Assignment #1: MIPS Assembly

CSI3102-02 (Architecture of Computers), Spring 2023

Welcome to the very first programming assignment for CSI3102-02 @ Yonsei University for Spring 2023! You are given **about 2.5 weeks** to complete this assignment. Please carefully read the instructions below, and make sure you upload your assignment to the LearnUs system until the designated deadline.

In this programming assignment, you will write various functions using the MIPS assembly language (in particular, MIPS R2000 assembly language). The purposes of this assignment are (1) to make you get familiar with the MIPS assembly language, and (2) to give you hands-on experience on SPIM, a MIPS processor simulator designed to run the MIPS assembly code. Note that this assignment deals with the MIPS assembly language, not the MIPS machine language which consists of 0s and 1s.

Before working on the assignment, we strongly recommend you to study the textbook's Chapter A.10 for more details on the assembly language and the SPIM simulator.

Preparing a MIPS Simulation Environment

1-a. Intel/AMD CPU Users:

First, install VMware Workstation Player by visiting this site. Then, download a compressed image from https://drive.google.com/file/d/1f0PIurypF6gGC449AUGHx2wwby5ta1MI/view?usp=share_link. After that, decompress the image and import the image to VMware Workstation Player.

1-b. Apple Silicon Users:

Apple Silicon implements ARM64 ISA, not x86-64 ISA, so you should use a different virtual machine image. First, install UTM, an alternative to VMware Workstation Player, by referring to this site. Then, get the image from https://drive.google.com/file/d/1XLgQMF0EW02v8k7sVOOYC44HfHzPI2D7/view?usp=share_link. After that, decompress the image and import it to UTM.

2. Both Intel/AMD CPU and Apple Silicon Users:

Boot the image and you will encounter a login screen. Both the ID and password are `csi3102`. Log into the account by typing in the password. You will use the VMware image throughout this semester, so make sure you properly set up the VMware image and can successfully login.

To make sure that you have a proper version of the SPIM simulator, launch a terminal after logging in. Then, type `which spim` in the terminal and check if the output matches follows:

```
csi3102@csi3102:~$ which spim
/usr/bin/spim
```

Now, try executing the SPIM simulator by typing `spim`. You should see the following output:

```
csi3102@csi3102:~$ spim
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
```

```
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
(spim)
```

Make sure that you are using SPIM version 8.0. To inspect many useful built-in commands provided by SPIM, type `help` followed by the enter key in SPIM's command-line. Type `quit` and press the enter key to exit SPIM's command-line environment.

Downloading the Assignment

Please download this assignment and perform a sanity check by typing the following commands:

```
csi3102@csi3102:~$ wget
https://hpcp.yonsei.ac.kr/files/csi3102/2023sp/assn1.tar.gz
csi3102@csi3102:~$ tar zvxf ./assn1.tar.gz
csi3102@csi3102:~$ cd ./assn1
csi3102@csi3102:~/assn1$ ls
examples  Q1_Fibonacci.asm  Q2_FindIndex.asm  Q3_LCM.asm  Q4_QuickSort.asm
```

Example MIPS Assembly Code

As a small guide on how to write MIPS assembly code for the SPIM simulator, we provide four example code snippets under the `examples/` directory. The examples are named `0_helloWorld.asm`, `1_addTwoInts.asm`, `2_add1To10.asm`, and `3_updateMemory.asm`. The four code snippets demonstrate four different code patterns.

0_helloWorld.asm

This code demonstrates a simple "Hello World!" example. The code consists of a single function named `main`. The `main` label is the start point of any SPIM simulator invocation; the SPIM simulator always jumps to the `main` label after it sets up some preparation assembly code (e.g., system calls).

Try executing the file with `spim -file ./examples/0_helloWorld.asm`. You should see the following output:

```
csi3102@csi3102:~/assn1$ spim -file ./examples/0_helloWorld.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Hello world
```

When you invoke this file with the above command, the SPIM simulator does the following operations:

1) Load the file named `./examples/0_helloWorld.asm` which contains MIPS assembly code
2) Initialize a MIPS simulation environment
3) Invoke the function named `main` by executing a `jal` instruction with `main` as the target label
4) Print a string "`Hello world\n`" to the terminal
5) Function `main` returns control by invoking `jr $ra` instruction.

