Computer Architecture CSF342
Lab sheet 1

Topic: Introduction to MARS simulator and basic MIPS programming

---

Computer Architecture Lab: Introduction to MARS (MIPS Assembler and Runtime Simulator)

Lab 1: Basic MIPS Assembly Programming

---

1. Introduction to MARS

MARS is a lightweight simulator for the MIPS assembly language. It allows you to:

- Write, assemble, and run MIPS code.
- Debug programs step-by-step.
- Inspect registers, memory, and system resources.

Know more about MARS from here: https://dpetersanderson.github.io/

---

2. Starting MARS in Ubuntu

1. **Prerequisites**: Ensure Java is installed:

```
sudo apt install openjdk-17-jre
```

2. **Run MARS**: Open a terminal and execute:

```
java -jar Mars4_5.jar   # Replace with your MARS .jar filename
```

   The MARS GUI will launch.

---

3. Basic Operations

- **Create a New File**: Click `File > New` to open an editor.
- **Write Code**: Type MIPS assembly code (save as `*.s`).
- **Assemble**: Click  (or press `F3`).
- **Execute**: Click  (or press `F5`).

Key Settings:

- **"Assemble all"**: Combines `.text` (code) and `.data` (variables) segments.
- **"Initialize Program Counter"**: Sets starting execution point (default: `0x00400000`).

---

4. Memory Segments

- **Text Segment**: Stores your code (instructions).
- **Data Segment**: Stores global variables and strings.
    - Example:

```
.data
msg: .asciiz "Hello World!"   # Declares a string in data segment
```

---

5. Execution Modes

- **Run All**: Executes the entire program at once (use ).
- **Step-by-Step**: Debug line-by-line (use ).
    - Observe changes in registers after each instruction.

---

6. Side Panels

- **Registers Panel**: Shows values of all 32 MIPS registers (`$t0`, `$s0`, etc.).
- **Coprocessor 1 (FPU)**: Handles floating-point operations.
- **Memory Panel**: Displays content of `data` and `text` segments.

---

7. I/O via Syscalls

Use `syscall` to interact with the console:

1. Load service code into `$v0`.
2. Set arguments (e.g., `$a0` = address of string).
3. Execute `syscall`.

Service	**`$v0 - control register`**	**Arguments**
Print integer	1	`$a0` = integer
Print string	4	`$a0` = address of string
Read integer	5	Input saved to `$v0`
Exit	10	Terminates program

---

8. Example 1: Hello World

```
.data
msg: .asciiz "Hello World!"   # String declaration

.text
main:
    li $v0, 4          # Service 4: print string
    la $a0, msg         # Load address of 'msg'
    syscall             # Execute
    li $v0, 10         # Service 10: exit
    syscall
```

To Run: Assemble → Execute. Output appears in the "Run I/O" console.

---

9. Example 2: Hardcoded Adder

```
.text
main:
    li $s0, 5          # Load 5 into $s0
    li $s1, 3          # Load 3 into $s1
    add $t0, $s0, $s1  # $t0 = $s0 + $s1

    # Print result
    li $v0, 1          # Service 1: print integer
    move $a0, $t0      # Copy $t0 to $a0
    syscall

    li $v0, 10         # Exit
    syscall
```

---

10. Lab Task: User-Input Adder

Modify the adder to:

1. Prompt the user for two integers.
2. Read the integers.
3. Compute and print their sum.

Hint: Use `syscall` services 4 (print string), 5 (read integer), and 1 (print integer).

```
.data
msg1 : .asciiz "Enter first number : "
msg2 : .asciiz "Enter second number : "
msg3 : .asciiz "sum : "

.text
main:
   li $v0, 4
   la $a0, msg1
```

```
syscall

li $v0, 5
syscall
move $s0, $v0

li $v0,4
la $a0, msg2
syscall

li $v0, 5
syscall

move $s1, $v0

add $t0, $s0, $s1

li $v0, 4
la $a0, msg3
syscall

li $v0,1
move $a0, $t0
syscall

li $v0, 10
syscall
```

Example Workflow:

```
Enter first number: 5
Enter second number: 3
Sum = 8
```

Appendix: MARS Settings and Help Manual

A.1 Key MARS Settings (Bulleted Overview)

- **Assemble all**: Combines `.text` (code) and `.data` (variables) segments into one executable.
- **Initialize Program Counter to global 'main' if defined**: Sets start address to `main` label (else defaults to `0x00400000`).
- **Permit extended (pseudo) instructions**: Enables simplified instructions (e.g., `move`, `li`) that translate to core MIPS commands.

