CSCE 22104

Lab Report

Brent Marcus Orlina

ID: 011019116

Lab Section 001

Lab 3

Introduction

This lab's goal was to learn basic MIPS assembly by creating a basic program and translating two C++ programs, `arrays.cpp` and `occurence.cpp`, into MIPS assembly. The goal `sum3.s` was allow the user to input 3 integers and its sum. The `arrays.cpp` program consisted of initializing an array of integers and printing its elements. Finally, the `occurences.cpp` program consisted of allowing the user to input 10 integers into an array and another integer, n, outputting the number of occurences of n in the array.

Approach

```
.data
    input_prompt: .asciiz "Enter an Integer >> "
    sum_text: .asciiz "The sum is: "

.text
...
main:
    # print prompt
    li $v0, 4               # opcode to print string
    la $a0, input_prompt    # load address of string you want to print
    syscall

    # read first integer
    li $v0, 5       # opcode to read in an integer
    syscall
    move $s0, $v0   # syscall puts int to £v0, so move it to £s0 since you want to use £v0 again later
    ...
```

Listing 1:

For the `sum3.s` program its goal is to read three integers from the user and output its sum to the terminal. First, a prompt for the user to input an integer is first printed. This is done by loading an opcode of `4` into the register `$v0` and loading the address of the string to be printed into the register `$a0`, using the instructions `li` and `la` respectively. Then, the instruction `syscall`, short for system call, is called which asks the operating system to execute the given opcode in the register `$v0` which in this case is to print a string.

Similarly, the program uses another system call to read an integer from the user input. An opcode of `5` is loaded into the register `$v0` and `syscall` is called. After the user inputs an integer, the operating system writes the given integer into register `$v0`. This is moved into a different register, in this case register `$s0`, since `$v0` will be used again for more system calls.

The program has successfully read an integer from the user. To read the two other integers, the six instructions shown in listing 1 is simply repeated two more times, with the two new integers moved to registers `$s1` and `$s2` respectively.

```
...
.text
...
main:
    ...
    add $t0, $s0, $s1   # temporary for £s0 + £s1
    add $s3, $t0, $s2   # temporary + £s2
    ...
```

Listing 2:

To calculate the sum of the three integers, the instruction `add` is used. Since it is impossible to add more than two integers at once in MIPS assembly (excluding the use of SIMD instructions), the program must use two addition instructions, as shown in listing . The program first adds the two integers in the registers `$s0` and `$s1` and stores the result to register `$t0`. Then, the program adds the result of the first sum and the integer in `$s2`, storing the result in register `$s3`.

```
.data
    input_prompt: .asciiz "Enter an Integer >> "
    sum_text: .asciiz "The sum is: "

.text
...
main:
    ...
    # print sum text
    la $a0, sum_text
    li $v0, 4
    syscall

    # print sum
    li $v0, 1       # opcode to print an integer
    move $a0, $s3   # move the integer you want to print to the argument register
    syscall
    ...
```

Listing 3:

To output the sum of the three integers, a string is first printed, stating that the following integer is the sum of the three given integers. The sum is then printed. This is done similarly to printing the string `input_prompt` earlier, with the exception that opcode `1` is used and the sum is moved from register `$s3` to `$a0` when printing the sum, as shown in listing .

```
1  ...
2  .text
3  .global _start
4  _start:
5      jal main
6      li $v0, 10
7      syscall # Use syscall 10 to stop simulation
8  
9  main:
10     ...
11     jr $ra
12 
```

Listing 4:

The program actually starts at the label `_start`, which immediately jumps to the label `main`, as shown in listing . The `jal` instruction jumps to the given label, and saves the address it of the instruction after from where it jumped from in the register `$ra`. The program, after executing all of the instructions described previously, jumps back to line 9 in listing , with the instruction `jr`, so that the program can properly exit using the correct syscall. The two translated C++ programs also exit similarly as described. Describing the syscalls to read or print data will also be skipped since it has already been described in this program and so that the attention can be focused on the interesting parts of the implementation.

```cpp
#include <iostream> // Do not translate!

int main() {
    std::cout << "Numbers: " << std::endl;
    int x[] = { 18, 12, 6, 500, 54, 3, 2, 122 };

    for(int i = 0; i < 8; i++)
        std::cout << x[i] << std::endl;

    return 0;
}
```

