Lab 3

Objectives: This lab shows you some basic techniques and syntax to write a MIPS program. Syntax includes system calls, load address instruction, load integer instruction, and arithmetic instructions (e.g. addition, subtraction, etc).

MIPS Programming – Part I

Recall the one we executed in the first lab, the ten lines "Hello World" MIPS program. For the ten program statements, they are:

```
        .data
msg:    .asciiz "Hello World"
        .extern foobar 4
        .text
        .globl main
main:   li $v0, 4         # syscall 4 (print_str)
        la $a0, msg       # argument: string
        syscall           # print the string
        lw $t1, foobar

        jr $ra            # retrun to caller
```

In this example, we have a `string` (char array) stored in "`msg`". Then, we use a `syscall` (system call or kernel call) to instruct the simulator to print out the message stored in "`msg`". First of all, let's take a look at the program structure.

Program Structure

Similar to most high level programming languages, MIPS have to declare variable names in the "`.data`" (assembler directive) section. In addition, program codes (instructions) are placed in "`.text`" (another assembler directive) section. You can place your comments in a program at anywhere by following the symbol "`#`" in a line. Below is a MIPS program template.

```
# Comment giving name of program and description of function
# File Name: Template.s
# Description: Bare-bones outline of MIPS assembly language program
# Author: Put your name here

            .data       # variable declarations follow this line
                        # ...

            .text       # instructions follow this line

main:                   # indicates start of code (first instruction to execute)
                        # ...

# End of program, leave a blank line afterwards to make SPIM happy.
```

Directive

A directive is a message to the assembler that tells the assembler something it needs to know in order to carry out the assembly process. This includes noting where the data is declared or the code is defined. (*Note: Assembler directives are not executable statements.*)

Some common assembler directives you should know.

Syntax	**Description**
`.data <addr>`	The following data items should be stored in the data segment. If the optional argument *`addr`* is present, the items are stored beginning at address *`addr`*.
`.asciiz str`	Store the string in memory and null-terminate it.
`.text <addr>`	The next items are put in the user text segment. In SPIM, these items may only be instructions or words (see the *`.word`* directive below). If the optional argument *`addr`* is present, the items are stored beginning at address *`addr`*.
`.globl sym`	Declare that symbol *`sym`* is global and can be referenced from other files. For example, `.globl main` means that the identifier `main` will be used outside of this source file (i.e. used *globally*) as the label of a particular location in main memory.
`.space n`	Allocate *n* bytes of space in the current segment (which must be the data segment in SPIM).
`.word w1,..., wn`	Store the *n* 32-bit quantities in successive memory words. SPIM does not distinguish various parts of the data segment (`.data`, `.rdata` and `.sdata`).
`.byte b1,..., bn`	Store the *n* values in successive bytes of memory.
`.extern sym size`	Declare that the datum stored at sym is size bytes large and is a global symbol. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register `$gp`.

Besides, those directives, there are "`li`", "`la`", "`lw`" and "`jr`" in the program. All these are the instructions (*executable codes*) of MIPS.

Data Types / Sizes

`byte`	8-bit integer
`halfword`	16-bit integer
`word`	32-bit integer
`float`	32-bit floating-point number
`double`	64-bit floating-point number

