Assignment 5

CSL3020: Computer Architecture

Vaibhav Gupta (B22CS058)
Akarsh Katiyar (B22CS006)

Report on MIPS Processor Operation

Table of Contents

---

1. Introduction

The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture is a widely studied instruction set architecture (ISA) used in computer engineering and computer science. Understanding how MIPS processes instructions is essential for students and professionals working with computer architecture. This report provides a detailed overview of the MIPS instruction processing cycle, focusing on the entire journey from high-level code to machine code, and how these instructions are executed within a MIPS processor.

---

2. Program Compilation

Before a program can be executed on a MIPS processor, it must first be compiled from a high-level programming language to machine code that the processor can understand.

2.1. High-Level Code to Assembly

Source Code: The process begins with source code written in a high-level language (e.g., C, C++). For instance, consider the following C program:

```
int main() {

    int a = 10, b = 20;

    int c = a + b;

}
```

Compilation: The source code is compiled into assembly language by a compiler (like GCC). The assembly equivalent of the above program in MIPS might look like this:

```
addi $t0, $zero, 10   # Load immediate value 10 into register $t0

addi $t1, $zero, 20   # Load immediate value 20 into register $t1

add  $t2, $t0, $t1    # Add the values in $t0 and $t1, store result in
$t2
```

2.2. Assembly to Machine Code

1. **Machine Code Generation**: The assembler translates each assembly instruction into a corresponding machine code instruction, which consists of binary representations.
    - For example:
        - `addi $t0, $zero, 10`: `001000 00000 01000 0000000000001010` (opcode for `addi`, source register, destination register, immediate value)
        - `add $t2, $t0, $t1`: `000000 01000 01001 01010 00000 100000` (R-type format with opcode, source, and destination registers)

2.3. Machine Code Storage

- **Executable Storage**: The generated binary machine code is stored in memory, typically as a sequence of 32-bit words. This memory will be accessed by the MIPS processor during the instruction fetch stage.

---

3. Instruction Execution Process

Once the program is compiled into machine code, the MIPS processor executes these instructions through a well-defined sequence of operations.

3.1. Processor Stages

The MIPS processor operates in a multi-cycle manner, breaking down the execution of each instruction into multiple stages:

1. **Instruction Fetch (IF)**: The processor fetches the next instruction from memory using the Program Counter (PC) to identify the address of the instruction. The fetched instruction is loaded into the Instruction Register (IR).
2. **Instruction Decode (ID)**: The binary instruction in the IR is decoded by the Control Unit. During this stage:
    - The opcode is extracted to determine the instruction type (R-type, I-type, or J-type).
    - Source register addresses are identified, and any immediate values are extracted.
3. **Execution (EX)**: Based on the decoded instruction:
    - The Arithmetic Logic Unit (ALU) performs the specified operation (e.g., addition, subtraction).
    - For arithmetic operations, the ALU will take inputs from the specified registers.
4. **Memory Access (MEM)**: If the instruction involves accessing memory (such as `lw` or `sw`):
    - The processor reads from or writes to memory based on the address calculated during the execution stage.
5. **Write-Back (WB)**: The result of the ALU operation or the data fetched from memory is written back to the appropriate register, allowing it to be used in subsequent instructions.

---

4. Control Signal Generation

Control signals are crucial for directing the operation of different components within the MIPS processor. The Control Unit generates these signals based on the instruction being executed.

4.1. Instruction Format Breakdown

Each instruction in MIPS follows a specific format, influencing how control signals are generated:

R-Type Format:

```
opcode |  rs  |  rt  |  rd  | shamt | funct
```

```
000000 | 01000| 01001| 01010| 00000 | 100000
```

I-Type Format:

```
opcode | rs   | rt   | immediate

001000 | 01000| 01001| 0000000000010100
```

J-Type Format:

```
opcode | address

000010 | 00000000000000000000100100
```

4.2. Control Signals

Control signals are generated based on the opcode and the instruction format:

