**Our First Few Simple MIPS Assembly Programs**

**Exercise 0**

For this exercise, you will download the MARS assembler, and run a few simple MIPS assembly programs from the course website.

First, download MARS (link on course website). Launch it by double-clicking the .jar file. Familiarize yourself with the menu.

Configure it as follows:***Settings->Memory Configuration->Compact, Data at Address 0*.** This setting tells the assembler where it can expect to place data and code in memory.

Download three assembly files from the course website (under Lecture 7): [Sum.asm](http://www.cs.unc.edu/~montek/teaching/Comp411-Fall12/Home/Home_files/Sum.asm), [SumArray.asm](http://www.cs.unc.edu/~montek/teaching/Comp411-Fall12/Home/Home_files/SumArray.asm) and [Fibonacci1.asm](http://www.cs.unc.edu/~montek/teaching/Comp411-Fall12/Home/Home_files/Fibonacci1.asm) .

Use the ***File->Open* menu** to open one of these programs. Assemble (i.e., compile) the program by hitting ***Run->Assemble***, or by hitting the screwdriver/wrench icon, or by pressing **F3**. Run the program by hitting ***Run->Go***, or by hitting the icon with the play button. You may single-step through the program by hitting ***Run->Step***, or hitting the icon with the '1' to the right of the play button. You can also set one or more breakpoints by checking the box to the left of the instructions where you want execution to break, and then hit ***Run***.

The bottom pane has two tabs: *Mars Messages* shows errors or warnings during assembly; *Run I/O* is the input/output.

Run each of the three programs. Make sure you understand every single line of code. Here is what to expect for each:

* **Sum.asm**: Single-step through the program, and observe that the result (i.e., sum of numbers 0..4) will be in register $8 at the end of execution.
* **SumArray.asm**: Single-step through it, and observe that the result (sum of 7, 8, 9, 10 and 8) will be in the **sum** variable (at data address 0x0 in the memory).
* **Fibonacci1.asm**: Single-step through it. This program computes the largest Fibonacci number that is less than 100. Follow the logic of the program and see where the Fibonacci numbers are stored. See where the final answer (largest Fibonacci less than 100) is stored in data memory.

**MARS syscall**

There are several library routines provided by MARS that an assembly program can use. These are called *system calls*, or **syscall**. These services include support for printing integers and strings (similar to the **printf()** function in C), reading integers and strings from the keyboard (similar to **scanf()** in C), memory allocation (similar to **malloc()** in C), exiting from a program (similar to return from **main()** in C), etc.

An assembly program accesses those services using the **syscall** command. There is only one **syscall** command for all these services, but which service is requested is determined by the values provided in certain registers. The value in register **$v0** determines which service is requested, and often parameters are passed to the service using registers **$a0**, **$a1**, **$a2** and **$a3**. If a value needs to be returned to the program (e.g., reading an integer from keyboard), it is typically returned in register **$v0**.

For a full listing of system calls available in MARS, please refer to <http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html>. We will mostly be using system calls numbered 1 to 17.

For example, to exit a program, you would use **syscall** with 10 in **$v0**:

ori $v0, $0, 10 #System call code 10 for exit

syscall #exit the program

Note: Sometimes the instruction ori $v0, $0, 10 is shortened to the pseudo instruction li $v0, 10, which stands for ("load the immediate value 10 into $v0"). As another example, to print a string located at location **myString**, you would use **syscall** with 4 in **$v0**:

li $v0, 4 #System call code 4 for printing a string

ori $a0, $0, myString #address of string to print is in $a0

syscall #print the string

Study all of the system calls from 1 to 17.

**Exercise 1: Familiarizing yourself with MARS**

For this exercise, we calculate the Fibonacci numbers using**fib[0] = 0; fib[1] = 1; fib[n] = fib[n-1] + fib[n-2].**

**Getting started:**

**Run MARS.**

Load [lab3\_ex1.s](http://inst.eecs.berkeley.edu/~cs61c/fa16/labs/3/lab3_ex1.s) using **File-->Open**.

View and edit your code in the "Edit" tab. Notice the code highlighting and 'completion suggestion' features.

When ready, assemble your code using **Run-->Assemble (or press F3).**

This will take you automatically to the "Execute" tab, which is where you can run and debug your program.

Step through the program using **Run-->Step (or press F7)**.

You should take the time to familiarize yourself with everything in the Run menu (and the keyboard shortcuts).

使用Help(问号图标)回答下列关于MARS的问题.

