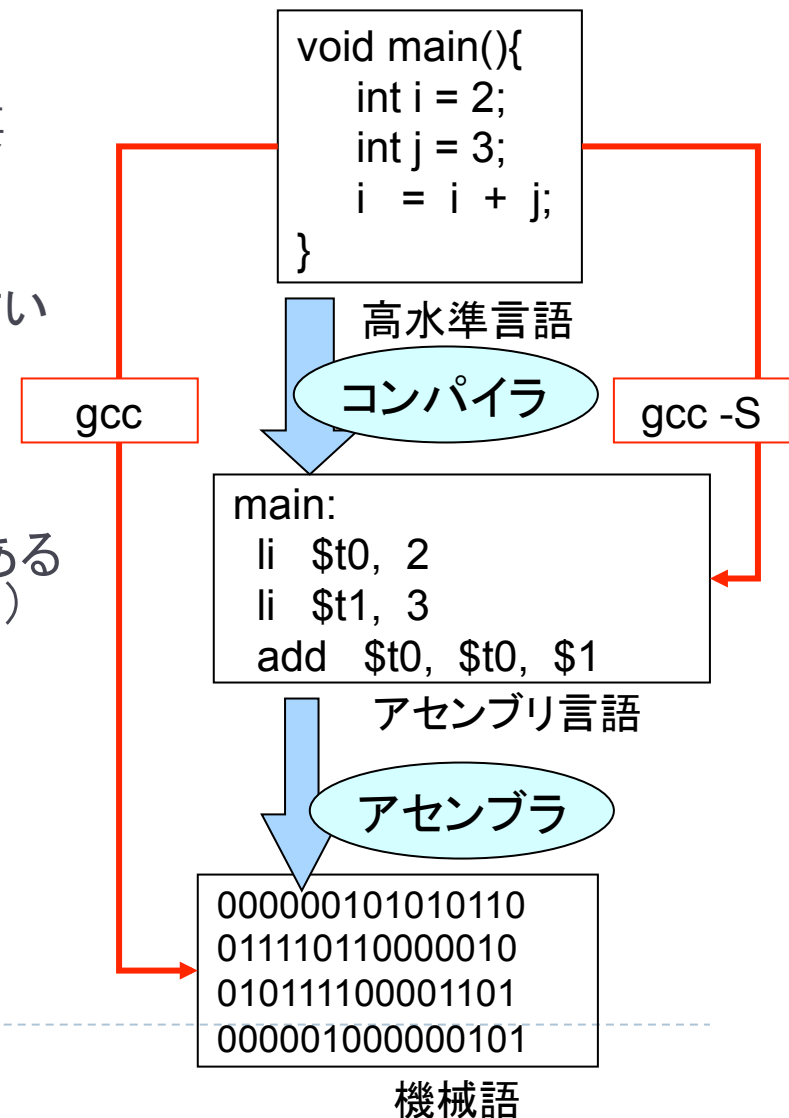


今日の内容

続・C言語(関数ポインタ)
アセンブラ言語

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

```
.data  
str:  
.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

main:

li \$v0, 4

レジスタ \$v0 に 4 を代入 (\$v0 = 4)

la \$a0, str

レジスタ \$a0 に str を代入 (\$a0 = str)

syscall

システムコールを呼ぶ

jr \$ra

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	返り値	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://www.cs.wisc.edu/~larus/spim.html>
- ▶ Windows, Mac OS X, Linux 版

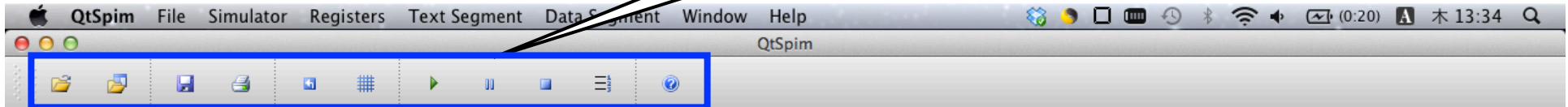
- ▶ インストール & 利用方法

- ▶ 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 Windows PC
 - ▶ <http://sourceforge.net/projects/spimsimulator/files/>
 - ▶ QtSpim_*_Windows.zip をダウンロード
 - QtSpim_9.1.7_Windows.zip など
 - ▶ 解凍 => setup.exe を実行

基本的に西7のMacを用いる。選択肢2は、家で課題をやりたい学生向け

QtSpim

制御ボタン



FP Regs Int Regs [16] Data Text

Int Regs [16]