1. **R-Type Control Signals**:
    - **RegDst**: Set to `1` (select `rd` as the destination register).
    - **ALUSrc**: Set to `0` (ALU input is from registers).
    - **MemToReg**: Set to `0` (the result comes from the ALU).
    - **RegWrite**: Set to `1` (the register file is updated).
    - **MemRead**: Set to `0` (no memory access).
    - **MemWrite**: Set to `0` (no memory write).
    - **Branch**: Set to `0` (not a branch instruction).
    - **ALUOp**: Set based on operation (e.g., addition).
2. **I-Type Control Signals**:
    - **ALUSrc**: Set to `1` (ALU operand is the immediate value).
    - **MemRead**: Set to `1` for `lw` (read from memory).
    - **MemWrite**: Set to `1` for `sw` (write to memory).
    - **Branch**: Set to `1` for branch instructions.
3. **J-Type Control Signals**:
    - **Jump**: Set to `1` (indicates a jump instruction).
    - No register or memory control signals are needed for jumps.

---

5. ALU Control and Operations

The ALU Control Unit interprets the signals generated by the Control Unit to determine which operation the ALU should perform. This involves:

- Mapping `ALUOp` signals to specific operations based on instruction type.
- Using the `funct` field for R-type instructions to specify the exact ALU operation (e.g., addition, subtraction).

For instance:

- `funct 100000` corresponds to `ADD`.
- `funct 100010` corresponds to `SUB`.
- `funct 101010` corresponds to `SLT`.

6.Explanation of Core Functions

1. `int execute()` Function:

- **Function Parameters:**
  - `opcode (const string&)`: A binary string representing the operation code of the instruction (e.g., "001000" for `addi`).
  - `rsVal (int)`: Value of the first source register (`rs`), used for both R-type and I-type instructions.
  - `rtVal (int)`: Value of the second source register (`rt`), mainly used in R-type instructions.
  - `immediate (int)`: Immediate value used for I-type instructions (e.g., in `addi`).
  - `ALUOp (bool*)`: Pointer to the ALU control signal array, defining the operation the ALU should perform.
  - `control (ControlSignals)`: A structure containing control signals like `ALUSrc` (determines instruction type) and `ALUControl` (operation specifics).
- **Function Logic:**
  - **I-Type Instruction (e.g., `addi`, `lw`, `sw`):**
    - When `control.ALUSrc == 1`, it identifies I-type instructions and uses the immediate value in the computation.
    - If the instruction is `addi`, it performs `rsVal + immediate` and stores the result in `ALUResult`.

        - For other I-type instructions, it invokes the `ALU()` function with `rsVal` and `immediate`.
    - **R-Type Instruction (e.g., `add`, `sub`):**
        - When `control.ALUSrc == 0`, the function recognizes R-type instructions that use both `rsVal` and `rtVal`.
        - The `ALU()` function is invoked with `rsVal` and `rtVal`, performing operations like addition or subtraction.
    - **Debugging:**
        - Debug statements print the result of the ALU operation for both I-type and R-type instructions.
- **Return Value:**
    - Returns `ALUResult`, which contains the result of the ALU operation.
- **Control Signals Role:**
    - `ALUSrc`: Determines whether the operation uses an immediate value (I-type) or register values (R-type).
    - `ALUControl`: Specifies the exact operation for the ALU, such as addition or subtraction.

---

2. `compileRType(std::vector<std::string> &tokens)` Function:

- **Input:**
    - Vector `tokens` representing an R-type instruction (e.g., `add $rd, $rs, $rt`).
        - `tokens[0]`: Operation (e.g., `add`).
        - `tokens[1]`: Destination register (`rd`).
        - `tokens[2]`: First source register (`rs`).
        - `tokens[3]`: Second source register (`rt`).
- **Working:**
    - Extracts:
        - Opcode (`op`) from `opcodeMap` (for R-type instructions, this is usually `000000`).
        - Register values (`rs`, `rt`, `rd`) from `registerMap`.
        - Function code (`funct`) from `functMap`.
    - Constructs the final binary instruction by concatenating:
        - Opcode (`op`), source registers (`rs`, `rt`), destination register (`rd`), shift amount (`shamt`), and function code (`funct`).
- **Return:**
    - A string representing the complete 32-bit machine code for the R-type instruction.