.data, .word, .text 指示器（directives）的含义是什么(即, 在每段中放入什么内容)?

在MARS中如何设置断点breakpoint? 请在第15行设置断点，并在所有问题解答完后，将此结果给老师检查。

在程序运行到断点处停止时，如何继续执行? 如何单步调试代码?

如何知道某个寄存器register的值是多少? 如何修改寄存器的值.

**n** 存储在内存中的哪个地址? 通过修改此内存处的值来计算第13个fib数.

16 和 18 行使用了**syscall**指令. 其功能是什么，如何使用它? (提示: syscall 在Help中有说明!如何英文不是太好，可以一边运行，一边看效果，来体会其用途),把答案给老师看.

**Exercise 2: Compiling from C to MIPS**

Compile The file [lab3\_ex2.c](http://inst.eecs.berkeley.edu/~cs61c/fa16/labs/3/lab3_ex2.c) into MIPS code.

|  |  |
| --- | --- |
| [lab3\_ex2.c](http://inst.eecs.berkeley.edu/~cs61c/fa16/labs/3/lab3_ex2.c)  int source[] = {3, 1, 4, 1, 5, 9, 0};  int dest[10];  int main ( ) {  int k;  for (k=0; source[k]!=0; k++) {  dest[k] = source[k];  }  return 0;  } | **Reference:**  .globl source  .data  source:  .word 0x3 , 0x1 , 0x4 ,0x1,0x5,0x9,0x0  dest: .space 10  .globl main  .text  main: addiu $sp,$sp,-16  sw $30,0($sp)  move $30,$0  b L\_5  L\_2: sll $24,$30,2  lw $15,source($24)  sw $15,dest($24)  L\_3: la $30,1($30)  L\_5: sll $24,$30,2  lw $24,source($24)  bne $24,$0,L\_2  move $2,$0  L\_1: lw $30,0($sp)  addiu $sp,$sp,16  jr $31 |

Find the section of code in lab3\_ex2.s that corresponds to the copying loop and explain how each line is used in manipulating the pointer.

**Exercise 3**

This exercise uses the file [listmanips.s](http://inst.eecs.berkeley.edu/~cs61c/fa16/labs/3/listmanips.s).

In this exercise, you will complete an implementation of map in MIPS. Our function will be simplified to mutate the list in-place, rather than creating and returning a new list with the modified values. Our map procedure will take two parameters; the first parameter will be the address of the head node of a singly-linked list whose values are 32-bit integers. So, in C, the structure would be defined as:

struct node {

int value;

struct node \*next;

};

Our second parameter will be the address of a function that takes one int as an argument and returns an int. We'll use jalr (see below) to call this function on the list node values.

Our map function will recursively go down the list, applying the function to each value of the list nodes, storing the value returned in that node. In C, this would be something like this:

void map(struct node \*head, int (\*f)(int))

{

if(!head) { return; }

head->value = f(head->value);

map(head->next,f);

}

If you haven't seen the int (\*f)(int) kind of declaration before, don't worry too much about it. Basically it means that f is a pointer to a function, which in C is used exactly like any other function.

You'll need to use an instruction you might not have learned before to implement this: jalr. jalr is to jr as jal is to j. It jumps to the address in the given register and stores the address of the next instruction (i.e., PC+4) in $ra. So, if I didn't want to use jal, I could use jalrto call a function like this:

# I want to call the function garply, but not use jal.

la $t0 garply # so I use la to load the address of garply into a register ($t0)

jalr $t0 # and then use jalr to jump and link to it.

There are 7 places (6 in map and 1 in main) in the provided code where it says "YOUR\_INSTRUCTION\_HERE". Replace these with instructions that perform as indicated in the comments to finish the implementation of map, and to provide a sample call to mapwith square as the function argument. When you've filled in these instructions, running the code should provide you with the following output:

List Before: 9 8 7 6 5 4 3 2 1 0

List After: 81 64 49 36 25 16 9 4 1 0

#### Checkoff

* Show your TA your test run.

**Exercise 4**

Add the prologue and epilogue to the code in [nchoosek.s](http://inst.eecs.berkeley.edu/~cs61c/fa16/labs/3/nchoosek.s) so that it computes "n choose k". the number of combinations of n distinct elements taken k at a time. This can be computed through factorials, however we will be computing this through finding the (n,k) entry in [Pascal's triangle](https://en.wikipedia.org/wiki/Pascal's_triangle).

Checkoff

* Show your TA your code and its test run.