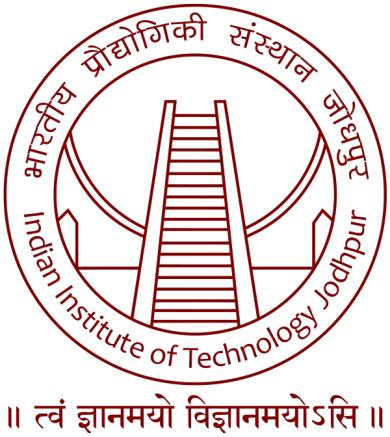
CSL3020

Computer Architecture



Lab-2 Report

By-B22EE024

Dhyey Findoriya

Contents

1. Software Installlation

MARS (MIPS Assembler and Runtime Simulator) is a software tool used to assemble and run MIPS assembly language programs. To install MARS:

- Download the MARS .jar file from the official website..
- Ensure **Java Runtime Environment (JRE)** is installed on your machine.
- Run the MARS .jar file by double-clicking it or using the command line:
  `java -jar Mars.jar`

MARS provides an easy-to-use interface to write, assemble, and execute MIPS assembly code, making it an excellent tool for learning and development.

2. Task-1: User defined Input-Output

(a) Code Explanation:

Input Collection

```
    li $v0, 4
    la $a0, prompt1
    syscall

    li $v0, 5
    syscall
    move $t0, $v0

    li $v0, 4
    la $a0, prompt2
    syscall

    li $v0, 5
    syscall
    move $t1, $v0
```

- **Load Immediate (li)**: This instruction loads an immediate value into a register.
- The value `4` is loaded into register `$v0`. In the MIPS syscall convention, setting `$v0` to `4` prepares the system for a "Print String" syscall, which will print a string to the console.
- **Load Address (la)**: This instruction loads the memory address of a label (in this case, `prompt1`) into a register.
- The address of the string `prompt1,prompt2` (which should be defined in the `.data` section) is loaded into register `$a0`. This string is what will be printed to the console.
- **Syscall**: This instruction triggers the system call specified by the value in `$v0`.
- `4`- Print String, `5`- Read Integer

- **Move**: This instruction copies the value from one register to another.

Performing Addition

```
    add $t2, $t0, $t1
    li $v0, 4
    la $a0, result_add
    syscall

    li $v0, 1
    move $a0, $t2
    syscall
```

- Syscall 1 - Print Integer
- **Add (add)**: This instruction adds the values of two registers and stores the result in a third register.$t2 = $t0 + $t1.
- Rest other calls are the same as explained above.
- **Syscall**: Executes the system call specified by the value in $v0.
- **Purpose**: The syscall uses the value in $v0 (1) to execute the "Print Integer" operation, printing the integer value stored in $a0 (which is the result of the addition) to the console.

Performing Multiplication

```
    mul $t2, $t0, $t1
    li $v0, 4
    la $a0, result_mul
    syscall

    li $v0, 1
    move $a0, $t2
    syscall
```

- All similar to addition except:mul
- **Multiply (mul)**: This instruction multiplies the values of two registers and stores the result in a third register.$t2 = $t0*$t1

Perform Subtraction

```
    sub $t2, $t0, $t1
    li $v0, 4
    la $a0, result_sub
    syscall
```

```
li $v0, 1
move $a0, $t2
syscall
```

- All similar to addition except:sub
- **Subtraction (msub)**: This instruction subtracts the values of two registers and stores the result in a third register.$t2 = $t0-$t1

```
li $v0, 10
syscall
```

- The value 10 is loaded into register $v0. In the MIPS syscall convention, setting $v0 to 10 indicates the "Exit" system call. This syscall tells the MIPS simulator to terminate the program.

3. Task-2: Payroll Calculation

(a) Code Explanation

Input Collection

```
li $v0, 4
la $a0, prompt_hours
syscall

li $v0, 5
syscall
move $t0, $v0
```

- **Load Immediate (li)**: This instruction loads an immediate (constant) value into a register.
- **Purpose**: The value 4 is loaded into register $v0. This value specifies the "Print String" syscall in MIPS, which is used to print a string to the console.
- **Load Address (la)**: This instruction loads the address of a data label into a register.
- **Purpose**: The address of the string prompt_hours (which should be defined in the .data section) is loaded into register $a0. This string will be the prompt message that is printed to the user, asking them to input the number of regular hours worked.
- **Syscall**: This instruction triggers the system call specified by the value in $v0.
- 4- Print String, 5- Read Integer
- Similary the code is for `prompt_overtime`, `prompt_wage`.

Gross Salary Calculation

```
    mul $t3, $t0, $t2           # Regular salary = regular hours * wage

    mul $t4, $t1, $t2           # Overtime pay = overtime hours * wage
    mul $t4, $t4, 3             # Multiply by 1.5 (using 3/2 approach)
    srl $t4, $t4, 1             # Divide by 2 (logical shift right by 1)

    add $t5, $t3, $t4           # Gross salary =regular salary+overtime pay
```

- $t3 will store the regular salary as commented.
- $t4 will store overtime pay as commented.
- 3 is used for: **Purpose**: Since overtime pay is calculated at 1.5 times the regular wage, this step effectively multiplies the overtime pay by 3 to apply the 1.5 multiplier (1.5 = 3/2). This step prepares the result for the next division.
- **Shift Right Logical (srl)**: This instruction performs a logical right shift on the value in a register.
- **Purpose**: The value in $t4 is **right-shifted** by 1 bit (essentially dividing the value by 2). This finalizes the calculation for the overtime pay, converting the previously scaled result into the correct value for 1.5 times the wage.
- $t5 will store the gross salary.