---

3. `compileIType(std::vector<std::string> &tokens)` Function:

- **Input:**
  - Vector `tokens` representing an I-type instruction (e.g., `addi $rt, $rs, imm` or `lw $rt, offset($rs)`).
    - `tokens[0]`: Operation (e.g., `addi`, `lw`, `sw`).
    - `tokens[1]`: Destination register (`rt`).
    - `tokens[2]`: Source register (`rs`) or memory offset.
    - `tokens[3]`: Immediate value (for arithmetic or load/store instructions).
- **Working:**
  - **`li` (Load Immediate):**
    - Treated as `addi $reg, $zero, imm`.
  - **`addi` (Add Immediate):**
    - Computes `rsVal + immediate` and stores the result in `rt`.
  - **`lw`/`sw` (Load/Store Word):**
    - Parses the offset and base register from the format `offset(base)` or retrieves the offset from `dataMap`.
    - Constructs the instruction by assembling the opcode, registers, and immediate offset.
- **Return:**
  - A string representing the final binary instruction for the I-type operation, including the opcode, source register (`rs`), destination register (`rt`), and 16-bit immediate value.

---

4. `compileJType(std::vector<std::string> &tokens)` Function:

- **Input:**
  - Vector `tokens` representing a J-type instruction (e.g., `j label`).
    - `tokens[0]`: Operation (`j` for jump).
    - `tokens[1]`: Label or address to jump to.
- **Working:**
  - Retrieves the opcode from `opcodeMap`.
  - Converts the target label's address (from `labelling`) to a 26-bit binary value.
- **Return:**
  - A binary string composed of the opcode and the 26-bit jump address, forming the J-type machine code.

Summary:

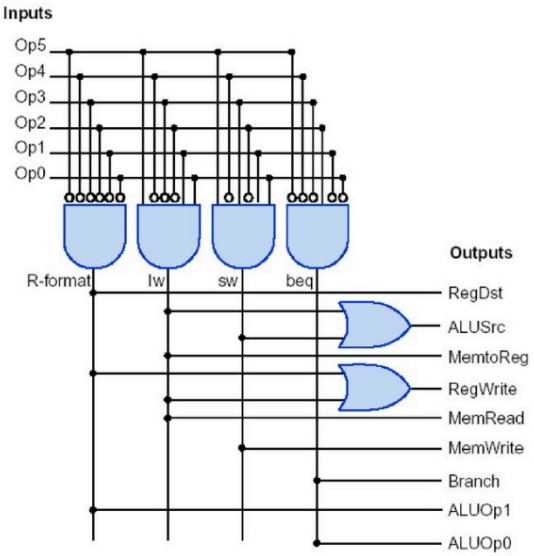
- **`compileRType`:** Converts R-type instructions by assembling the opcode, registers, shift amount, and function code into a 32-bit machine code.
- **`compileIType`:** Translates I-type instructions, including handling immediate values and memory access, into machine code.
- **`compileJType`:** Encodes J-type instructions by converting the jump address to its binary form, along with the opcode.

Each function leverages opcode maps, register maps, and specific instruction formats to output the correct binary sequence for machine execution.

5. `ALU()` Function:

The `ALU()` function simulates the Arithmetic Logic Unit (ALU) of a processor. It performs the core arithmetic and logical operations such as addition, subtraction, AND, OR, and set-less-than.

- **Parameters**:
  - `rsVal` and `rtVal`: These are the source operands used for operations.
  - `ALUOp`: Determines the specific operation to be executed (e.g., addition, subtraction).
  - `ALUControl`: This is the control signal that defines the specific ALU operation to execute.
- **Logic**: The function examines the `ALUControl` signal and performs operations such as:
  - **Addition** (`ALUControl == 010`): Performs `rsVal + rtVal`.
  - **Subtraction** (`ALUControl == 110`): Performs `rsVal - rtVal`.
  - **AND/OR** (`ALUControl == 000/001`): Performs bitwise AND or OR operations between `rsVal` and `rtVal`.
  - **Set Less Than** (`ALUControl == 111`): Checks whether `rsVal` is less than `rtVal` and sets the result accordingly.
- **Return Value**: The result of the arithmetic or logic operation, which can be used in the `execute()` function.



