Computer Organization and Design (ECE 219)
Winter Semester 2021-2022

Laboratory 1

Objectives of the workshop

- Input-output commands
- Using conditions in selection structures
- Using conditions in iteration structures
- Memory access commands
- Use of tables (numeric and alphanumeric)

Input-output commands

Commands for entering integers from the console:

```
li $v0, 5                 # code 5 from syscalls in $v0
syscalls
move $t0, $v0             # register the value we entered, in # register $t0
```

Commands to display integer on console:

```
li $v0, 1                 # pseudo-instruction for addi $v0, $0, 1
                          # code 1 from syscalls in $v0
move $a0, $t0             # o $t0 has the value we want to # display and we load it into
                          $a0 # as a parameter

syscall
```

Commands to display messages on the console:

To print a string to the screen you must first declare it in the .data section
of your program as follows:

You must first assign the string to the memory area with a label (in the case of the String_name1 example) so that you can access it using its address.

```
.data
String_name1: .asciiz "msg"
```

In case you want to break a line (to make the execution more readable) you must define a newline string.

```
String_name2: .asciiz "\n"
```

To print the string to the console you must write the following commands.

```
li $v0, 4 # code 4 from syscalls in $v0
la $a0, String_name1 # load $a0 with the address
                                    # of the String to print
syscall
```

The syscall commands are system function calls. With them you can, in addition to I/O operations, terminate a program (syscall with parameter $v0=10) or dynamically request memory (corresponding malloc command of C). You can find all functions in the Help menu of MARS.

Implementation of program flow control structures

Flow control instructions are the set of instructions that change the flow of program execution. Normally the order of execution of the commands is after each command is executed or immediately below. But there are commands which change either statically (without a condition) or dynamically (with a condition) the execution flow of the commands.

Flow control example, converting from C to assembly:

Example 1 - if

```
if( x = = 0) {
        … // Case true
}
else
{
        … // Case false.
}
…               // Next instruction
```

We assign the variable x to the register $t3

```
bnez $t3, else
…                       # case true
j endif
```

```
otherwise:
…                       # case false

endif:
…                       # next instruction
```

Example 2 - for

```
for (i = 0; i < 10; i++ )
{
        …               // code
}
…                       // next instruction
```

We assign the variable i to register $t3 and store the limit of the loop (here 10) in register $t4.

```
li $t3,0        # add $t3, $0, $0
li $t4, 10      # add $t4,$0.10

for:
bge $t3, $t4,endfor

…               # code

add $t3,$t3,1
j for

endfor:
…               # next instruction
```

Example 3 - multiple if

```
if (x > 0 && x < 10)
{
        … // code
}
…               // next instruction
```

We assign the variable x to register $t3 and store the constant to compare with (here 10) in register $t4.

```
li $t4.10
blez $t3, endif
bge $t3,$t4, endif

… # code

endif :
```

```
… # next instruction
```

b way:

```
add $t4,$0.10
bgtz $t3, cond2          # if (x > 0)
j endif

cond2:
blt $t3,$t4,is_true      # if (x < 10)
j endif

is_true:

… #code

endif:
… # next instruction
```

Memory usage

The data is stored in a separate memory section which is used exclusively for this purpose. For this reason the declaration of the table you will use must be done in the .data **section.** How to declare the array and the syntax of the load (lw) and store (sw) commands, which are used to transfer data between memory and register, are explained below.

Table declaration

```
.data              # in the .data section the array must be declared

Name_array: .space N # Name_array: give that name
                   # you want in the table
                   # .space N: indicates that we want to
                   # reserve space equal to N bytes for
                   # the Name_array array
```

load command (lw)

This instruction stores data in a register, which it has taken from a specific memory address.

Syntax: **lw Rt, Address(Rs)**	Meaning: **Rt=Memory[Address+Rs]**

Attention: where Rt, Rs are registers and where Address is the table name (label) you gave in the .data section.
Based on the above statement (in the .data section) we should write:
lw Rt, Name_array(Rs)

From the syntax of this command it is understood that we are requesting the memory data at address [Name_array+Rs] and which will be stored in the Rt register.

store command (sw)

This instruction stores data from a register at a specific address in memory.

Syntax: **sw Rt, Address(Rs)**	Meaning: **Memory[Address+Rs]=Rt**

The operation mode is similar to the **lw(load word) command.** The function that what this command does: Store the contents of the Rt register in memory specifically at address **[Address+Rs].**

The difference with the first command is that sw stores data in memory while lw reads the memory data.

