Computer Architecture (CSL3020)

Lab Assignment 2 - Report

Soumen Kumar
Roll No-B22ES006

1 Introduction

1.1 Objective

This report outlines the development of MIPS assembly programs for performing basic arithmetic operations and a payroll calculator. The tasks aimed to enhance understanding of MIPS architecture, instruction sets, and assembly language programming.

1.2 Scope

The report covers two main tasks:

- **Task 1:** Implementation of basic arithmetic operations (addition, subtraction, multiplication) and input/output operations.
- **Task 2:** Development of a payroll calculator that computes gross salary, deductions, and net salary for employees.

2 MIPS Programming Experience

2.1 Experience with MARS

Working with the MARS (MIPS Assembler and Runtime Simulator) software provided hands-on experience in assembly programming. MARS offers a robust environment for testing and debugging MIPS code, allowing real-time observation of register and memory changes. The intuitive interface made it easier to identify errors and understand the flow of data within the program.

2.2 Challenges Encountered and Solutions

2.2.1 Understanding MIPS Instructions

Initially, understanding how to effectively use the MIPS instruction set for arithmetic operations was challenging. The MIPS instruction set is quite low-level, and managing register states across multiple calculations required a deep understanding of how data is stored and manipulated in registers. To overcome this challenge:

- **Study and Practice:** I spent additional time studying the MIPS instruction set and consulted various resources to gain a clearer understanding of the instructions and their usage.
- **Code Examples:** Reviewing and analyzing example code snippets helped in understanding practical applications of the instructions.
- **Incremental Testing:** I implemented arithmetic operations incrementally and tested them thoroughly to ensure correct behavior before moving on to more complex calculations.

2.2.2 Handling Input/Output

Managing I/O operations in MIPS, particularly the correct use of system calls for reading and writing, was another challenge. The MIPS assembly language requires explicit handling of system calls to perform I/O operations. To address this challenge:

- **Consult Documentation:** I referred to the MIPS syscall documentation to understand the correct usage of system call numbers and arguments for I/O operations.
- **Example Code and Debugging:** I tested I/O operations with example code to see how different system calls work and used debugging techniques to identify and resolve issues with incorrect outputs or inputs.

2.2.3 Avoiding Register Overwrite

Ensuring that register values were correctly reset between iterations in the payroll calculator was a key challenge. Incorrect register handling could lead to erroneous calculations and results. To overcome this issue:

- **Thorough Testing:** I performed extensive testing to check that register values were correctly managed and reset as needed between calculations.
- **Debugging Tools:** I used debugging tools to step through the code and monitor register values to ensure they were being handled correctly.
- **Code Reviews:** I reviewed the code to ensure that register usage was consistent and followed best practices to prevent unintended overwrites.

3 Task 1: Arithmetic Operations

3.1 Objective

Implement MIPS programs to perform basic arithmetic operations and manage input/output.

3.2 Code Explanation

- **Addition:** The program adds two integers and stores the result in a designated register. Key operations include loading the values into registers and using the `add` instruction.
- **Subtraction:** The program subtracts one integer from another using the `sub` instruction, demonstrating basic arithmetic capabilities in MIPS.
- **Multiplication:** The program multiplies two integers using the `mul` instruction. This subtask highlighted the importance of managing register states to avoid overwriting values.
- **Input/Output Operations:** The program prompts the user for input using system calls, stores the values in registers, and prints the results after performing arithmetic operations. The sequence of `syscall` instructions was crucial in managing input and output.

3.3 Code Implementation

```
.data
prompt1: .asciiz "Enter first integer: "
prompt2: .asciiz "Enter second integer: "
result:  .asciiz "Result: "

.text
main:
    # Prompt and input first integer
    li v0, 4
    la a0, prompt1
    syscall

    li v0, 5
    syscall
    move t0, v0  # Store first integer in $t0

    # Prompt and input second integer
    li v0, 4
    la a0, prompt2
    syscall

    li v0, 5
    syscall
    move t1, v0  # Store second integer in $t1

    # Perform addition
    add t2, t0, t1  # $t2 = $t0 + $t1

    # Display result
    li v0, 4
    la a0, result
```

```
    syscall

    li v0, 1
    move a0, t2
    syscall

    # Exit
    li v0, 10
    syscall
```

