

**CPSC 321:501-503 - Computer Architecture**  
**Texas A&M University**  
**Department of Computer Science**

**Fall 2006**

**Lab 2 (50 pts) – Introductory MIPS Assembly Code Programming**  
**Complete by yourself**

---

**Release date: 11 September 2006**  
**Due date: One week following lab**

---

## Objective

This laboratory assignment will help you understand loops, procedures, and the parameter passing conventions of the MIPS assembly language.

## Assignment

**[15 points]** For this part of the assignment, you are required to write a MIPS assembly program to perform character manipulation on a given string input.

The following are the tasks to complete this part:

1. Accept a string input from the console. The input may contain any ASCII character.
2. You are required to convert all lower case characters into upper case characters.
3. Display the converted string on the console.

## Example

```
Please enter a string: Abc123xY.,;"sdfGH
New string: ABC123XY.,;"SDFGH
```

**[35 points]** You are required to use *recursion* to invert an input sentence, which will be delimited by the whitespace character.

The following are the tasks to complete this part:

1. Accept a sentence as an input from the console.
2. Use recursion to invert the sequence of the words in the sentence. The only delimiter to be considered is the whitespace character. All other delimiters should be excluded when splitting the sentence.
3. Display the inverted sentence on the console.

## Example

```
Please enter a whitespace delimited sentence: The quick brown fox
jumped over the lazy dog
The inverted sentence is: dog lazy the over jumped fox brown quick
The
```

## Documentation on MIPS Assembler and SPIM

This section explains the various directives of the MIPS assembler, as well as, the “OS-like” services

provided by the SPIM simulator to MIPS programs.

### MIPS Assembler Syntax

**Comments** in assembler files begin with a sharp-sign (#) (also known as the pound sign or hash). Everything from the sharp-sign to the end of the line is ignored.

**Identifiers** are a sequence of alphanumeric characters, under-bars (\_), and dots (.) that do not begin with a number. Opcodes for instructions are reserved words that are **not** valid identifiers. **Labels** are declared by putting them at the beginning of a line followed by a colon, for example:

```

        .data
item:   .word 1
        .text
        .globl    main      # main must be global
        la       $t0, item  # load address(item) to register
main:   lw       $s0, 0($t0) # $t0 == &item;

```

Strings are enclosed in double-quotes ("). Special characters in strings follow the C convention:

```

newline  \n
tab      \t
quote    \"

```

SPIM supports a subset of the assembler directives provided by the actual MIPS assembler:

#### **.align n**

Align the next datum on a  $2^n$  byte boundary. For example, **.align 2** aligns the next value on a word boundary. **.align 0** turns off automatic alignment of **.half**, **.word**, **.float**, and **.double** directives until the next **.data** or **.kdata** directive.

#### **.ascii str**

Store the string in memory, but do not null-terminate it.

#### **.asciiz str**

Store the string in memory and null-terminate it.

#### **.byte b1, ..., bn**

Store the  $n$  values in successive bytes of memory.

#### **.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*.

#### **.double d1, ..., dn**

Store the  $n$  floating-point double precision numbers in successive memory locations.

#### **.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**.

**.float f1, ..., fn**

Store the  $n$  floating-point single precision numbers in successive memory locations.

**.globl sym**

Declare that symbol **sym** is global and can be referenced from other files.

**.half h1, ..., hn**

Store the  $n$  16-bit quantities in successive memory halfwords.

**.kdata <addr>**

The following data items should be stored in the kernel data segment. If the optional argument *addr* is present, the items are stored beginning at address *addr*.

**.ktext <addr>**

The next items are put in the kernel 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*.

**.space n**

Allocate  $n$  bytes of space in the current segment (which must be the data segment in SPIM).

**.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*.

**.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**).

**System Calls**

SPIM provides a small set of operating-system-like services through the MIPS system call (**syscall**) instruction. To request a service, a program loads the system call code (see Table 1) into register **\$v0** and the arguments into registers **\$a0**, ..., **\$a3** (or **\$f12** for floating point values). System calls that return values put their result in register **\$v0** (or **\$f0** for floating point results). For example, to print “**the answer = 5**”, use the commands:

```

        .data
str:    .asciiz "the answer = "
        .text
        li    $v0, 4      # $system call code for print_str
        la    $a0, str    # $address of string to print
        syscall          # print the string
        li    $v0, 1      # $system call code for print_int
        li    $a0, 5      # $integer to print
        syscall          # print it

```

**Table 1: System Services**

--	--	--	--

Service	System Call Code	Arguments	Result
<code>print_int</code>	1	<code>\$a0 = integer</code>	
<code>print_float</code>	2	<code>\$f12 = float</code>	
<code>print_double</code>	3	<code>\$f12 = double</code>	
<code>print_string</code>	4	<code>\$a0 = string</code>	
<code>read_int</code>	5		integer (in <code>\$v0</code> )
<code>read_float</code>	6		float (in <code>\$f0</code> )
<code>read_double</code>	7		double (in <code>\$f0</code> )
<code>read_string</code>	8	<code>\$a0 = buffer, \$a1 = length</code>	
<code>sbrk</code>	9	<code>\$a0 = amount</code>	address (in <code>\$v0</code> )
<code>exit</code>	10		
<code>print_character</code>	11	<code>\$a0 = integer</code>	
<code>read_character</code>	12		char (in <code>\$v0</code> )

`print_int` is passed an integer and prints it on the console.

`print_float` prints a single floating point number.

`print_double` prints a double precision number.

`print_string` is passed a pointer to a null-terminated string, which it writes to the console.

`read_int`, `read_float`, and `read_double` read an entire line of input up to and including the newline. Characters following the number are ignored.

`read_string` has the same semantics as the UNIX library routine `fgets`. It reads up to  $n - 1$  characters into a buffer and terminates the string with a null byte. If there are fewer characters on the current line, it reads through the newline and again null-terminates the string.

`sbrk` returns a pointer to a block of memory containing  $n$  additional bytes.

`exit` stops a program that is running.

`print_character` is passed an integer (character) and prints it on the console.

`read_character` has the same semantics as the UNIX library routine `fgetc`. It reads one character into an integer.

## Dishonesty

Make sure that you complete the assignment by yourself. Do not copy the code from others, nor provide others with your code. Do not copy and modify the code from other sources.