Generating Control Signals

6. `loadData()` Function:

This function loads data into the memory from a given file or an external source.

- **Purpose**:
  - It is primarily responsible for initializing the simulated memory with data values, such as loading instructions and data into memory before the processor begins execution.
- **Working**:
  - The function reads data from a file, parses it into individual memory addresses and values, and stores these values in the appropriate memory locations in the data segment.
  - It allows for simulation of memory loads, such as loading values used in instructions like `lw` (load word).
- **Return Value**: It returns a success/failure flag or status that indicates whether the data was loaded properly.

---

7. `decodeInstruction()` Function:

This function takes a binary instruction and decodes it into its constituent parts: opcode, registers, immediate values, etc.

- **Purpose**:
  - It dissects a 32-bit binary instruction into its relevant fields (opcode, rs, rt, rd, shamt, funct) to identify what type of operation should be performed.
  - The decoded fields are used to determine the type of instruction (R-type, I-type, or J-type) and its operands.

- **Working**:
  - The function extracts the binary fields of the instruction based on MIPS format. For example, it retrieves the opcode (first 6 bits), the registers (`rs`, `rt`, `rd`), and immediate values for I-type instructions.
  - It also identifies specific operation codes such as `add`, `sub`, `lw`, and `sw`.
- **Return Value**: A structure or tuple that contains the decoded fields, which are passed to other functions like `execute()`.

---

8. `fetchInstruction()` Function:

This function is responsible for fetching the next instruction to be executed from memory.

- **Purpose**:
  - It mimics the instruction fetch stage of a processor, where the Program Counter (PC) is used to retrieve the current instruction from the instruction memory.
  - This is the first step in the instruction execution cycle.
- **Working**:
  - The function accesses the instruction memory at the address indicated by the Program Counter (PC) and returns the instruction at that location.
  - After fetching the instruction, the PC is incremented to point to the next instruction in the sequence.
- **Return Value**: The 32-bit binary representation of the fetched instruction, which is passed to the `decodeInstruction()` function for further processing.

---

9. `updatePC()` Function:

This function updates the Program Counter (PC) based on the result of the current instruction.

- **Purpose**:
  - The Program Counter is a crucial element of the instruction flow. After each instruction, the PC needs to be updated to fetch the next instruction.
  - For branch and jump instructions, the PC might be updated with a target address; otherwise, it is simply incremented by 4 (to point to the next instruction).
- **Working**:
  - For non-branching instructions, it simply increments the PC by 4.
  - For branch instructions like `beq` or `j`, it calculates the target address and updates the PC accordingly.
- **Return Value**: The updated value of the PC, which is used in the next fetch stage.

---

10. `branchControl()` Function:

This function handles branch-related operations, such as checking branch conditions and updating the PC.

- **Purpose**:
  - For branch instructions like `beq` (branch if equal) or `bne` (branch if not equal), it checks whether the condition is met, and if so, updates the PC to the target address.
- **Working**:
  - It checks the values of the registers involved in the branch condition (e.g., `rs` and `rt`).
  - If the branch condition is true, it calculates the branch target address (relative offset) and updates the PC.
- **Return Value**: A boolean flag indicating whether the branch was taken or not, along with an updated PC if the branch was taken.

---

7. Execution of Specific Instructions

This section illustrates how the processor executes specific MIPS instructions.

- **Example 1: ADD Instruction**
  - **Instruction Fetch**: The processor fetches `add $t0, $t1, $t2`.
  - **Decode**: The opcode is `000000` (R-type), and the funct is `100000` (ADD).
  - **Control Signal Generation**:
    - RegDst = 1
    - ALUSrc = 0
    - RegWrite = 1
    - ALUOp = 10 (indicating an R-type operation)
  - **Execution**: The ALU adds the values in `$t1` and `$t2`, storing the result in `$t0`.
  - **Write-Back**: The result is written back to register `$t0`.