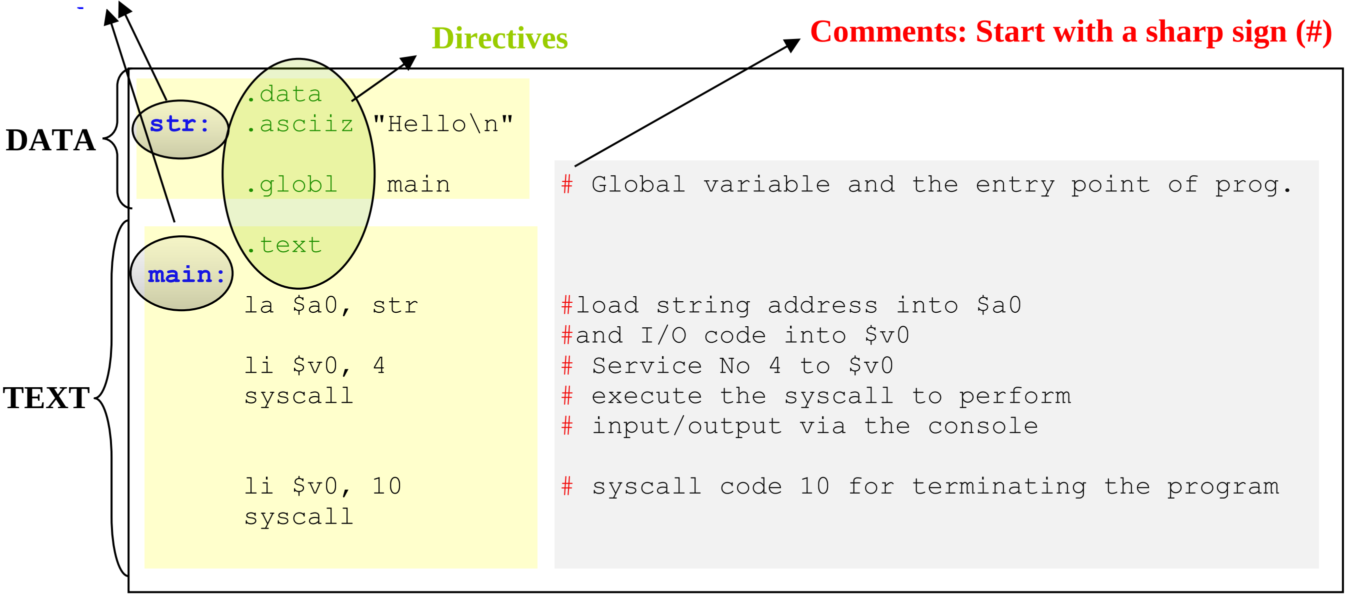
CPU Registers

Register Name	Register Number	Register Usage
`$zero`	`$0`	Hardware set to 0
`$at`	`$1`	Assembler temporary (reserved by the assembler)
`$v0 - $v1`	`$2 - $3`	Function result (low/high)
`$a0 - $a3`	`$4 - $7`	Argument Registers – First four parameters for subroutine. Not preserved across procedure calls.
`$t0 - $t7`	`$8 - $15`	Temporary registers – Caller saved if needed. Subroutines can use without saving. Not preserved across procedure calls.
`$s0 - $s7`	`$16 - $23`	Saved registers – Callee saved. A subrountine using one of these must save original and restore it before exiting. Preserved across procedure calls.
`$t8 - $t9`	`$24 - $25`	Temporary registers. (These are in addition to `$t0 - $t7` above.)
`$k0 - $k1`	`$26 - $27`	Reserved for OS kernel (use by interrupt/trap handler)
`$gp`	`$28`	Global pointer – points to the middle of the 64K block of memory in the static data segment.
`$sp`	`$29`	Stack pointer – points to last location on the stack.
`$fp`	`$30`	Frame pointer – saved value. Preserved across procedure calls
`$ra`	`$31`	Return address

Quick Review

ID (Identifier): a sequence of char, underbar(“_”), and dots that does not begin with a

Labels: put at the beginning of a line followed by a



Comments: Start with a sharp sign (#)

```
        .data
str:    .asciiz "Hello\n"
        .globl  main       # Global variable and the entry point of prog.
        .text
main:
        la $a0, str        #load string address into $a0
                           #and I/O code into $v0
        li $v0, 4          # Service No 4 to $v0
        syscall            # execute the syscall to perform
                           # input/output via the console

        li $v0, 10         # syscall code 10 for terminating the program
        syscall
```

Basic Structure:

`.data`

Declare variables here

`.text`

Write the program body (instructions) here

System Calls

System Calls: Input/output integer, float, double, and string by `syscall`. For example, the following statement "move" a data value from one register to another one and then use a system call to print out the value.

```
move $a0, $t0    # move value from $t0 to $a0
li   $v0, 1      # use a system call to print out integer
syscall          # print the value on console
```

Service	Code in `$v0`	Arguments	Results
print_int	1	`$a0` = integer to be printed	
print_float	2	`$f12` = float to be printed	
print_double	3	`$f12` = double to be printed	
print_strint	4	`$a0` = address of string in memory	
read_int	5		integer returned in `$v0`
read_float	6		float returned in `$v0`
read_double	7		double returned in `$v0`
read_string	8	`$a0` = memory address of string input buffer `$a1` = length of string buffer (n)	
sbrk	9	`$a0` = amount	address in `$v0`
exit	10		

In most cases, we simply use those print / read (output / input) calls as well as the exit call. For "sbrk", it is used in Unix or Unix-like machines to management memory (`malloc`) dynamically in the old days.

