1. Reading and Printing Integers

For our next example, we'll write a program named add2.asm that computes the sum of two numbers specified by the user at runtime, and displays the result on the screen.

The algorithm this program will follow is:
1. Read the two numbers from the user - We'll need two registers to hold these two numbers. We can use $t0 and $t1 for this.

2. Compute their sum- We'll need a register to hold the result of this addition. We can use $t2 for this.
3. Print the sum.
4. Exit. We already know how to do this, using syscall.

The only parts of the algorithm that we don't know how to do yet are to read the numbers from the user, and print out the sum. Luckily, both of these operations can be done with a syscall. Looking again in table from lecture 7, we see that syscall 5 can be used to read an integer into register $v0, and and syscall 1 can be used to print out the integer stored in $a0.

The syscall to read an integer leaves the result in register $v0, however, which is a small problem, since we want to put the first number into $t0 and the second into $t1. We find the move instruction, which copies the contents of one register into another. Note that there are good reasons why we need to get the numbers out of $v0 and move them into other registers: first, since we need to read in two integers, we'll need to make a copy of the first number so that when we read in the second number, the first isn't lost. In addition, when reading through the register use guidelines, we see that register $v0 is not a recommended place to keep anything, so we know that we shouldn't leave the second number in $v0 either.

This gives the following program:

```
# Daniel J. Ellard -- 02/21/94
# add2.asm-- A program that computes and prints the sum
# of two numbers specified at runtime by the user.
# Registers used:
# $t0 - used to hold the first number.
# $t1 - used to hold the second number.
# $t2 - used to hold the sum of the $t1 and $t2.
# $v0 - syscall parameter and return value.
# $a0 - syscall parameter.

main:
## Get first number from user, put into $t0.
li $v0, 5                       # load syscall read_int into $v0.
syscall                         # make the syscall.
move $t0, $v0                   # move the number read into $t0.

## Get second number from user, put into $t1.
li $v0, 5                   # load syscall read_int into $v0.
```

```
syscall                 # make the syscall.
move $t1, $v0           # move the number read into $t1.
add $t2, $t0, $t1       # compute the sum.

## Print out $t2.
move $a0, $t2           # move the number to print into $a0.
li $v0, 1               # load syscall print_int into $v0.
syscall                 # make the syscall.
li $v0, 10              # syscall code 10 is for exit.
syscall                 # make the syscall.
# end of add2.asm.
```

2. Strings: the hello Program

The next program that we will write is the \Hello World" program. Looking in table 4.6.1 once again, we note that there is a syscall to print out a string. All we need to do is to put the address of the string we want to print into register $a0, the constant 4 into $v0, and execute syscall. The only things that we don't know how to do are how to de_ne a string, and then how to determine its address. The string "Hello World" should not be part of the executable part of the program (which contains all of the instructions to execute), which is called the text segment of the program. Instead, the string should be part of the data used by the program, which is, by convention, stored in the data segment.

To put something in the data segment, all we need to do is to put a .data before we define it. Everything between a .data directive and the next .text directive (or the end of the file) is put into the data segment. Note that by default, the assembler starts in the text segment, which is why our earlier programs worked properly even though we didn't explicitly mention which segment to use. In general, however, it is a good idea to include segment directives in your code, and we will do so from this point on. We also need to know how to allocate space for and define a null-terminated string.

In the MIPS assembler, this can be done with the .asciiz (ASCII, zero terminated string) directive. For a string that is not null-terminated, the .ascii directive can be used

Therefore, the following program will fulfill our requirements:

```
# Daniel J. Ellard -- 02/21/94
# hello.asm-- A "Hello World" program.
# Registers used:
# $v0 - syscall parameter and return value.
# $a0 - syscall parameter-- the string to print.
.text
main:
la $a0, hello_msg       # load the addr of hello_msg into $a0.
li $v0, 4               # 4 is the print_string syscall.
syscall                 # do the syscall.
li $v0, 10              # 10 is the exit syscall.
syscall                 # do the syscall.
# Data for the program:
.data
```

```
hello_msg: .asciiz "Hello World\n"
# end hello.asm
```