Manage characters

A character needs 1byte to be stored. That's why there are the commands **lb(load byte)** and **sb(store byte)** so that 1 byte is read and written respectively, instead of 4 as is done with **lw** and **sw.** The way in which the commands for 1 byte are written is in full correspondence with those for 4 bytes, ie for example:

Syntax: **sb Rt, Address(Rs)**	Meaning: **Memory[Address+Rs]=Rt**

Command to read character from the console

```
li $v0, 12          # code 12 from syscalls in $v0
syscalls

move $s0, $v0       # record the value we read in register $s0
```

Command to print character to console

```
li $v0, 11          # code 11 from syscalls in $v0
li $s0, 'h'         # load $a0 with the address
move $a0, $s0       # this can also be written directly as li $a0, 'h'

syscall
```

Alignment

As you know from theory MIPS is a 32 bit processor. This implies that all its registers have a size of 32 bits or otherwise 4 bytes. We define as **word** the data of size 4 bytes which fit in a register.

When we define an array in memory with the .space command we specify its size equal to a number of **bytes** as shown before. But since the elements stored in it will be of size 4 bytes each, you must specify the size of the array to be a multiple of 4. So the first element of the array is in bytes 0 to 3. If you want to load it you should write the command *lw Rt, Address(Rs)* with Rs having a content of 0. If you now want the second element of the table it is in bytes 4 to 7. To load it you should write the same command but Rs it should contain the number 4.

Note: When defining a table you must also state what size the data will be stored in it. This is done with the command .align **n** where **n** is an integer that you will provide. This command means that the elements of the array are of size **2n**
. So for a 10 position array with 4 byte words you must write the command:

```
.align          # element size 4 bytes
2 label: .space 40    #array size 40 bytes = 10 elements
```

So, one easily follows that if one wants to declare an array with N characters, you must write the following commands:

```
.align          #item size 1 byte
0 label: .space N     #array size N bytes = N elements
```

An example of memory usage

```
.data
.align          # words 4 bytes
2 vector: .space 24     # array of 6 places


.text

# …


li $t1,4        # a way to access the 2nd element
lw $t0,vector($t1)      # of the table


# …
```

```
la $t2,vector                # one more way to access the 2nd
lw $t3,4($t2)                # element of the array

# …

la $t2, vector               # one more way to access the 2nd
addi $t2, $t2, 4             # element of the array
lw $t0,0($t2)
```

The number of commands is relatively large but their operation is very simple. It is recommended when programming in assembly to use the instruction set. You can also use pseudo-commands as if they were regular commands (for example *li, la, bgez, blt* kok). Memorizing all the commands is not required in this lab, however you should be able to use the commands listed in this file.

To get an A in the exam, your solution should be correct but also efficient. You should use as few commands as possible in your solution, but also make your solution readable and well commented.

For the next lab you have to implement the following lab exercises. During the workshop you will be tested orally on the codes you will deliver.

Exercise 1 – Byte manipulation (2 units)

Implement a program in MIPS assembly that will read an integer of type integrer *(num)* and display the number of 1's and 0's present in the number *num.* For example, your program should write to the console “Number 45 has 4 ones and 28 zeros” if the number you entered is num = 45 (= 00…0101101).

Execution Example:

Please give a number: 45
Number 45 has 4 ones and 28 zeros.

-- program is finished running --

Comments:
1) In red is the input of the user.
2) **Do not print** the last line. MARS prints it by default.

Exercise 2-Finding overlapping intervals (3 credits)

Implement in assembly a program that will ask the user to repeatedly enter two integers N1 and N2, which represent an interval [N1, N2]. If the user enters a negative N1 < 0 the program terminates. Also, it should be checked that N1 is smaller than N2.

The program should find the maximum interval resulting from either overlapping intervals or not. To find the maximum interval each time, four cases illustrated in Figure 1 should be checked. The program should keep at each iteration the left edge and right edge of the current maximum interval (illustrated as min and max in Figure 2). In the last iteration, min and max contain the final maximum interval.

More specifically, the user, at iteration 0, enters an interval [N1, N2], so initially min = N1 and max = N2. In iteration 1, the user enters the next interval [N1, N2], and min, max are updated according to the control cases in Figure 1 between the result of the previous iteration and the entered interval, and so on. If two intervals do not overlap (case 4 of Figure 1), then the result should be the maximum interval of these two intervals. At the end of the program, the final maximum interval is printed.

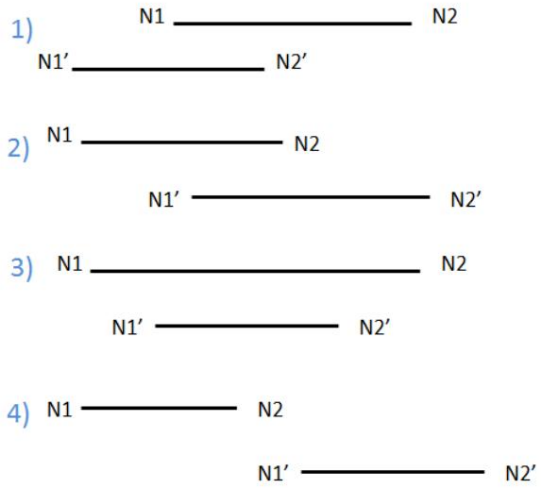
To find the maximum of the interval, there are four cases of overlapping intervals:

Case 1: The maximum interval of the previous iteration is further to the right, and there is an overlap with the result of the current iteration. So the maximum interval is [N1', N2].