Load / Store Instructions

Instruction Syntax	Description
`lw   register_destination, RAM_source`	Load word or copy word (4 bytes) at source RAM location to destination register.
`lb   register_destination, RAM_source`	Load word or copy byte at source RAM location to low-order byte of destination register.
`li   register_destination, value`	Load immediate value into destination register.
`sw   register_source, RAM_destination`	Store word (4 bytes) in source register location to RAM destination.
`sb   register_source, RAM_destination`	Store byte (low-order) in source register location to RAM destination.

Arithmetic Instructions

Instruction Syntax	Description
`add  $t0, $t1, $t2`	`# $t0 = $t1 + $t2;` add as signed integers
`sub  $t0, $t1, $t2`	`# $t0 = $t1 - $t2;` substract as signed integers
`addi  $t0, $t1, 5`	`# $t0 = $t1 + 5;` add immediate
`addu  $t0, $t1, $t2`	`# $t0 = $t1 + $t2;` add as unsigned integers
`subu  $t0, $t1, $t2`	`# $t0 = $t1 - $t2;` substract as unsigned integers
`mult  $t1, $t2`	`# (Hi,Lo) = $t1 * $t2;` multiply 32-bit quantities in `$t1` and `$t2`, and store 64-bit result in special registers `Lo` and `Hi`
`div  $t1, $t2`	`# Lo = $t1 / $t2;     (integer quotient)` `# Hi = $t1 mod $t2;   (remainder)`
`mfhi  $t0`	`# $t0 = Hi` move quantity in special register `Hi` to `$t0`
`mflo  $t0`	`# $t0 = Lo` move quantity in special register `Lo` to `$t0`
`move  $t1, $t2`	`# $t1 = $t2`

Example

BMI Calculator

```
#
# This program returns you the Body Mass Index (BMI) figures
#
# Procedures:
#1. Print message: "Enter Weight (whole pound): "
#2. Read the input integer from the console
#3. Print message: "Enter Height (whole inch): "
#4. Read the input integer from the console
#5. Calculate the BMI
#   6. Show the result on Console: "Your BMI is: "
#

   .data
str1: .asciiz "Enter Weight (whole pound): "
str2: .asciiz "Enter Height (whole inch): "
str3: .asciiz "Your BMI is: "

      .globl main       # Global variable: the entry point of the prog.

   .text
main:
 #
 #Step 1: Print the prompt message using system call 4
 #
 la $a0, str1     # load string address into $a0 and I/O code into $v0
 li $v0, 4
 syscall          # print the message on console

 #
 #Step 2: Read the integer from the console using system call 5
 #
 li $v0, 5
 syscall
 move $s0, $v0

 #
 #Step 3: Repeat Step 1 and Step to read in "Height"
 #
 la $a0, str2     # load string address into $a0 and I/O code into $v0
 li $v0, 4
 syscall          # print the message on console

 li $v0, 5
 syscall
 move $s1, $v0

 #
 #Step 4: Calculate the BMI (mass * 703) / (height)^2
 #
 li   $t0, 703
 mult $t0, $s0
 mflo $t1
 mult $s1, $s1
 mflo $t2
 div  $t1, $t2
```

```
mflo $s2


#
#Step 5: Print the result message using system call 4
#
la $a0, str3     # load string address into $a0 and I/O code into $v0
li $v0, 4
syscall          # print the message on console

#
#Step 6: Print the BMI
#
move $a0, $s2
li $v0, 1
syscall

li $v0, 10       # syscall code 10 for terminating the program
syscall
```

Exercise

Based on the example program above, rewrite one for converting the temperature in Celsius (C) to Fahrenheit (F).

Equation for the conversion: $F = (C * (9 / 5)) + 32$

- End -