Assembly Programming Coursework

Deadline: 17th December 2019 11:59pm

1. Synopsis

This coursework is about MIPS implementation and testing. You will implement and test your code which answers the two questions below, before submitting them in the Moodle page. You should develop and test your code ONLY using the MARS simulator (v4.5).

2. Deliverable

- **Submit your UNCOMPRESSED assembly program files to Moodle.**
- **We will evaluate your submissions using MARS as well. Should you develop using other simulators, inability of compiling/running your code on MARS on our machine could cause zero marks.**
- **No report needs to be written.**

3. Plagiarism

You are gently reminded that we are at liberty to use plagiarism detection tools on your submission. Plagiarism will not be tolerated, and academic offenses will be dealt with in accordance with UNNC policy and as detailed in the student handbook. This means you may informally discuss the coursework with other students but you must implement your own programs (absolutely *do not* copy and paste from others) and provide your own answers where appropriate.

4. Assessment

- **This coursework consists of a total of 50 marks, which constitutes 50% of the module weight. Individual marks are shown next to each question.**
- **Note that lacking proper comments and user prompts will lose mark.**

Question 1

Implement the following C function using MIPS, which calculates the Fibonacci number in a recursive way. Please note that proper testing is needed before you submit your code. We will possibly test your code by entering strange inputs. The C code below does not handle the "strange inputs" as mentioned.

[**20 Marks**, Basic Functionality: 60%, Documentation: 20%, Input Test: 20%]

```
int fib (int n) {
      if (n == 0) {
            return 0;
      } else if (n == 1) {
            return 1;
      }
      return fib(n-1) + fib(n-2);
}
```

Question 2 [30 Marks]

The `syscall` routines only provide simplified functionalities, such as printing a single number, string, etc. Your code would be messed up quickly in more complex scenarios such as printing a series of integers, floats, strings, etc., and their combinations. Printing with formatting is even worse.

In this problem, you are required to develop/complete a `printf` procedure, which behaves like a simplified `printf` function in the standard C library. It implements only a subset of the functionality of the `printf` in C. Several sample syntax and functionality of the standard C `printf` are listed below.

```
/***********************************/
int i = 10;
char ch = 'A';
char *str = "This is a string message.";

// to print an integer:
printf ("%d", i);
// to print a character:
printf ("%c", ch);
// to print a string:
```

```
printf ("%s", str);
// to print a percentage symbol '%'
printf ("%%");

/***********************************/
```

In the above examples, in the parenthesis following the `printf`, the first element is a string, named format, which indicates the data type and format to be printed (E.g., "`%d`" tells `printf` to print an integer). The second element is a variable that contains the data to be printed, according to the format.

The `printf` procedure to be developed in this question should intake a comprehensive format string, which contains ***at most*** three (3) embedded formats as described above. Their corresponding variables should be stored in `$a1~$a3` before calling the procedure. If there are more than 3 embedded formats, all but the first 3 are completely ignored (not even printed).

A code skeleton is provided for you below. Answer the following questions.

QA: Complete the "printf.asm" by implementing the print string, print ASCII char, and print '%' functionalities.
[10 Marks. Basic Functionality: 80%, Documentation: 20%]

QB: Complete the printf.asm by implementing the testing section in the code. Specifically, you code should test all possible normal input cases of the `printf` procedure, and all possible abnormal input cases. So design your test scheme carefully. If an abnormal case is not handled in the code, feel free to modify the `printf` procedure.
[10 Marks. Ten (10) normal cases: 50%, Ten (10) abnormal cases: 50%]

QC: Add a "`%S`" format in the `printf` procedure, which is similar to the "`%s`" format, except that all lower case alphabets are converted into upper case ones in the output.
[10 Marks. Basic Functionality: 80%, Documentation: 20%]

```
## printf.asm
##
## Register Usage:
## $a0,$s0 - pointer to format string
## $a1,$s1 - format argument 1 (optional)
## $a2,$s2 - format argument 2 (optional)
## $a3,$s3 - format argument 3 (optional)
## $s4 - count of formats processed.
## $s5 - char at $s4.
## $s6 - pointer to printf buffer
##
## Source Courtesy D. J. E.

      .text
      .globl main

main:

############################################################
##    Your test code starts here.                         ##
##    You may add test data in the .data segment.         ##
############################################################

printf:
      subu  $sp, $sp, 36      # set up the stack frame
      sw    $ra, 32($sp)      # save local variables
      sw    $fp, 28($sp)
      sw    $s0, 24($sp)
      sw    $s1, 20($sp)
      sw    $s2, 16($sp)
      sw    $s3, 12($sp)
      sw    $s4, 8($sp)
      sw    $s5, 4($sp)
      sw    $s6, 0($sp)
      addu  $fp, $sp, 36

                              # grab the arguments
      move  $s0, $a0          # fmt string
      move  $s1, $a1          # arg1, optional
      move  $s2, $a2          # arg2, optional
      move  $s3, $a3          # arg3, optional

      li    $s4, 0            # set # of fmt = 0
      la    $s6, printf_buf   # set s6 = base of buffer

printf_loop:            # process each character at fmt
      lb    $s5, 0($s0)       # get the next character
      addu  $s0, $s0, 1       # $s0 pointer increases
      beq   $s5, '%', printf_fmt
      beq   $0, $s5, printf_end     # if zero, finish

printf_putc:
      sb    $s5, 0($s6) # otherwise, put this char
      sb    $0, 1($s6)  # into the print buffer
```

```
      move  $a0, $s6    # and print it using syscall
      li    $v0, 4
      syscall
      j     printf_loop

printf_fmt:
      lb    $s5, 0($s0) # get the char after '%'
      addu  $s0, $s0, 1
      # check if already processed 3 args.
      beq   $s4, 3, printf_loop


      # if 'd', then print as a decimal integer
      beq   $s5, 'd', printf_int
      # if 's', then print as a string
      beq   $s5, 's', printf_str
      # if 'c', then print as an ASCII char
      beq   $s5, 'c', printf_char
      # if '%', then print as a '%'
      beq   $s5, '%', printf_perc
      j     printf_loop

printf_shift_args:
      move  $s1, $s2
      move  $s2, $s3
      addi  $s4, $s4, 1 # increment no. of args processed
      j     printf_loop

printf_int:             # printf('%d', 100)
      move  $a0, $s1    # print the value stored in $s1
      li    $v0, 1
      syscall
      j     printf_shift_args

#################################################################
##    You may add code to process string, char, "%" here.  ##
#################################################################


printf_end:
      lw    $ra, 32($sp)
      lw    $fp, 28($sp)
      lw    $s0, 24($sp)
      lw    $s1, 20($sp)
      lw    $s2, 16($sp)
      lw    $s3, 12($sp)
      lw    $s4, 8($sp)
      lw    $s5, 4($sp)
      lw    $s6, 0($sp)
      addu  $sp, $sp, 36
      jr    $ra


exit:
```

```
        li    $v0, 10
        syscall



#################################################################
##    You may add whatever necessary in the .data segment.  ##
#################################################################
            .data
printf_buf: .space 2


## end of printf.asm ##
```