- **Example 2: LW Instruction**
  - **Instruction Fetch**: The processor fetches `lw $t0, 4($t1)`.
  - **Decode**: The opcode is `100011`, indicating a load instruction.
  - **Control Signal Generation**:

        - ALUSrc = 1 (indicates use of an immediate value)
        - MemRead = 1 (memory read operation)
        - RegWrite = 1 (write result to register)
    - **Execution**: The ALU calculates the effective address by adding `$t1` and `4`.
    - **Memory Access**: Data is read from the calculated address.
    - **Write-Back**: The data read from memory is written to register `$t0`.

---

8. Results and Simulation

This section illustrates the working of various results and working simulation of the code.

- ## Instruction 1: R-Type Instruction (ADD)
    - **Instruction Compilation**:

```
vaibhav@vaibhav-ka-PC:~/C_Arch$ g++ MIPS.cpp
vaibhav@vaibhav-ka-PC:~/C_Arch$ ./a.out
 .data

  .word 10

  .word 20

 .text

 lw $t0 num1
 Instruction code: 10001100000010000001001110001000

 lw $t1 num2
 Instruction code: 10001100000010010001001110001100

 add $t2 $t0 $t1
 Instruction code: 00000001000010010101000000100000

 sub $t3 $t1 $t0
 Instruction code: 00000001001010000101100000100010

 and $t4 $t0 $t1
 Instruction code: 00000001000010010110000000100100

 or  $t5 $t0 $t1
 Instruction code: 00000001000010010110100000100101

 slt $t6 $t0 $t1
 Instruction code: 00000001000010010111000000101010

 sw $t6 num1
 Instruction code: 10101100000011100001001110001000
```

- **Control Signals**:

```
PC:       12
Instruction:00000001000010010101000000100000

ALU operation with register values: rsVal + rtVal = 30
Control Signal Values:
--------------------------
Signal         Value
--------------------------
RegDst         1
ALUSrc         0
MemToReg       0
RegWrite       1
MemRead        0
MemWrite       0
Branch         0
Jump           0
ALUOp[2]       0 1
ALUControl     0
--------------------------
```

- **Registers**:

```
---------------------
Register  |Value
---------------------
Reg 0     | 0
Reg 1     | 0
Reg 2     | 0
Reg 3     | 0
Reg 4     | 0
Reg 5     | 0
Reg 6     | 0
Reg 7     | 0
Reg 8     | 10
Reg 9     | 0
Reg 10    | 0
Reg 11    | 0
Reg 12    | 0
Reg 13    | 0
Reg 14    | 0
```

    - **Result**: Here we can see that value 10 stored in num1 is loaded inside the memory location of Reg 8.

- **Example 2: I-Type Instruction (LW)**
    - **Instruction Compilation**:

```
vaibhav@vaibhav-ka-PC:~/C_Arch$ g++ MIPS.cpp
vaibhav@vaibhav-ka-PC:~/C_Arch$ ./a.out
.data

 .word 10

 .word 20

.text

lw $t0 num1
Instruction code: 10001100000010000001001110001000

lw $t1 num2
Instruction code: 10001100000010010001001110001100

add $t2 $t0 $t1
Instruction code: 00000001000010010101000000100000

sub $t3 $t1 $t0
Instruction code: 00000001001010000101100000100010

and $t4 $t0 $t1
Instruction code: 00000001000010010110000000100100

or  $t5 $t0 $t1
Instruction code: 00000001000010010110100000100101

slt $t6 $t0 $t1
Instruction code: 00000001000010010111000000101010

sw $t6 num1
Instruction code: 10101100000011100001001110001000
```