Listing 5:

Listing shows the C++ program that was translated. The first part of the program is printing a string `"Numbers: "` followed by a newline. The second part of the program is initializing an integer array with eight pre-defined values. The third part of the program is that each integer in the array is printed followed by a newline using a for-loop.

```
.data
    numbers_string: .asciiz "Numbers: "
    x_array: .word 18, 12, 6, 500, 54, 3, 2, 122
    newline: .asciiz "\n"

.text
main:
    # print "Numbers: \n"
    la $a0, numbers_string
    li $v0, 4
    syscall

    # print newline
    la $a0, newline
    li $v0, 4
    syscall
    ...
```

Listing 6:

Listing shows the implementation of the first two parts of the `arrays.cpp` program. Initializing the intger array `x` in MIPS assembly was done by declaring a label for the array in the `.data` section and using the `.word` directive to indicate that the following sequence of data has the size of a word, 32-bits in the used architecture.

```
...
.text
main:
    ...
    li $s1, 0               # initialize i
    li $s2, 32              # array size
    la $s0, x_array         # address of number array

loop_start:
    bge $s1, $s2, loop_exit    # check if i is equal to array size; goto after_loop if equal

    add $t0, $s0, $s1   # get address of array[index]
    lw $t1, 0($t0)      # load the actual content in the address

    # print integer in £t1
    # print newline

    addi $s1, 4 # increment offset

    j loop_start      # loop back

loop_exit:
    jr $ra
```

Listing 7:

The for-loop can be broken down into four parts: intialization, conditions checking, main body, incrementing. In listing , each part can be mapped explicitly. Initialization occurs from lines 5 to 7. The variable `i` is initialized to 0 in register `$s1`, which will be used to index into the integer array. The array size is initialized to `32` in register `$s2`. The array size is 32 and not 8 since each element in the array is 4 bytes. Therefore it is initialize it to $8 \times 4 = 32$. Finally, the address of the array is initialized to register `$s0`.

The next part is checking the conditions of the loop to keep running. In listing , the condition is that the variable `i` is less than `8`, which is the size of the array. That condition can be inverted so that if the condition is met, the loop can exit and otherwise, the loop can keep running. This is shown in line 10 of listing with the instruction `bge` checking that the value in register `$s1` is greater than or equal to (the inversion of strictly less than) the value in register `$s2`. If the condition is met, the program jumps to the label `loop_exit` in which case the program exits. Otherwise, the loop continues to the main body of the loop.

The main body of the loop consists of printing the element at the current index. To do this, the program must calculate the address of the element in memory and load it to a register as shown in lines 12 and 13. This is done by adding the base address of the array and the current index and using the result to load it into the register `$t1`. Then, the element is printed along with a new line. It is not shown in listing since it is trivial. However the implementation can be seen in the appendix in listing .

The final part of the loop increments the index, `i`, by `4` with the instruction `addi`. It is incremented by 4 since the elements are each 4 bytes in size. Thus, to get to the next element, the program must skip every 4 bytes to get to the first byte of the next element. After incrementing, the loop is ready to start over by jumping back to the `loop_start` label as shown on line 20. With the loop complete, the translation of the `arrays.cpp` program is completed.

```cpp
#include <iostream> // Do not translate!

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

    std::cout << "Type in array numbers:" << std::endl;

    for(int i = 0; i < 10; i++)
        std::cin >> x[i];

    std::cout << "Type in occurrence value:" << std::endl;
    std::cin >> occur;

    std::cout << "Occurrences indices are:" << std::endl;
    for(int i = 0; i < 10; i++) {
        if(x[i] == occur) {
            std::cout << i << std::endl;
            count++;
        }
    }

    std::cout << "Number of occurrences found:" << std::endl;
    std::cout << count;
    return 0;
}
```

Listing 8:

Listing shows the `occurences.cpp` program to be translated. The first part of the program is initializing the variables that the program will need. The second part is printing an input prompt and reading ten integers from user input and storing them in an integer array `x`. Similarly, the third part is printing an input prompt and reading an integer from user input storing it to the variable `occur`. The fourth part is checking the number of times that `occur` is in the array `x` and printing the indices that it is in. The final part is printing the number of times that `occur` is in, stored in the variable `count`.

```
.data
    number_array: .space 40
    input_prompt:  .asciiz "Type in array numbers:\n"
    occur_prompt:  .asciiz "Type in occurence value:\n"
    occur_indices: .asciiz "Occurence indices are:\n"
    occur_number:  .asciiz "Number of occurences found:\n"
    newline: .asciiz "\n"
...
```

Listing 9:

For the first part, only the strings and the array are needed to be initialized in the `.data` portion of the program since those live in memory while `occur` and `count` can simply be initialized into registers once they are needed. The `x_array` is initialized with the `.space` directive to signify that it should be empty with a size of `40`, since there are ten elements in the array, each the size of 4 bytes.