- **Delayed branching**: Simulates MIPS pipeline behavior (disable for simplicity in beginners' labs).
- **Self-modifying code**: Allows code to alter itself during runtime (advanced; keep disabled).
- **Highlight execution path**: Colors the next instruction to execute during step-by-step debugging.
- **Popup dialog for input syscalls**: Opens a separate window for user input (disable to use the console).

A.2 Help Manual (F1) Overview

The built-in help manual (`Help → Help` or `F1`) provides:

- **Basic Instructions**:
  Core MIPS commands (e.g., `add`, `lw`, `beq`) with syntax and usage examples.
- **Pseudo-Instructions**:
  Simplified commands translated to core instructions (e.g., `move $t0, $t1` → `add $t0, $t1, $zero`).
- **Directives**:
  Assembly-time controls (e.g., `.data`, `.asciiz`, `.word`) for data allocation and program structure.
- **Syscalls**:
  Complete list of I/O services (e.g., `$v0=1` prints integer; `$v0=8` reads string) with argument requirements.
- **Troubleshooting**:
  Common errors (e.g., alignment issues, undefined labels) and debugging tips.

  Tip: Use `F1` as a quick reference during coding!

Early habits to Adopt (as suggested by deepseek) will be needed in future.

Pro Tips for MIPS Assembly Programming

Adopt these habits from Day 1 to write cleaner, more efficient, and debuggable code:

1. Comment Religiously

```
add $t0, $s1, $s2   # $t0 = current_score + bonus (avoid "adds s1 and s2")
```

- **Why**: Assembly is opaque. Explain the **purpose** (e.g., "calculating total score"), not just the operation.

---

2. Use Labels, Not Magic Numbers

❌ `li $v0, 4`
✅ `li $v0, 4       # syscall: print_string`

- **Better**: Define constants:

```
.eqv PRINT_INT 1
.eqv EXIT 10
```

Then use: `li $v0, PRINT_INT`

---

3. Stick to Register Conventions

Register	Purpose
`$t0-$t9`	**Temporaries** (caller-saved)
`$s0-$s7`	**Saved values** (callee-saved)
`$a0-$a3`	**Arguments**
`$v0-$v1`	**Return values/syscalls**

- **Why**: Prevents bugs in larger programs and nested calls.

---

4. Always Exit Cleanly

```
end_program:
  li $v0, 10   # syscall: exit
  syscall
```

- **Why**: Prevents "falling off" into undefined memory.

---

5. Test Incrementally

After writing 3-5 lines:

1. **Assemble** (check for syntax errors).
2. **Step-through** (verify register values).
- **Tip**: Set breakpoints early.

---

6. Isolate Subroutines

```
# Good structure:
```

```
.text
main:
  ...
  jal get_input   # Jump to subroutine
  ...

get_input:        # Subroutine
  ...
  jr $ra          # Return
```

---

7. Initialize Registers

❌ Assuming `$s0` is 0 at startup.
✅ Explicitly set:

```
start:
  li $s0, 0      # Initialize counter
```

- **Why**: Simulators may not zero registers.

---

8. Use MARS Features

- **Auto-complete**: Press `Ctrl+Space` for directives/syscalls.
- **Debugger**: Set breakpoints (click left margin) to pause at critical points.

---

9. Avoid Pseudo-Instructions Early

❌ `move $t0, $t1` (pseudo-instruction)
✅ `add $t0, $t1, $zero` (core instruction)

- **Why**: Deepens understanding of MIPS internals.

---

10. Sanitize User Input

After `syscall` reads:

```
# Example: Check if input is negative
  blt $v0, $zero, invalid_input
```

- **Why**: Prevents overflow/logic errors.

Golden Rule: If it feels hacky, it probably is. Rewrite.

Adopting these habits early saves hours of debugging and builds foundational skills for advanced labs! 🚀