6) The simulation gets terminated.

The code consists of two `.`-prefixed regions: `.data` and `.text`. `.data` and `.text` correspond to the data segment and the text segment of MIPS ISA, respectively. You can think of the two segments as: `.data` is where the heap and stack get located at, and `.text` is where the assembly instructions should be.

You will notice `greet: .asciiz "Hello world\n"` under the data segment. `.asciiz` is an assembler directive for storing a null-terminated string in memory, and `"Hello world\n"` is the string being stored in the memory. Once the string is stored in the memory, we need to know the starting memory address of the string. This is done by using identifiers; the starting memory address is associated with identifier `greet`. The identifiers can later be used by MIPS assembly instructions to access the string. More details on assembler directives can be found in the textbook's chapter A.10.

Now, let's look at the instructions placed at the text segment. You will see that there are three instructions (i.e., `li`, `la`, and `syscall`) followed by `jr $ra` under the `main` label. The first three instructions are needed to invoke a system call which prints a string stored in memory to the terminal. The system call we are currently interested in is `print_string` (see Figure A.9.1 in the textbook), and invoking `print_string` requires `$v0` to have 4, the system call code for `print_str`, and `$a0` to contain the starting memory address of the string to be printed. The `li` and `la` instructions are used to prepare `$v0` and `$a0`. `li` is a pseudoinstruction (i.e., an instruction not defined by the machine language, but defined by the assembly language for easier programming) which loads an immediate into a register (see page A-57 of the textbook). `la` is also a pseudo-instruction which loads a computed address, not the contents, of a specified memory location into a destination register (see page A-66 in the textbook). In our example, the memory location is `greet`, the identifier for the string to be printed to the terminal, and the destination register is `$a0`. The code then invokes `print_string` system call by executing syscall instruction. After printing the string to the terminal output, the main function returns by executing the `jr $ra` instruction. This is a classic function-terminating instruction pattern in the MIPS ISA.

1_addTwoInts.asm

This example demonstrates how to perform an arithmetic operation on temporary registers (e.g., `$t0`, `$t1`, `$t2`) and utilize other system calls (e.g., `print_int`). In particular, the example performs `$t2` = `$t0` + `$t1` and prints the values of `$t0`, `$t1`, and `$t2` to the terminal.

```
csi3102@csi3102:~/assn1$ spim -file ./examples/1_addTwoInts.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
value1 ($t0): 3
value2 ($t1): 1
sum ($t2): 4
```

2_add1To10.asm

This example demonstrates a loop which adds integers from one to ten and prints the result to the terminal. To implement the loop, note that there are two labels in the code: `main` and `loop`. `$t0` stores the sum and `$t1` serves as the iterator for the loop.

```
csi3102@csi3102:~/assn1$ spim -file ./examples/2_add1To10.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
55
```

3_printMemory.asm

This example demonstrates more complex scenarios: (1) invoking a callee function `print_array` from the caller function `main`, (2) loading and printing an array of integers stored in the memory, (3) spilling `$ra` to `$s0` before calling `print_array` and recovering `$ra` after the callee function completes.

```
csi3102@csi3102:~/assn1$ spim -file ./examples/3_printMemory.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
1
2
3
4
5
6
7
8
```

First, invoking a callee function demands a use of `jal` instruction. You can see this behavior through `jal print_array` instruction under the main label. Note that `$ra` gets spilled to a callee-saved register `$s0` before executing the `jal` instruction as the `jal` instruction overwrites `$ra` upon execution. Second, under the `print_array_loop` label, the array's elements are accessed with `lw $a0, ($t1)` instruction. This instruction shows the register-relative addressing mechanism of the MIPS ISA; the instruction performs `$a0 = memory[$t1]`.

Problems

By analyzing the examples, we believe you will get a sense of how to write MIPS assembly instructions and execute the assembly instructions on the SPIM simulator. Now, it is time to write some MIPS assembly code! Please carefully read the instructions below. There are four problems.

Q1: Fibonacci Numbers (15 Points)

Implement a function `fibonacci` which returns the n-th Fibonacci number. The n-th Fibonacci number F(n) is defined as F(n) = F(n - 1) + F(n - 2) where F(0) = F(1) = 1.