Listing 1: Part of Task1 Code

3.4 Output

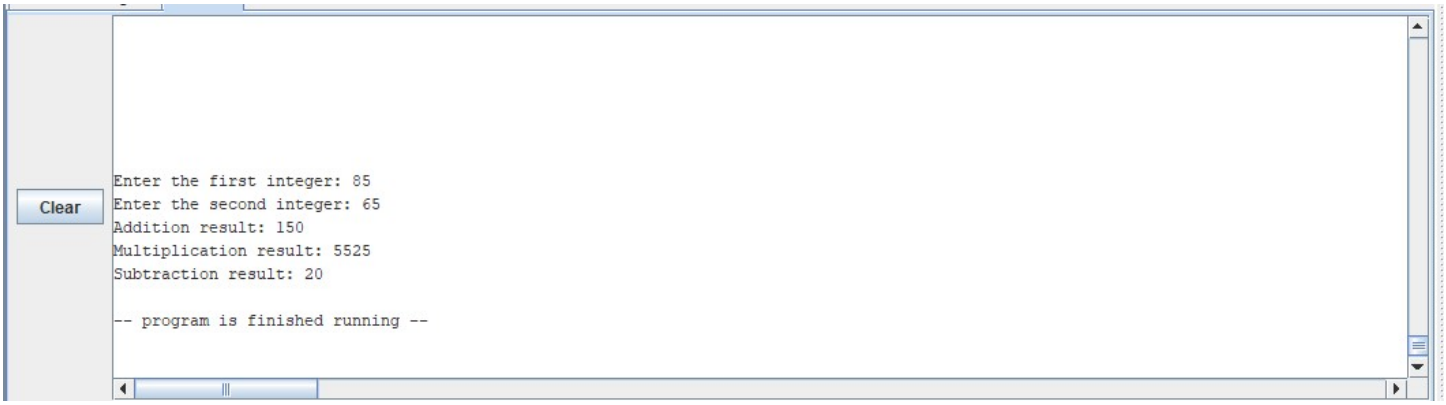

Figure 1: Screenshot of the output after code implementation

4 Task 2: Payroll Calculator

4.1 Objective

Develop a payroll calculator to compute gross salary, deductions, and net salary for employees.

4.2 Code Explanation

- **Input:** The program prompts the user to input regular hours, overtime hours, and hourly wage. These inputs are stored in registers for further calculations.
- **Gross Salary Calculation:** Gross salary is computed as the sum of regular salary and overtime pay, with overtime being calculated at 1.5x the regular rate.
- **Deductions Calculation:** The program calculates tax and insurance deductions based on predefined percentages. This was a critical section where managing register values correctly across multiple iterations was essential.
- **Net Salary Calculation:** Net salary is derived by subtracting total deductions from gross salary. The program then outputs the calculated values.
- **Looping for Multiple Employees:** The program asks if the user wants to enter details for another employee, looping back if necessary. Managing register states across multiple iterations required careful attention to ensure accurate results.

4.3 Code Implementation(Snippet of main logic)

```
.text
main:
    # Main loop for multiple employees
    loop:
        # Input regular hours
        li v0, 4
        la a0, prompt_hours
```

```
        syscall

        li v0, 5
        syscall
        move t0, v0  # Store regular hours in $t0

        # Input overtime hours
        li v0, 4
        la a0, prompt_ot_hours
        syscall

        li v0, 5
        syscall
        move t1, v0  # Store overtime hours in $t1

        # Input hourly wage
        li v0, 4
        la a0, prompt_wage
        syscall

        li v0, 5
        syscall
        move t2, v0  # Store hourly wage in $t2

        # Calculate gross salary
        mul t3, t0, t2  # Regular salary: $t3 = $t0 * $t2
        mul t4, t1, t2  # Overtime pay before 1.5x: $t4 = $t1 * $t2
        li t5, 3
        div t4, t5  # Overtime pay 1.5x: $t4 = $t4 * 1.5
        add t6, t3, t4  # Gross salary: $t6 = $t3 + $t4

        # Output gross salary
        li v0, 4
        la a0, output_gross
        syscall

        li v0, 1
        move a0, t6
        syscall

        # Calculate deductions
        li t7, 10
        div t7, t6  # Example tax deduction: 10%
        move a0, v0  # Tax deduction in $a0
        li t8, 200  # Example insurance deduction
        add t9, a0, t8  # Total deductions

        # Output deductions
        li v0, 4
        la a0, output_deductions
        syscall

        li v0, 1
        move a0, t9
        syscall

        # Calculate net salary
        sub t10, t6, t9  # Net salary: $t10 = $t6 - $t9

        # Output net salary
        li v0, 4
        la a0, output_net
        syscall

        li v0, 1
        move a0, t10
        syscall

        # Prompt for another calculation
        li v0, 4
        la a0, prompt_another
        syscall

        li v0, 5
        syscall
        bnez v0, loop  # Repeat if non-zero (yes)
```

```
84
85      # Exit
86      li v0, 10
87      syscall
```

