CSCE 22104

Lab Report

Brent Marcus Orlina

ID: 011019116

Lab Section 001

Lab 4

Introduction

This lab's goal was to create and call a function, `compare_and_swap`, in MIPS that is used by a bubble sort algorithm provided. The function is provided with two memory addresses in the registers `$a0` and `$a1`. If the value in the memory address `$a1` is less than the value in the memory address `$a0`, then the values should be swapped. In other words, the lesser value should live in `$a0` and the greater the value should live in `$a1`.

To correctly call the function for the bubble sort algorithm, the addresses of the element of a given index and the element after it is stored in the registers `$a0` and `$a1`. The given index has already been provided in the bubble sort algorithm. Then, the function is called using the jump-and-link instruction, `jal`.

Approach

```
1 compare_and_swap:
2     lw $t0, 0($a0)
3     lw $t1, 0($a1)
4     ble $t0, $t1, compare_and_swap_exit
5     sw $t1, 0($a0)
6     sw $t0, 0($a1)
7 compare_and_swap_exit:
8     jr $ra
```

Listing 1: Implementation of the `compare_and_swap` function in MIPS.

The function `compare_and_swap` in listing 1 is called with the addresses of the elements to be compared and swapped in registers `$a0` and `$a1`. The contents in the two addresses is loaded into two registers `$t0` and `$t1` respectively. The values are then compared with the branch instruction `ble`. That is, if the value in `$t0` is already less than or equal to the value in `$t1`, there is no need to swap and jumps to `compare_and_swap_exit`. Otherwise, the value in `$t0` is greater than in `$t1` which means that the two elements must be swapped. The swap occurs in line 5 and 6. The value in `$t1` is stored into the memory address in `$a0` and the value in `$t0` is stored into the memory address in `$a1`. In `compare_and_swap_exit`, it returns to the caller by jumping to the return address in `$ra`.

Listing 2 shows the bubble sort algorithm. Translation from C++ to MIPS of lines 1 to 3 were provided, leaving the call to `compare_and_swap` to be implemented. In listing 3, the base address

```c
void bubble_sort(int arr[], int size) {
    for(int i = 0; i < size - 1; i++)
        for(int j = 0; j < size - i - 1; j++)
            compare_and_swap(&(arr[j]), &(arr[j + 1]));
}
```

Listing 2: The bubble sort algorithm.

```asm
1 bubble_sort_array_accesses:
2     sll $t0, $s3, 2
3     add $a0, $s0, $t0
4     move $a1, $a0
5     addi $a1, $a1, 4
6     jal compare_and_swap
```

Listing 3: Implementation of the call to the `compare_and_swap` function in MIPS.

of the array is stored in register `$s0` and the inner-loop index, `j` is stored in register `$s3`. The arrays store an `int` which have a size of 4 bytes. Therefore, we must multiply the inner-loop index by 4 to achieve the correct offset. Line 2 of listing 3 achieves the same goal by left shifting by 2. Then, the address `&(arr[j])` is calculated by adding the base address of the array with the offset, which is stored into the first argument register, `$a0`. The address `&(arr[j + 1])` is calculated by adding 4 to the previously calculated address, which is stored into the second argument register, `$a1`. Now that the arguments have been provided for the `compare_and_swap` function, it is finally called with the `jal` instruction.

Experimentation

The bubble sort algorithm, therefore the function `compare_and_swap` implementation as well, was tested by sorting an array containing random integers in random order and an array of powers of two in decreasing order. The bubble sort algorithm worked as expected and sorted each array in increasing order.

Results & Discussion

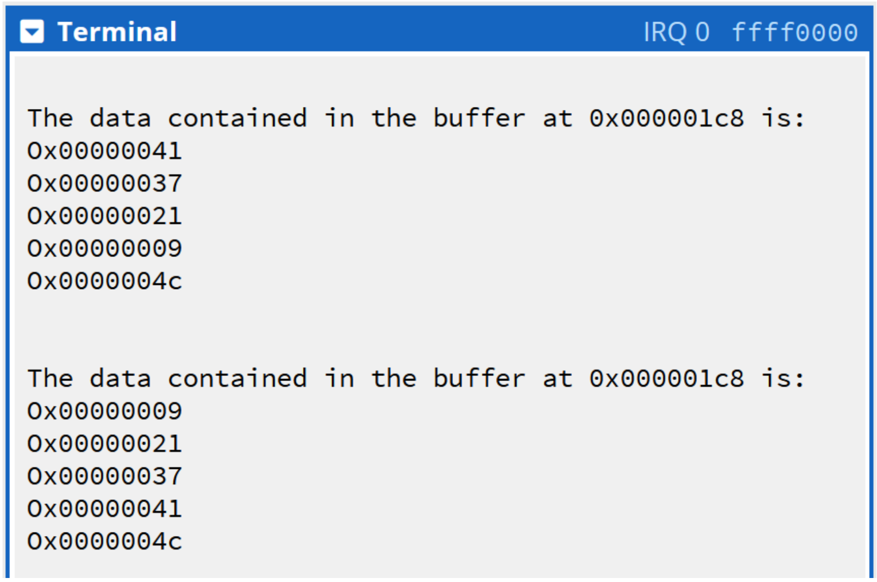

Figure 1: The output of the bubble sort algorithm written in MIPS given a random unsorted array before and after the bubble sort

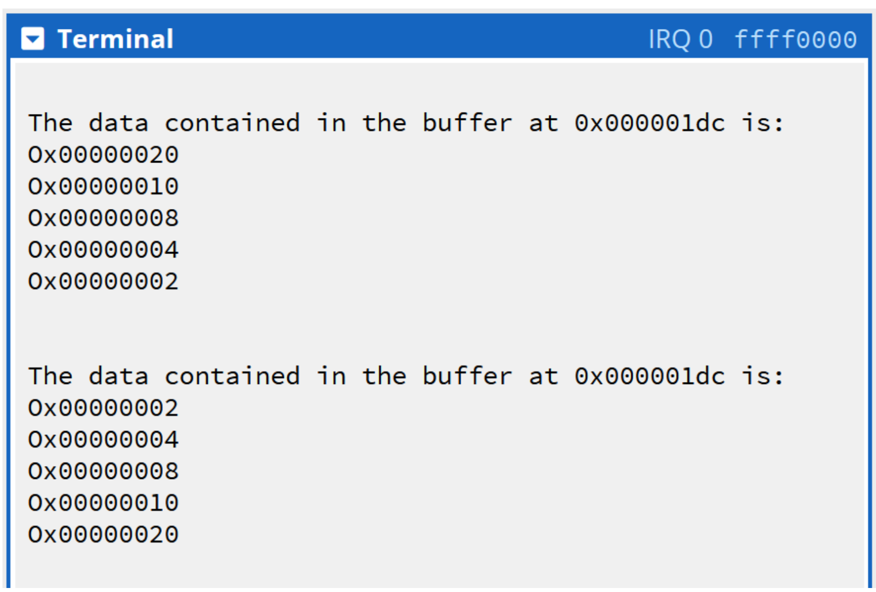

Figure 2: The output of the bubble sort algorithm written in MIPS given a reverse sorted array before and after the bubble sort.

The bubble sort algorithm works as expected. Figures 1 and 2 shows the arrays before and after the running the bubble sort algorithm, sorting in increasing order. Since the bubble sort displays the correct behavior, the `compare_and_swap` function works correctly.

Conclusions

The `compare_and_swap` function works correctly. The knowledge learned from this lab was learning how to translate a function from C++ into MIPS assembly as well as correctly providing the expected arguments and calling the translated function. The skills practiced in this lab was writing and reading MIPS assembly.