The second part of the program is trivial as it is extremely similar to the implementation the translated `arrays.cpp` program, with the only difference that it reads in an integer from user input in the main body of the loop, and uses the store word instruction `sw` instead of the load word instruction `lw`. The third part is also trivial since it is simply printing an input prompt and reading an integer from user input, which is done in the `sum3.s` program. The implementations can be seen in the appendix, listings and .

```
.data
    number_array: .space 40
    ...

.text
main:
    li $s0, number_array    # initialize array address
    li $s1, 40              # initialize array size
    li $s2, 0                # initialize i

occur_loopstart:
    bge $s2, $s1, occur_loopexit

    add $t0, $s0, $s2
    lw $t1, 0($t0)

    # if(x[i] = occur) ...

    addi $s2, $s2, 4
    j occur_loopstart

occur_loopexit:
    ...
```

Listing 10:

The fourth part is not as trivial as the loop contains an if-statement, however it is similar to checking the conditions in a loop. Listing shows the loop that contains the if-statement. It can be seen that it is extremely similar to the translation of the `arrays.cpp` program, as shown in listing . Note that the index, stored in register `$s2`, still increments by 4.

```
1  .data
2      number_array: .space 40
3      ...
4
5  .text
6  main:
7      li $s0, number_array    # initialize array address
8      li $s1, 40              # initialize array size
9      li $s2, 0                # initialize i
10
11 occur_loopstart:
12     ...
13     bne $s3, $t1, occur_ifexit
14
15 occur_ifstart:
16     li $v0, 1       # print index
17     srl $a0, $s2, 2 # need to divide index by 4
18     syscall
19
20     li $a0, newline
21     li $v0, 4
22     syscall
23
24     # increment count
25     addi $s4, $s4, 1
26
27 occur_ifexit:
28     ...
29 occur_loopexit:
30     ...
```

Listing 11:

Listing shows the translation of the if-statement in listing . The if-statement can be broken into two parts: conditions checking, and the main body. The condition for the if statement in listing is that if the current element is equal to `occur`, then the main body can run. Otherwise, the main body is skipped. Similar to conditions checking for for-loops, the condition can be inverted so that if the inverted condition is met, the main body can be skipped. Otherwise, the main body runs. Line 13 of listing does this, by checking the value in register `$s3`, the variable `occur`, is not equal to the value in register `$t1`, the value in the current index. If the condition is met, it jumps to the label `occur_ifexit`, skipping the main body of the if-statement, after the label `occur_ifstart`, entirely.

If the condition is not met, then the main body runs, which prints the current index and increments the variable `count` in register `$s4` on line 25. Note that the program is not simply moving the index from register `$s2` to register `$a0` since it is still skipping by 4. To output the correct looking index, it divides it by 4 by bit-shifting the index to the right by 2 with the instruction `srl`. The final part is, again, trivial as it simply prints a string and an integer. The implementation can be seen in the appendix in listing . With all parts complete, the translation for the `occurences.cpp` program is completed.

Experimentation

The program `sum3.s` was tested by using positive and negative integers as well as zero, verifying that the output of the program was the correct sum of the three given integers. The translation of the program `arrays.cpp` was verified by simply observing that all the elements in the array in the program were printed and in the correct order. The translation of the program `occurences.cpp` was verified by inputting ten integers, ensuring there were repeats, and choosing the occurence value to be one of the repeating integers. Then it was observed if the program outputted the correct indices and count of the chosen occurence value.

