Pre-Knowledge
In order to complete this lab you will need an understanding of the MIPS instruction set and some knowledge of assembly.

Objective
In this lab we learn to simulate a MIPS processor and write our own assembly code

Lab Execution : Week 1
Guidelines: For each lab there will be a report. This report will consist of a text that follows the report guideline, your code file, and screenshots of your working code. These files should be placed together in a single file, printed, and then turned into your TA. Your lab report is due the week after you finish lab. The lab be composed of a report worth 70 points (20 from the report and 50 for the vhdl files) and possibly a pre-lab worth 30 points. On labs that take place over multiple weeks there will be only a single lab report due the week after the lab is completed. Lab reports suffer a 10% per day overall penalty for late work.

Tools: In this lab we will be using a MIPS simulation tool called SPIM. Though it should be present on your computers you can obtain a free copy for home use.

- SPIM can be downloaded from this link: https://sourceforge.net/projects/spimsimulator/files/
  - Choose the QTSpim option for whatever operating system you may be running

ASM Programming Structure: Assembly code conforms to the following structure:

```
.data  # variable declarations follow this line
      # Format would be like “label:”
      # storage_type value(s)

.text  # instructions follow this line
main:  # indicates start of code (first instruction to execute)
```

The data portion of the code (which we will not be using) is where variables and other values can be stored. The text portion is where the machine instructions go. Functions in the text portions have labels so that a user can easily jump or branch to that instruction line. It is also worth noting that the # symbol causes any text after that symbol to be commented out.
**MIPS Register Set:** A table describing each of the registers available in the MIPS processor is as follows:

<table>
<thead>
<tr>
<th>Register Number</th>
<th>Alternative Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero</td>
<td>the value 0</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>(assembler temporary) reserved by the assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>$v0 - $v1</td>
<td>(values) from expression evaluation and function results</td>
</tr>
<tr>
<td>4-7</td>
<td>$a0 - $a3</td>
<td>(arguments) First four parameters for subroutine. Not preserved across procedure calls</td>
</tr>
<tr>
<td>8-15</td>
<td>$t0 - $t7</td>
<td>(temporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls</td>
</tr>
<tr>
<td>16-23</td>
<td>$s0 - $s7</td>
<td>(saved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls</td>
</tr>
<tr>
<td>24-25</td>
<td>$t8 - $t9</td>
<td>(temporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.</td>
</tr>
<tr>
<td>26-27</td>
<td>$k0 - $k1</td>
<td>reserved for use by the interrupt/trap handler</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>global pointer. Points to the middle of the 64K block of memory in the static data segment.</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>stack pointer. Points to last location on the stack.</td>
</tr>
<tr>
<td>30</td>
<td>$s8/$fp</td>
<td>saved value / frame pointer. Preserved across procedure calls</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>return address</td>
</tr>
</tbody>
</table>

The registers that we will be using are the values, arguments, and temporaries registers. These registers can be referenced either by their number or letter. So, register 2 would be $2 or $v0.

- The 0 register, which is always 0.
- The values registers are used for storing returned values from a function as well as selecting which system call to use.
- The arguments registers are used to pass in arguments to a function that the system is calling.
- The Temporaries registers are used to store temporary data for calculations
  - These registers are Caller Saved, this means that if one function wishes to call another function and guarantee that the values in these registers are untouched, that function must save these to memory. Thus the function caller must save the values.
  - This is opposed to Callee Saved registers which, if used, must first be saved by the calling function. Thus the Callee must save the values.
ASM Programming instructions: For this lab your goal is to run some example assembly code (provided below), and then modify the code to read in and add 3 numbers instead of just 2.

1. .data  # variable declarations follow this line
2. .text  # instructions follow this line

3. main:
4. ## Code Part 1: Get first number from user, put into $t0.
5. ori $v0, $0, 5 # OUR CODE BEGINS HERE: load syscall read_int into $v0.
6. syscall # make the syscall.
7. addu $t0, $0, $v0 # move the number read into $t0.

8. ## Get second number from user, put into $t1.
9. ori $v0, $0, 5 # load syscall read_int into $v0.
10. syscall # make the syscall.
11. addu $t1, $0, $v0 # move the number read into $t1.

12. add $t2, $t0, $t1 # compute the sum.

13. ## Print out $t2.
14. addu $a0, $0, $t2 # move the number to print into $a0.
15. ori $v0, $0, 1 # load syscall print_int into $v0.
16. syscall # make the syscall.

17. ori $v0, $0, 10 # syscall code 10 is for exit.
18. syscall # make the syscall.

19. ## end of add2.asm.

A discussion of each line of the code is as follows. The first two lines declare the .data and the .text segments of our code. There is nothing after the .data segment because we have no variables to store. All of our instructions follow the .text segment.

Next come the declaration of the main: label in line 3 which is required for any code we wish to run. Lines 4-7 read in and store a value. To read a value we must make a system call (or syscall), and tell our processor we need input. Each syscall (there are many types) has an associated number. In order to make a syscall we put the call’s associated number in $v0 and then execute the syscall instruction. In our case the associated number for reading in a numerical value is 5. Once a syscall finishes its return values are stored in $v0 and/or $v1.

Lines 8-11 repeat the reading process but store the value in a different register. Line 12 calculates the sum of the two values we read in.

Lines 13-16 handle printing out the computed sum. In this case we are using the print_int syscall whose number is 1. This syscall prints out whatever value is stored in the argument register $a0.