You may assume that n is smaller than or equal to 32; however, a negative value can be given as n. If the given n is a negative number, your fibonacci function should indicate the negative number by returning with `$v0` = 0. Otherwise, your fibonacci function should return with `$v0` = 1 and `$v1` = F(n). You can see that, in real-world software, `$v0` corresponds to the error code returned by the function, and `$v1` represents a valid value if no error occurred.

Example #1: F(0) = 1

```
csi3102@csi3102:~/assn1$ spim -file ./Q1_Fibonacci.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Enter a positive integer: 0
INFO: fibonacci returned 1
```

Example #2: F(4) = 5

```
csi3102@csi3102:~/assn1$ spim -file ./Q1_Fibonacci.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Enter a positive integer: 4
INFO: fibonacci returned 5
```

Example #3: F(-4) = INVALID INPUT

```
csi3102@csi3102:~/assn1$ spim -file ./Q1_Fibonacci.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Enter a positive integer: -4
ERROR: received a negative integer!
```

Q2: Finding Index (20 Points)

Implement a function `findIndex` which finds the index of a character in a string. The comparison between a string `str` and a character `char`, is performed with respect to the following Python-based pseudocode:

```
def findIndex(str, char):
  ret = -1
  for i in range(len(str)):
    c0 = ord(str0[i])
    c1 = ord(char)
    if c0 == c1:
      ret = i
      break
  return ret
```

A string is stored in the memory, is null-terminated (i.e., the end-of-string is denoted by a zero-filled byte), and consists of ASCII characters. For example, if we store a null-terminated string “ABCDEF” in the memory with 0x1000 as the starting memory address, the byte-addressable memory contents are: memory[0x1000] = 65, memory[0x1001] = 66, memory[0x1002] = 67, memory[0x1003] = 68, memory[0x1004] = 69, memory[0x1005] = 70, and memory[0x1006] = 0.

The function `findIndex` takes as input the starting memory address of `str` through `$a0`, and the memory address of `char` through `$a1`. The length of `str is` not provided. Your implementation should automatically identify their lengths by detecting the null characters. The return value of the function, `ret`, should be stored in `$v0`.

You can test your implementation with different strings and characters by changing the values of `str` and `char` located at the top of the `Q2_FindIndex.asm` file. Here are some examples:

Example #1: str = “TestString0”, char = “n”

```
csi3102@csi3102:~/assn1$ spim -file ./Q2_FindIndex.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
INFO: findIndex returned: 8
```

Example #2: str = “TestString1”, char = “t”

```
csi3102@csi3102:~/assn1$ spim -file ./Q2_FindIndex.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
INFO: findIndex returned: 3
```

Example #3: str = “TestString”, char = “T”

```
csi3102@csi3102:~/assn1$ spim -file ./Q2_FindIndex.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
INFO: findIndex returned: 0
```

Example #4: str = “TestString”, char = “k”

```
csi3102@csi3102:~/assn1$ spim -file ./Q2_FindIndex.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
INFO: findIndex returned: -1
```

Example #5: str = “TestString0”, char = “0”

```
csi3102@csi3102:~/assn1$ spim -file ./Q2_FindIndex.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
```

```
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
INFO: findIndex returned: 10
```

Q3: Least Common Multiple (25 Points)

Implement a **recursive** function `gcd` which calculates the greatest common divisor (GCD) of two 32-bit unsigned integers. And implement the function `lcm` which calculates the least common multiple (LCM) using the function `gcd`.

The function `lcm` takes as input two 32-bit unsigned integers through `$a0` and `$a1`, and returns the LCM of the two integers by storing it in `$v0` before invoking its `jr` instruction. You should invoke the function `gcd` by using jump-and-link (`jal`) instruction. A Python-based pseudocode of the function is shown below.

```
def gcd(a0, a1):
  if a1 == 0:
    return a0
  else:
    return gcd(a1, a0 % a1)

def lcm(a0, a1):
  return (a0 * a1) / gcd(a0, a1)
```

You must implement the function `gcd` as a recursive function. That is, your implementation of the function should not be based on a loop, Euclidean algorithm, etc.