    - **Control Signals**:

```
PC:       4
Instruction:10001100000010000001001110001000

ALU operation with immediate: rsVal + immediate = 10
Control Signal Values:
--------------------------
Signal         Value
--------------------------
RegDst         0
ALUSrc         1
MemToReg       1
RegWrite       1
MemRead        1
MemWrite       0
Branch         0
Jump           0
ALUOp[2]       0 0
ALUControl     0
--------------------------
```

- **Registers**:

```
---------------------------
Register  |Value
---------------------------
Reg 0     | 0
Reg 1     | 0
Reg 2     | 0
Reg 3     | 0
Reg 4     | 0
Reg 5     | 0
Reg 6     | 0
Reg 7     | 0
Reg 8     | 10
Reg 9     | 20
Reg 10    | 30
Reg 11    | 0
Reg 12    | 0
Reg 13    | 0
Reg 14    | 0
Reg 15    | 0
```

- **Result**: Here we can see that values stored in `$t0` and `$t1` is added and stored inside the memory location of Reg 10 (`$t2`).

- **Example 3: J-Type Instruction (J)**

- **Instruction Compilation**:

```
j   end
Instruction code: 00001000000000000000000000001101
```

- **Control Signals**:

```
Control Signal Values:
--------------------------
Signal         Value
--------------------------
RegDst         0
ALUSrc         0
MemToReg       0
RegWrite       0
MemRead        0
MemWrite       0
Branch         0
Jump           1
ALUOp[2]       0 0
ALUControl     0
--------------------------
```

- **Next Instructions**:

```
PC:       48
Instruction:00100000000000100000000000000001

ALU operation with immediate: rsVal + immediate = 1
Control Signal Values:
--------------------------
Signal         Value
--------------------------
RegDst         0
ALUSrc         1
MemToReg       0
RegWrite       1
MemRead        0
MemWrite       0
Branch         0
```

- **Result**: Here we can see that the jump statement is executed and control went to the label.

- **Example 4: BEQ Instruction**

  - **Instruction Compilation**:

```
beq $t0 $t1 equal_case
Instruction code: 00010001000010010000000000000110
```

  - **Control Signals**:

```
PC:       16
Instruction:00010001000010010000000000000110

ALU operation with register values: rsVal + rtVal = 0
Control Signal Values:
--------------------------
Signal         Value
--------------------------
RegDst         0
ALUSrc         0
MemToReg       0
RegWrite       0
MemRead        0
MemWrite       0
Branch         1
Jump           0
ALUOp[2]       1 0
ALUControl     0
```

  - **Next Instruction**:

```
PC:       28
Instruction:00000001000010010101100000100000

ALU operation with register values: rsVal + rtVal = 10
Control Signal Values:
--------------------------
Signal         Value
--------------------------
RegDst         1
ALUSrc         0
MemToReg       0
RegWrite       1
MemRead        0
MemWrite       0
Branch         0
```

  - **Result**: Here we can see that branch is **taken** and then went on to the label.

---

9. Output Based on the Various codes Executed

This section illustrates the working of various results and working simulation of the code.

- **Code test_code_1_mips_sim.asm:**

  Instruction Compilation:

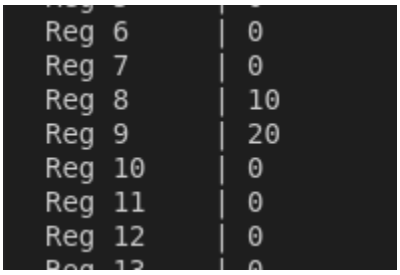
```
vaibhav@vaibhav-ka-PC:~/C_Arch$ g++ MIPS.cpp
vaibhav@vaibhav-ka-PC:~/C_Arch$ ./a.out
.data

 .word 10

 .word 20

.text

lw $t0 num1
Instruction code: 10001100000010000001001110001000

lw $t1 num2
Instruction code: 10001100000010010001001110001100

add $t2 $t0 $t1
Instruction code: 00000001000010010101000000100000

sub $t3 $t1 $t0
Instruction code: 00000001001010000101100000100010

and $t4 $t0 $t1
Instruction code: 00000001000010010110000000100100

or  $t5 $t0 $t1
Instruction code: 00000001000010010110100000100101

slt $t6 $t0 $t1
Instruction code: 00000001000010010111000000101010

sw $t6 num1
Instruction code: 10101100000011100001001110001000
```