Results & Discussion

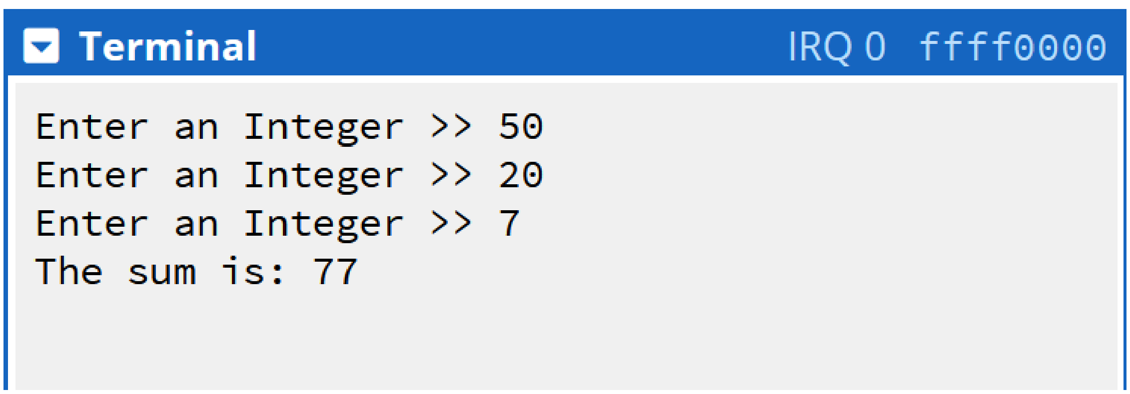

Figure 1:

The program `sum3.s` works as expected. Figures and shows the sums of three numbers. The first figure tests only positive numbers while the second figure also tests negative numbers and a zero with each figure showing the correct sums.

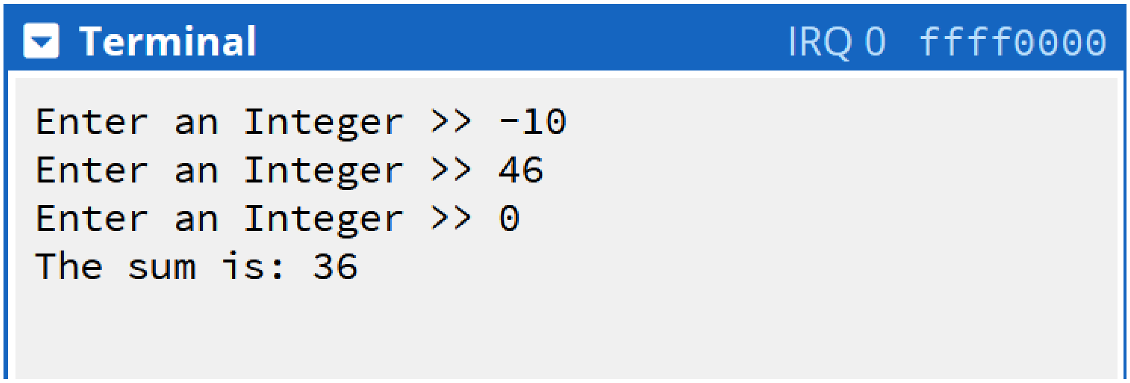

Figure 2:

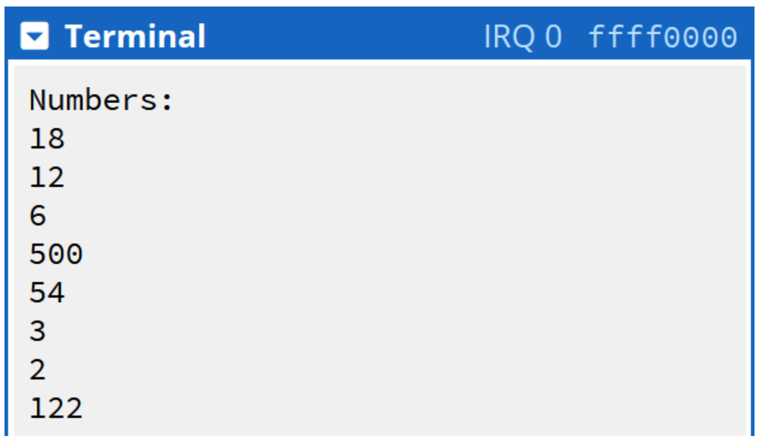

Figure 3:

The translation of the program `arrays.cpp` works as expected, with figure showing the translated program printing all the elements of the array in the correct order.