The two integers for the first invocation of the function are provided by the user through the command-line interface as shown in the following examples.

Example #1: LCM(12, 16) = 48

```
csi3102@csi3102:~/assn1$ spim -file ./Q3_LCM.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Enter a 32-bit unsigned integer: 12
Enter a 32-bit unsigned integer: 16
The LCM of the two integers is: 48
```

Example #2: LCM(12, 36) = 36

```
csi3102@csi3102:~/assn1$ spim -file ./Q3_LCM.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Enter a 32-bit unsigned integer: 12
Enter a 32-bit unsigned integer: 36
The LCM of the two integers is: 36
```

Example #3: LCM(12, 13) = 156

```
csi3102@csi3102:~/assn1$ spim -file ./Q3_LCM.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Enter a 32-bit unsigned integer: 12
Enter a 32-bit unsigned integer: 13
The LCM of the two integers is: 156
```

Q4: In-Place Quick Sort (40 Points)

Implement a **recursive** function `quickSort` which sorts an array of ASCII characters using **in-place** quick sort. Here is a C-based pseudocode for the function with a helper function named `partition`:

```
unsigned int partition(char *str, unsigned int left, unsigned int right) {
  unsigned int pivot = arr[right];
  unsigned int index = left - 1;

  for (unsigned int i = left; i < right - 1; i += 1) {
    if (arr[i] < pivot) {
      index += 1;
      swap str[index] and str[i]
    }
  }
  swap str[index + 1] and str[right]
  return (index + 1)
}

void quickSort(char *str, unsigned int left, unsigned int right) {
  if (left < right) {
    unsigned int pivot = partition(str, left, right);
    quickSort(str, left, pivot - 1);
    quickSort(str, pivot + 1, right);
  }
}
```

You should implement the two functions, `quickSort` and `partition`, for this problem.

The `quickSort` function takes as input three arguments: the starting memory address of a string through `$a0` (i.e., `str`), the first index of the range of the string to be sorted through `$a1` (i.e., `left`), and the last index of the range of the string to be sorted through `$a2` (i.e., `right`). For example, if the starting memory address of a string is 0x1000 and the range of the string to be sorted is [2, 15), `$a0` = 0x1000, `$a1` = 2, and `$a2` = 15 upon an invocation of the `quickSort` function.

The `partition` function takes as input three arguments: the starting memory address of a string through `$a0` (i.e., `str`), the first index of the range of the string to be sorted through `$a1` (i.e., `left`), and the last index of the range of the string to be sorted through `$a2` (i.e., `right`). The `partition` function should be invoked by the `quickSort` function for splitting smaller and larger numbers based on pivot.

You can modify the `str_buf` and `str_len` values located at the top of `Q4_QuickSort.asm` to test your implementation with various strings including the examples shown below.

Example #1: str_buf = “this is a string.”, str_len = 17

```
csi3102@csi3102:~/assn1$ spim -file ./Q4_QuickSort.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
The string to be sorted is: this is a string.
The sorted string is:    .aghiiinrssstt
```

Example #2: str_buf = “0xDEADBEEF”, str_len = 10

```
csi3102@csi3102:~/assn1$ spim -file ./Q4_QuickSort.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
The string to be sorted is: 0xDEADBEEF
The sorted string is: 0ABDDEEEFx
```

Example #2: str_buf = “abcdefgHIJKLMNOP”, str_len = 16

```
csi3102@csi3102:~/assn1$ spim -file ./Q4_QuickSort.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
The string to be sorted is: abcdefgHIJKLMNOP
The sorted string is: HIJKLMNOPabcdefg
```

Submitting Your Code

You should compress the files you modified and upload the compressed file to LearnUs. The deadline to submit your code is **11:59pm KST on Sunday, April 30th, 2023**. **No late submissions will be accepted**.

One way to create a compressed file is shown below.

```
csi3102@csi3102:~/assn1$ tar -czvf assn1-submission.tar.gz Q1_Fibonacci.asm
Q2_FindIndex.asm Q3_LCM.asm Q4_QuickSort.asm
```

After creating the compressed file (named `assn1-submission.tar.gz`), upload it to LearnUs.