```
PC = 0
EPC = 0
Cause = 0
BadVAddr = 0
Status = 3000fff10

HI = 0
LO = 0

R0 [r0] = 0
R1 [at] = 0
R2 [v0] = 0
R3 [v1] = 0
R4 [a0] = 0
R5 [a1] = 0
R6 [a2] = 7fffffe28
R7 [a3] = 0
R8 [t0] = 0
R9 [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
R28 [gp] = 10008000
R29 [sp] = 7fffffe28
R30 [s8] = 0
R31 [ra] = 0
```

User Text Segment [00400000]..[00440000]

```
[00400000] 8fa40000 lw $4, 0($29)
[00400004] 27a50004 addiu $5, $29, 4
[00400008] 24a60004 addiu $6, $5, 4
[0040000c] 00041080 sll $2, $4, 2
[00400010] 00c23021 addu $6, $6, $2
[00400014] 0c100009 jal 0x00400024 [main]
[00400018] 00000000 nop
[0040001c] 3402000a ori $2, $0, 10
[00400020] 0000000c syscall
[00400024] 34020004 ori $2, $0, 4
[00400028] 3c041001 lui $4, 4097 [str]
[0040002c] 0000000c syscall
[00400030] 03e00008 jr $31
[00400034] 34020004 ori $2, $0, 4
[00400038] 3c011001 lui $1, 4097 [AQ]
[0040003c] 3424000c ori $4, $1, 12 [AQ]
[00400040] 0000000c syscall
[00400044] 34020005 ori $2, $0, 5
[00400048] 0000000c syscall
[0040004c] 00422020 add $4, $2, $2
[00400050] 34020001 ori $2, $0, 1
[00400054] 0000000c syscall
[00400058] 34020004 ori $2, $0, 4
[0040005c] 3c011001 lui $1, 4097 [NL]
[00400060] 34240010 ori $4, $1, 16 [NL]
[00400064] 0000000c syscall
[00400068] 03e00008 jr $31
```

Kernel Text Segment [80000000]..[80010000]

```
[80000180] 0001d821 addu $27, $0, $1
[80000184] 3c019000 lui $1, -28672
[80000188] ac220200 sw $2, 512($1)
[8000018c] 3c019000 lui $1, -28672
[80000190] ac240204 sw $4, 516($1)
[80000194] 401a6800 mfc0 $26, $13
[80000198] 001a2082 srl $4, $26, 2
[8000019c] 3084001f andi $4, $4, 31
[800001a0] 34020004 ori $2, $0, 4
[800001a4] 3c011001 lui $4, -28672 [__m1_]
[800001a8] 0000000c syscall
[800001ac] 34020001 ori $2, $0, 1
[800001b0] 34020004 ori $2, $0, 4
[800001b4] srl $a0 $k0 2 # Extract ExcCode Field
[800001b8] andi $a0 $a0 0x1f
[800001bc] syscall
[800001c0] li $v0 4 # syscall 4 (print_str)
[800001c4] andi $a0 $k0 0x3c
[800001c8] 3c019000 lui $1, -28672
[800001cc] 8c240180 lw $4, 384($1)
[800001d0] 00000000 nop
[800001d4] 0000000c syscall
[800001d8] 34010018 ori $1, $0, 24
[800001dc] 143a0008 bne $1, $26, 32 [ok_pc-0x800001dc]
[800001e0] 00000000 nop
```

プログラム

レジスタ表示領域

エラー出力など

See the file README for a full copyright notice.
spim: (parser) Label is defined for the second time on line 8 of file /Users/shirahata/Documents/æ¥/ã|é"ã°éççç@/è"çç@æ@ã-ã"ää /2012/MIPS/m2.s
main:
^

Hello World (1/3)

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

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

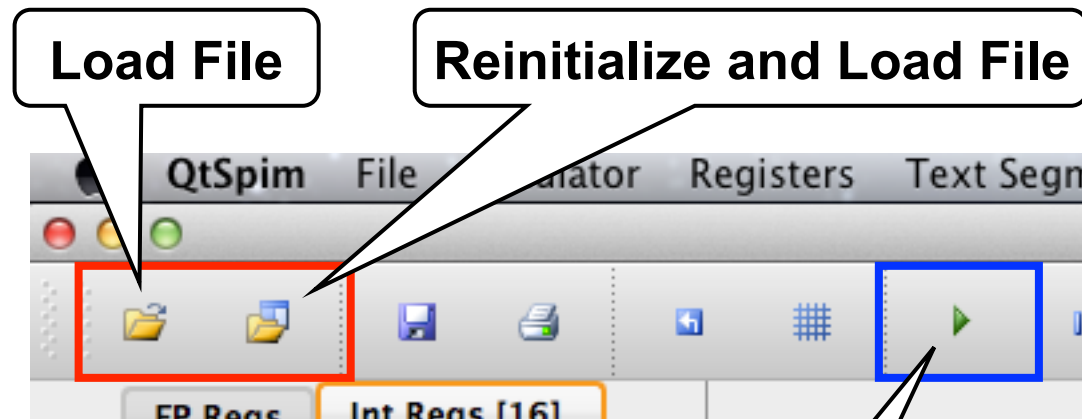
                .text
main:           li $v0, 4
                la $a0, str
                syscall
                jr $ra
```



