PROGRAMING IN MIPS

The subdirectories in this folder are each independent exercises to help you practice programming in MIPS assembly language. Although the exercises are independent and have separate goals, if this is your first time working in assembly language we suggest tackling the exercises in this order:

1. prompt-and-print -- in this exercise you will:
    - Write a self contained "script", meaning you will not have to implement a "calling convention" because you are not writing a "function".
    - Use `syscall` to perform a prompt, and a print operation.
2. simple-loop -- in this exercise you will:
    - Write a self contained assembly "script", meaning you will not have to implement a "calling convention" because you are not writing a "function".
    - Allocate an array in the stack.
    - Wrestle with data types in assembly.
    - Use `syscall` to print data.
3. binary-conversion -- in this exercise you will:
    - Convert input "strings" to integers.
    - Implement a single function using a calling convention.
    - Use "bitwise" thinking to convert data from binary "strings" to binary "ints".
4. nesting-function-calls -- in this exercise you will:
    - Implement several calling conventions, and call functions from inside functions.
5. fibonacci -- in this execrcise you will:
    - Implement a recursive function in assembly.
    - Gain a new level of understanding about stacks, function calls, and recursion.

RUNNING THE CODE

In order to execute MIPS assembly code you'll need to download the MARS Emulator. The first two exercises are designed to be self contained "scripts" so to speak; you are asked to write some assembly code which, when run, does something specific. For these exercises simply assembly and execute a single `.asm` file that you've written and examine the output in MARS to determine the correctness of your function.

In the last 3 exercises you are asked to implement functions, and provided with a separate `.asm` file which contains test cases and some assembly code to call your functions with those test cases. In this case, when you want to run the test cases you should assembly and run the `runner.asm` file located in each subdirectory. If you examine those runners you will see that they use the `.include` directive to reference your functions.

For example:

```
.include "fibonacci.asm"
```

This looks for a file called fibonacci.asm in the same directory as the `runner.asm` file where the `.include` appears. Additionally, notice the `.global` directive in each of the `.asm` files where you've been asked to implement a function, such as:

```
.global fibonacci

fibonacci:
    li $v0, 1  # simplistic version that passes first two tests...
    jr $ra
```

This `.global` makes the label that appears below available in the scope of mips files which include this file. Removing the `.global` directive will break the test runners!

SOME TIPS:

The first two exercises ask you to `print` which can be done in the MARS emulator via the use of `syscall`. Here is a reference sheet and an example of using `syscall` in MIPS.

When working with assembly, you need to adopt a new perspective on working with data. In particular you will have to internalize the idea that values no longer have types from the machine's perspective. Registers contain some binary data, and the only thing which "knows" what "type" of data it represents is the programmer. If register `$t0` contains the ASCII values for `"abc"` (which is 4 bytes of data, a,b,c,NULL). The hex for this is `0x00636261`, which might be the integer `636261`. Knowing which interpretation is the correct interpretation is up to the programmer.

When you're confused about a value in a memory location or a register, try asking yourself:

- Is this a pointer or a value?
- If it's a value, what type is the value supposed to be?
- If it's a pointer, does it point to:
    - The stack?
    - The heap?
    - Something loaded from the `.data` section?
- Try alternating the MARS emulator to show values in integer, hex, and ascii -- some patterns will only become clear when you view the data with a specific type.

Walk through your code as it executes line by line. Try to predict which registers and memory locations will change as a result of each line you execute, then execute a line and see if you were right. If not, pause and try to determine which of your assumptions was off -- did you misunderstand the assembly command, the data type, or something else?

> Good luck!