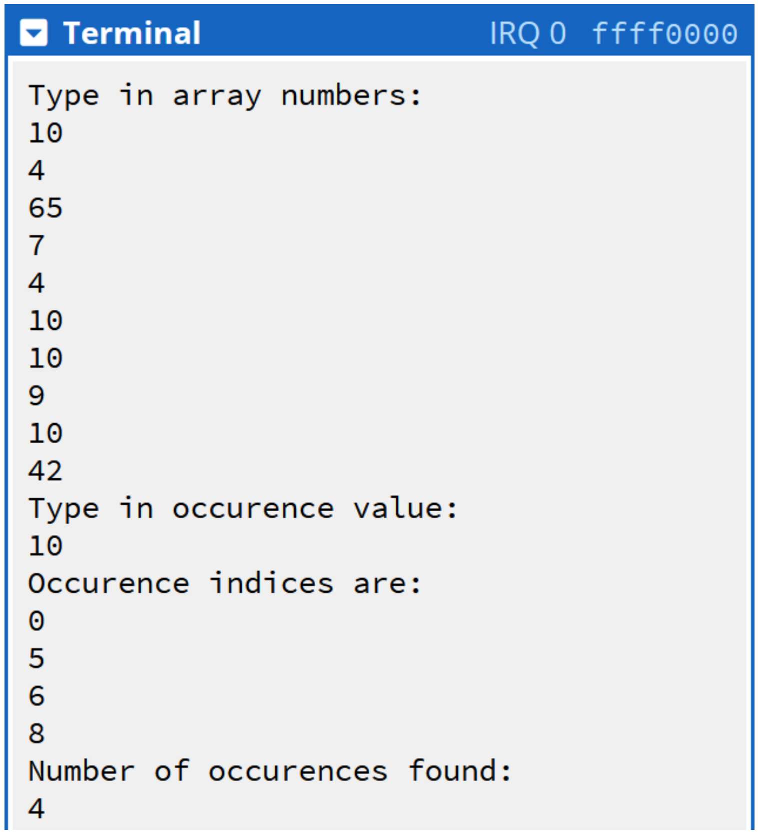

Figure 4:

The translation of the program `occurences.cpp` works as expected, with figure correctly printing the 0-indexed indices and the number of occurences found of the user-given occurence value in the user-given integer array.

Conclusions

The programs implemented in MIPS assembly in this lab works correctly. The knowledge learned from implementing the `sum3.s` program was reading user input, printing strings to the terminal, initializing strings into memory, and adding integers inside of registers. In translating the program `arrays.cpp`, initializing arrays into memory, creating a for-loop, and indexing into an array in MIPS assembly was learned. In translating the program `occurences.cpp`, creating an if-statement and writing into an array in MIPS assembly was learned.

Appendix

```
...
.text
main:
    ...
loop_start:
    ...
    # print integer in £t1
    move $a0, $t1
    li $v0, 1
    syscall

    # print newline
    la $a0, newline
    li $v0, 4
    syscall
    ...

loop_exit:
    ...
```

Listing 12:

```
.data
    number_array: .space 40
    input_prompt:  .asciiz "Type in array numbers:\n"
    ...

.text
main:
    li $a0, input_prompt
    li $v0, 4
    syscall

    li $s0, number_array    # intialize array address
    li $s1, 40              # array size
    li $s2, 0                # initialize i

input_loopstart:
    # check conditions
    bge $s2, $s1, input_loopexit

    li $v0, 5
    syscall

    add $t0, $s0, $s2
    sw $v0, 0($t0)        # store the int that's read to array

    addi $s2, $s2, 4

    j input_loopstart

input_loopexit:
    ...

```

Listing 13:

```
.data
    ...
    occur_prompt:  .asciiz "Type in occurence value:\n"
    ...

.text
main:
    ...
    li $a0, occur_prompt
    li $v0, 4
    syscall

    li $v0, 5
    syscall
    move $s3, $v0   # occurence value
    li $s4, 0       # count
    ...
```

Listing 14:

```
.data
    ...
    occur_number:  .asciiz "Number of occurences found:\n"

.text
main:
    ...
    li $a0, occur_number
    li $v0, 4
    syscall

    # print number of occurences found
    move $a0, $s4
    li $v0, 1
    syscall

    jr $ra

```