  Outputs:

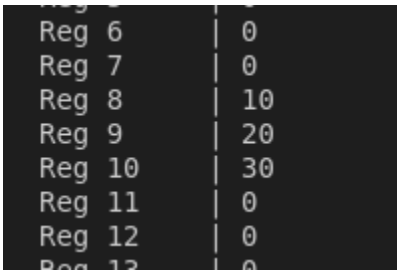
  First, the value of num1 is loaded inside Reg8 / $t0.

```
Reg 5     | 0
Reg 6     | 0
Reg 7     | 0
Reg 8     | 10
Reg 9     | 0
Reg 10    | 0
Reg 11    | 0
Reg 12    | 0
```

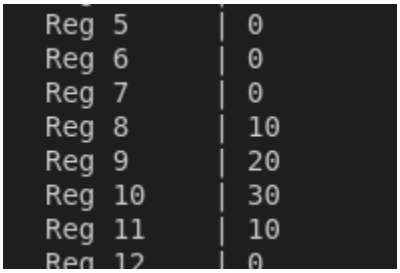
Next, the value of num2 is loaded inside Reg9 / $t1.



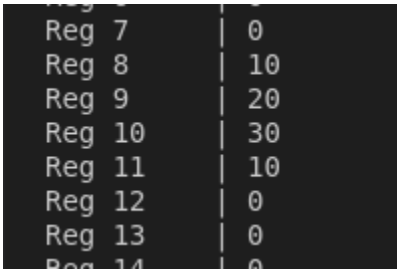
Then, add instruction is executed



After which, sub is preformed



Next, ‘and’ is performed, the output for which was



For ‘or’ instruction the output was

```
Reg 7      | 0
Reg 8      | 10
Reg 9      | 20
Reg 10     | 30
Reg 11     | 10
Reg 12     | 0
Reg 13     | 30
Reg 14     | 0
```

Next, slt is performed

```
Reg 7      | 0
Reg 8      | 10
Reg 9      | 20
Reg 10     | 30
Reg 11     | 10
Reg 12     | 0
Reg 13     | 30
Reg 14     | 10
```

Finally, the value of $t6 is stored inside num1

```
Store Word Instruction execute:
Stored 10 at  address 5000 in the Memory.
```

- **Code test_code_2_mips_sim.asm:**

Assembly Code:

test_code_2_mips_sim.asm

```
1   .data
2       val1: .word 5
3       val2: .word 5
4
5   .text
6       lw $t0, val1
7       lw $t1, val2
8       addi $t2, $t0, 10
9       beq $t0, $t1, equal_case
10      sub $t3, $t0, $t1
11      j end
12  equal_case:
13      add $t3, $t0, $t1
14
15  end:
```

Instruction Compilation:

```
lw $t0 val1
Instruction code: 10001100000010000001001110001000

lw $t1 val2
Instruction code: 10001100000010010001001110001100

addi $t2 $t0 10
Instruction code: 00100001000010100000000000001010

beq $t0 $t1 equal_case
Instruction code: 00010001000010010000000000000110

sub $t3 $t0 $t1
Instruction code: 00000001000010010101100000100010

j end
Instruction code: 00001000000000000000000000000111

add $t3 $t0 $t1
Instruction code: 00000001000010010101100000100000
```

Outputs:

Since, most basic instructions are already done so we would focus on the important portions of the program.

```
Register  |Value
---------------------------
Reg 0     | 0
Reg 1     | 0
Reg 2     | 0
Reg 3     | 0
Reg 4     | 0
Reg 5     | 0
Reg 6     | 0
Reg 7     | 0
Reg 8     | 5
Reg 9     | 5
Reg 10    | 0
```

Values loaded inside Reg8 and Reg9.

```
PC:       16
Instruction:10101100000010100001001110010000

ALU operation with immediate: rsVal + immediate = 5008

Store Word Instruction execute:
Stored 13 at  address 5008 in the Memory.

Control Signal Values:
---------------------------
```