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

Single Step



- ▶ 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

```
[00400024] 34020004 ori $2, $0, 4 ; 7:  
[00400028] 3c041001 lui $4, 4097 [str] ; 8:  
* [0040002c] 00000000 ; 9:  
[00400030] 03e00008 ; 10:
```

Copy	⌘C
Select All	⌘A
Set Breakpoint	
Clear Breakpoint	

本日の課題

課題1 (1/2)

- ▶ 足し算 or 引き算を行うアセンブリを記述せよ
 - ▶ add Dest, Src1, Src2
 - ▶ $\text{Dest} = \text{Src1} + \text{Src2}$
 - ▶ 例) add \$v0, \$v0, \$v1
 - ▶ sub Dest, Src1, Src2
 - ▶ $\text{Dest} = \text{Src1} - \text{Src2}$
 - ▶ 例) sub \$v0, \$v0, \$v1



課題1 (2/2)

```
.data
```

```
.text
```

```
main:
```

```
li $t0, 数值1
```

```
li $t1, 数值2
```

```
< 計算命令 >
```

```
li $v0, 1
```

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

```
syscall
```

```
jr $ra
```



レジスタ \$t0 の値を\$a0に
コピー



課題2

- ▶ 右のアセンブリプログラムを実行せよ。また、どのような処理を行うプログラムか？

m2.s

```
AQ:      .data
         .ascii "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
```

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 は改行を無視する
- ▶ 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
```

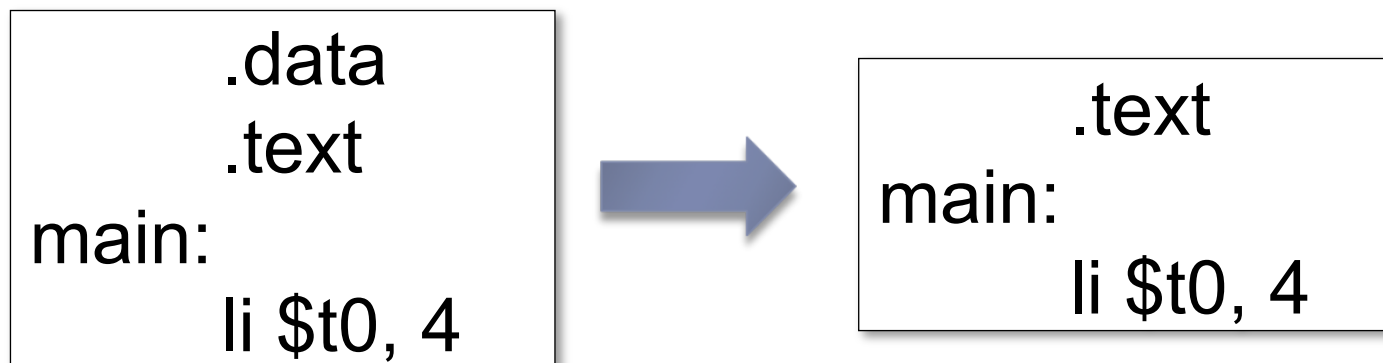
```
# println "HelloWorld"
li    $v0, 4
la    $a0, str
syscall                                # print_string
```

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

```
.asciiz '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



課題提出

- ▶ 〆切: 5/18(金) 23:59
- ▶ 提出物: 以下のファイルを圧縮したもの
 - ▶ ドキュメント(pdf, plain txt, wordなんでも可)
 - ▶ 課題1,2の実行結果
 - ▶ 課題2の解答
 - ▶ 感想、質問等
 - ▶ プログラムソース (課題1のみでよい)