Finally we must exit the main method. Lines 17 and 18 accomplish this by using the exit syscall.
Week 2

**ASM Programming instructions:** For this week the student’s goal is to translate the following C code into assembly. Your goal is to have a program that generates the same output as this one does, using the same concepts (loops, branches, etc).

```c
main()
{
    Cout << "Numbers:" << endl;

    int x[] = {18, 12, 6, 500, 54, 3, 2, 122};

    int i;

    for (i=0; i<8; i++)
    {
        Cout << x[i];
    }

    exit(0);
}
```

**ASM Tutorial Examples:**

1. Printing an integer (we did this last time):
   a. `addu $a0, $0, $t_`  # Store the integer to be printed in register $A0
   b. `ori $v0, $0, 1`  # Store the syscall to be executed in register $V0
   c. `syscall` # run the system call

2. Printing a string
   a. `la $a0, str1`  #Load Address of str1 (defined in the .data section) to $a0
   b. `ori $v0, $0, 4`  # Store the syscall to be executed in register $V0
   c. `syscall` # run the system call

3. Defining a string in .data
   a. `str1: .ascii "Demo to print string"`

4. Branching (testing a condition) Pg : 105 in the textbook
   a. “if ( i < 8)”  
      [ `slt $t0, $t1, 8` ] # sets $t0 to 1 if $t1 is less than 8
   b. [ `bne $t0, 1 branch 2` ] # if $t0 is not equal to 1 then branch
   c. `branch 1 :` [ `nop` ] # the instruction after a branch is executed even if we do
      # not branch so we should put a nop here
   d. [ `branch 1 code here` ] # must end with a jump to the After label so that
      # we do not execute the branch 2 code
   e. “else”
   f. `branch 2 :` [ `branch 2 code here` ]
   g. After: [code after the condition test]
5. Making a loop: Pg 107 in the textbook
   a. Example: counting from 0 to 6
   b. # init $s0 to 0. ($s0 just like i in for loop)
   c. addi $s0, $s0, 0
   d. Loop:
   e. # Compare $s0 with 7. If $s0 < 7 then $t0=1
   f. slt $t0, $s0, 7
   g. # if $t0 not equal to 1 branch out of the loop ($t0 acts like a Boolean value)
   h. bne $t0, 1, AfterLoop
   i. nop # you have to have a nop after branch instructions
   j. addi $s0, $s0, 1
   k. j Loop
   l. AfterLoop:

6. Reading from or writing to memory
   a. .data
   b. array1: .space 12 # declare 12 bytes of storage to hold array of 3 integers
   c. .text
   d.
   e. main: la $t0, array1 # load base address of array into register $t0
   f. li $t1, 5 # $t1 = 5 ("load immediate")
   g. sw $t1, ($t0) # first array element set to 5; indirect addressing
   h. li $t1, 13 # $t1 = 13
   i. sw $t1, 4($t0) # second array element set to 13 (we use the address of
   # the start of the array + 4, as integers are 4 bytes long)
   j. lw $t2, 4($t0) # Read in the value of the second array element and
   # store that value in $t2
   k. addi $t0, $t0, 4 # another way to read the next element, add 4 to $t0
   l. lw $t3, ($t0) # another way to get the second array element and
   # store that value in $t2. (note how no offset is needed)

NOTE: IF USING THE .space DECLARATION TO LOAD AND STORE INTEGERS (INSTEAD OF .word AS GIVEN BELOW). PUT THIS AS THE FIRST DECLARATION IN YOUR DATA SECTION. THIS IS BECAUSE .space DOES NOT AUTOMATICALLY START AT A WORD BOUNDARY. IF YOU HAVE OTHER DECLARATIONS BEFORE .space AND THEY DO NOT END ON A WORD BOUNDARY, lw INSTRUCTION WILL THROW AN EXCEPTION (AS IT WILL ONLY READ STARTING AT WORD BOUNDARIES).
ASM Code Shell: Though not required you can use the following code as a shell to form the basis of your program. Again, if you decide to use .space instead of .word make sure to pay attention to the note given above.

.data
   str1: .ascii "Numbers:"
   str2: .ascii "\n"
   x: .word 18,12,6,500,54,3,2,122 # Defines and stores the following 8 integers at the memory location defined at label x

.text # instructions follow this line
main:
   # Print out str1
   # define the variable i
   # start the Loop
   Loop:
      # Read the x[i]
      # Print out the x[i]
      # Print str2
      # Check the condition and make a branch

Exit:
   ori $v0, $0, 10
   syscall
Week 3

**ASM Programming instructions:** For this week the student’s goal is to write an occurrence finding function that builds off of the past two weeks of assembly programming. The requirements for this function are as follows.

1. The user is allowed to type in 10 positive integers plus an occurrence value (meaning 11 inputs).
   a. The 10 integers should be handled in a loop where each value input by the user (as in week 1) is stored into memory in an array like structure (as in week 2).
   b. The input occurrence is also an integer which represents a value present (or not) in the input array.
2. Then, the program will loop through the array and find any matching with the input occurrence value.
3. When finding a matching, its index inside the array will be output and the number of occurrences found will be incremented by 1.
4. At the end of the loop, the program will print out the number of occurrences found.

The C++ code for this program is as such:

```cpp
main()
{
    int x[10], occur, count = 0;

    COUT << "Type in array numbers:" << endl;
    for (int i=0; i<10; i++) // reading in integers
    {
        CIN >> x[i];
    }

    COUT << "Type in occurrence value:" << endl;
    CIN >> occur;

    // Finding and printing out occurrence indexes in the array
    COUT << "Occurrences indexes are:" << endl;
    for (int i=0; i<10; i++)
    {
        if (x[i] == occur)
        {
            cout << i;
            count ++;
        }
    }
    cout << "Number of occurrences found:" << endl;
}```
Cout << count;
exit(0);