Case 2: The maximum interval of the previous iteration is further to the left, and there is an overlap with the interval of the current iteration. So the maximum interval is [N1, N2'].

Case 3: The entered interval of the current iteration is included in the maximum interval of the previous iteration, so the maximum interval is [N1, N2].

Case 4 : The intervals of the previous and current iteration do not overlap, so the result is the maximum interval of the two intervals.



Picture 1

Example Execution (during the iterations in Figure 2):

Iteration 0
Please give N1: 1
Please give N2: 3
Given range: [1,3]

Iteration 1
Please give N1: 0
Please give N2: 2
Given range: [0,2]

Iteration 2
Please give N1: 1
Please give N2: 3
Given range: [1,3]

Iteration 3
Please give N1: 1
Please give N2: 4
Given range: [1,4]

Iteration 4
Please give N1: 5
Please give N2: 6
Given range: [5,6]

Iteration 5
Please give N1: 5
Please give N2: 10
Given range: [5,10]

Iteration 6
Please give N1: -4

The max final union of ranges is [5,10].

-- program is finished running –

Comments:

1) In red is the input of the user.
2) **Do not print** the last line. MARS prints it by default.
3) Notice that iteration 6 is terminated **before it asks** for N2 since N1<0.

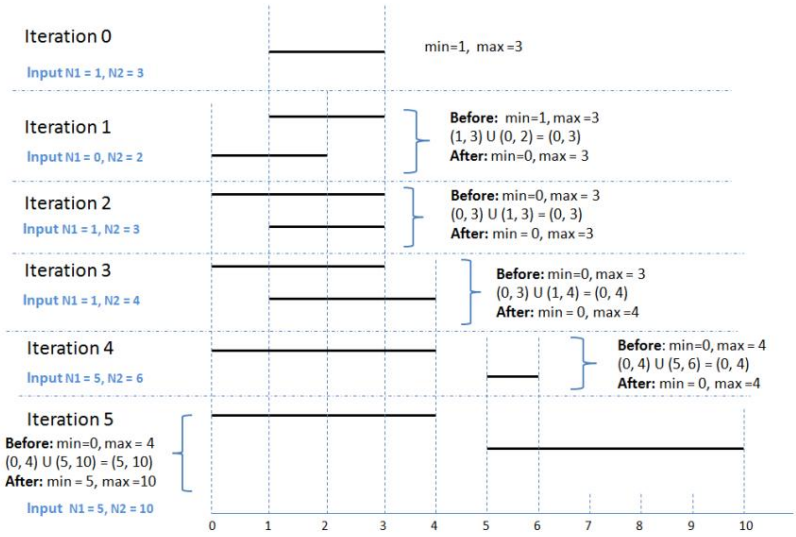


Figure 2

On each iteration, the current maximum interval is stored in the min and max variables. The figure shows the iterations of an example run. In each iteration, the before tag displays the maximum interval of the previous iteration, while the after tag displays the maximum interval calculated from the result of the previous iteration and the entered interval (Input).

Exercise 3- Merging two sorted arrays of characters (3 credits)

Implement in assembly a program that accepts as input two sorted strings which will be merged into a new single sorted string.

Execution Example:

Please enter the first string: aabcccdgjkk
Please enter the second string: abbefk

String 1: aabcccdgjkk
String 2: abbefk

Merged string: aaabbbcccdefgjkkk

Comments:
1) In red is the input of the user.
2) **Do not print** the last line. MARS prints it by default.

Exercise 4- Inner product of vectors (3 units)

Implement in assembly a program that will ask the user to enter two 5-position vectors and write the result of the inner product to the console. The result of the inner product of two vectors , ÿ ÿ! is given by the formula:

$$x^T y = \sum_{i=0}^{n-1} x_i y_i$$

Execution Example:

Please enter the size of the vectors: 3
Enter element 0 of vector x: 5
Enter element 1 of vector x: 3
Enter element 2 of vector x: 8
x = [5 3 8]

Enter element 0 of vector y: 1
Enter element 1 of vector y: 2
Enter element 2 of vector y: 6
y = [1 2 6]

The inner product of x*y is: 59

-- program is finished running –

Comments:
1) In red is the input of the user.
2) **Do not print** the last line. MARS prints it by default.

Assignments should be uploaded to eclass immediately after your lab exam.