Listing 2: MIPS Code for Payroll Calculator

4.4 Output

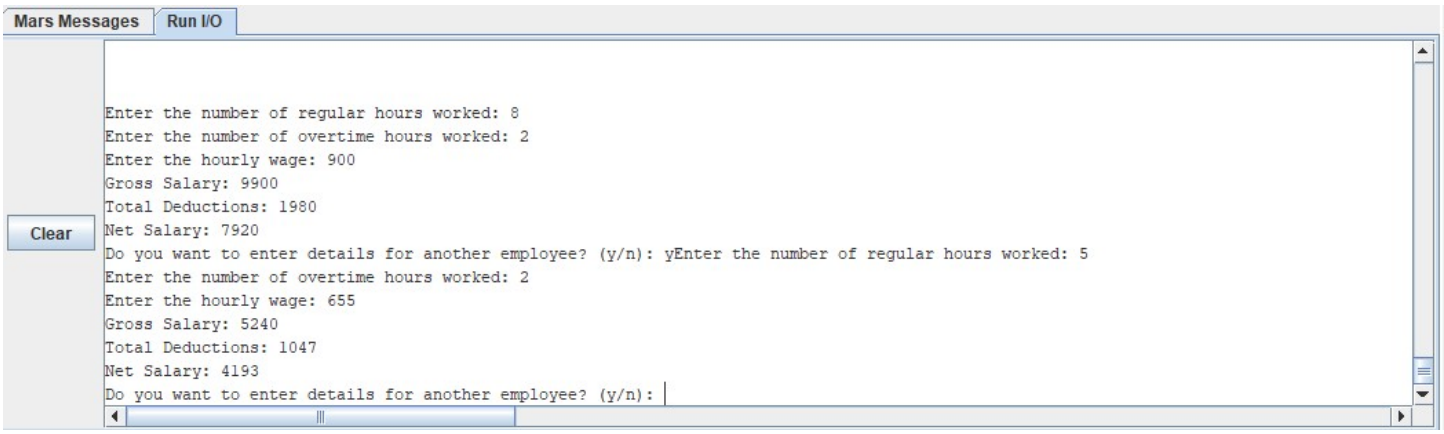

Figure 2: Screenshot of the output after code implementation

5 Reflection and Learning

5.1 Key Learnings

The tasks deepened my understanding of MIPS assembly language, particularly in handling arithmetic operations, managing register states, and implementing loops and conditional logic. The experience of developing a payroll calculator provided practical insights into how assembly programming can be applied to real-world problems.

5.2 Relevance of MIPS Programming

MIPS programming remains relevant in educational contexts, particularly for understanding the low-level operations of CPUs and the basics of computer architecture. While high-level languages are more commonly used in software development today, the principles learned from MIPS programming are foundational to understanding how compilers, assemblers, and processors work.

6 Conclusion

This assignment provided valuable hands-on experience in MIPS assembly programming, reinforcing the importance of careful register management and the challenges of low-level programming. The successful implementation of arithmetic operations and a payroll calculator in MIPS demonstrates a solid understanding of the architecture and programming paradigms.

7 References

- *Patterson, D. A., & Hennessy, J. L. (2013). Computer Organization and Design: The Hardware/Software Interface.*
- MIPS Assembly Language Programming - `https://chortle.ccsu.edu/AssemblyTutorial/index.html`
- MARS MIPS Simulator Documentation - `http://courses.missouristate.edu/KenVollmar/mars/`