Deductions Calculation

```
    # tax rate = (24 mod 30) = 24%
    li $t6, 24                  # Load 24 (tax rate)
    mul $t6, $t6, $t5           # Calculate 24% of gross salary
    div $t6, $t6, 100           # Divide by 100

    # Insurance deduction (2% of gross salary) (24 + 8) mod 30 = 2
    li $t7, 2                   # Load 2 (insurance rate)
    mul $t7, $t7, $t5           # Calculate 2% of gross salary
    div $t7, $t7, 100           # Divide by 100
```

- $t6 will store tax deductions.
- $t6 = $t6*24. 24 is for tax rate and then we will divide the whole expression by 100 to convert it into percentage tax deduction.
- Similarly with insurance deduction at a rate of 2%.
- $t7 will store Insurance deductions.

Net Salary Calculation and Output

```
    add $t8, $t6, $t7           # Total deductions = tax + insurance

    # Net Salary Calculation
    sub $t9, $t5, $t8           # Net salary=gross salary - net deductions

    # Output Results
    # Print Gross Salary
    li $v0, 4                  # Print string syscall
    la $a0, result_gross        # Load address of result_gross
    syscall                     # Print "Gross Salary: "

    li $v0, 1                  # Print integer syscall
    move $a0, $t5               # Move gross salary to $a0
    syscall                     # Print the gross salary

    # Print Total Deductions
    li $v0, 4                  # Print string syscall
    la $a0, result_deductions   # Load address of result_deductions
    syscall                     # Print "Total Deductions: "

    li $v0, 1                  # Print integer syscall
    move $a0, $t8               # Move total deductions to $a0
    syscall                     # Print the total deductions

    # Print Net Salary
    li $v0, 4                  # Print string syscall
    la $a0, result_net          # Load address of result_net
    syscall                     # Print "Net Salary: "

    li $v0, 1                  # Print integer syscall
    move $a0, $t9               # Move net salary to $a0
    syscall
```

- `$t8` will store Total deduction that will be done from net salary
- `$t9` will store the net **In-hand** salary of an employee.
- Syscall `1` - Printing any integer, `4` - Printing the string given in `.text` which is currently being stored in the register.
- **Move**: This instruction copies the value from one register to another.

Loop for Multiple Employees

```
    li $v0, 4                   # Print string syscall
    la $a0, prompt_continue      # Load address of prompt_continue
    syscall                      # Print the prompt

    li $v0, 5                   # Read integer syscall
    syscall                      # Read user's response
    beq $v0, $zero, exit         # If 0, exit the program
    j main                       # If 1, process another employee

exit:
    li $v0, 10                  # Exit syscall
    syscall                      # Exit the program
```

- **`beq $v0, $zero, exit`:**
    1. **Branch on Equal (beq)**: This instruction checks if the values in two registers are equal and, if so, branches to a specified label.
    2. **Purpose**: This instruction checks if the integer value read (stored in `$v0`) is equal to `0`. If it is, the program branches to the `exit` label, which will terminate the program.
- `j main` :
    1. **Jump (j)**: This instruction unconditionally jumps to the specified label.
    2. **Purpose**: If the value read from the user is not `0`, this instruction jumps back to the `main` label, which restarts the main loop to process another employee's details.
- `exit:`
    1. **Label (exit)**: This label marks the location in the code where the program should jump to if the user chooses to exit.
    2. **Purpose**: This is where the program will jump to if the user input was `0`.
- Syscall `10` will terminate the program.

(b) Challenges

- One challenge was accurately calculating the overtime pay as 1.5 times the hourly wage.
- I overcame this by using a shift operation to multiply by 1.5 efficiently
- Another challenge was managing the many variables that needed to be stored in registers posed some difficulty. I overcame this by creating a table to keep track of the variables effectively.

Register	Value stored	Formula
$t0	Regular Hours	–
$t1	Overtime Hours	–
$t2	Hourly wage	–
$t3	Regular pay	Regular hour*hourly wage
$t4	Overtime pay	Overtime hours*wage*(1.5)
$t5	Gross Salary	Regular + overtime pay
$t6	Tax deduction	Gross salary* (0.24)
$t7	Insurance deduction	Gross salary*(0.02)
$t8	Total deduction	Tax + Insurance
$t9	Net Salary	Gross - Total deductions

(c) Summarized program output

- The program successfully calculates and displays the gross salary, total deductions, and net salary for each employee. The loop functionality allows the user to enter details for multiple employees sequentially.
- Below is a sample output for an employee working 40 regular hours, 5 overtime hours, and an hourly wage of 15:

```
Gross Salary: 675
Total Deductions: 162
Net Salary: 513
```

(d) Relevance with real-world scenarios

- The MIPS assembly language program effectively calculates employee payroll and offers a practical example of how low-level programming can be applied to real-world tasks. The experience with MARS and MIPS has enhanced my understanding of assembly language programming and its potential applications.