Performed the Store Word instruction.

```
  Reg 4     | 0
  Reg 5     | 0
  Reg 6     | 0
  Reg 7     | 0
  Reg 8     | 12
  Reg 9     | 5
  Reg 10    | 13
  Reg 11    | 4
  Reg 12    | 0
  Reg 13    | 0
  Reg 14    | 0
```

And instruction preformed (12 and 5 = 4)

```
PC:       28
Instruction:00010001010010110000000000001000

ALU operation with register values: rsVal + rtVal = 9
Control Signal Values:
---------------------------
Signal         Value
---------------------------
RegDst         0
ALUSrc         0
MemToReg       0
RegWrite       0
MemRead        0
MemWrite       0
Branch         1
Jump           0
ALUOp[2]       1 0
ALUControl     0
---------------------------
Register  |Value
```

Branch not taken as ALU Result came out to be 9 != 0.

```
Reg 31    | 0
PC:       52
Instruction:00100000000001000000000000000001

ALU operation with immediate: rsVal + immediate = 1
Control Signal Values:
---------------------------
Signal         Value
---------------------------
RegDst         0
ALUSrc         1
MemToReg       0
RegWrite       1
MemRead        0
MemWrite       0
Branch         0
Jump           0
ALUOp[2]       0 0
ALUControl     0
---------------------------
Register  |Value
```

Jump statement executed and jumped on to the **not-equal** label

```
----------------------
Reg 0     | 0
Reg 1     | 0
Reg 2     | 1
Reg 3     | 0
Reg 4     | 0
Reg 5     | 0
```

Value 1 loaded inside Reg2 or $v0 and 0 loaded in Reg4 or $a0.

- **Code test_code_3_mips_sim.asm:**

Assembly Code:

test_code_3_mips_sim.asm

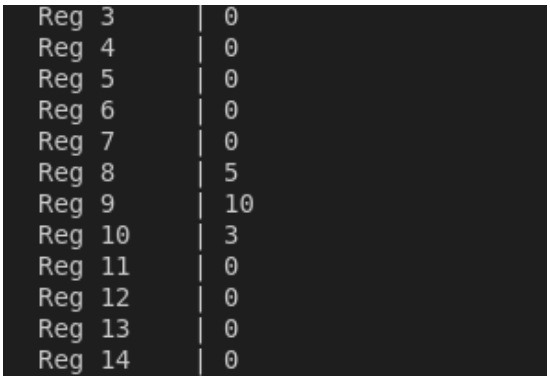
```
1   .data
2   number1:    .word 5
3   number2:    .word 10
4   number3:    .word 3
5   result:     .word 0
6
7   .text
8   main:
9       lw  $t0, number1
10      lw  $t1, number2
11      lw  $t2, number3
12
13      add $t3, $t0, $t1
14      sub $t4, $t3, $t2
15
16      sw  $t4, result
17
18      j   end
19
20  end:
21
```

Instruction Compilation:

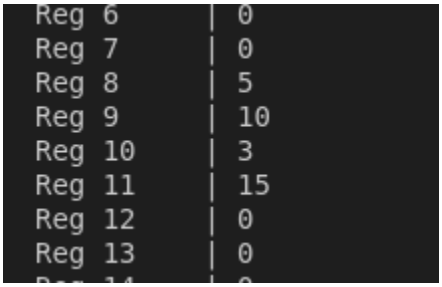
```
lw  $t0 number1
Instruction code: 10001100000010000001001110001000

lw  $t1 number2
Instruction code: 10001100000010010001001110001100

lw  $t2 number3
Instruction code: 10001100000010100001001110010000

add $t3 $t0 $t1
Instruction code: 00000001000010010101100000100000

sub $t4 $t3 $t2
Instruction code: 00000001011010100110000000100010

sw  $t4 result
Instruction code: 10101100000011000001001110010100

j   end
Instruction code: 00001000000000000000000000000111
```

Outputs:

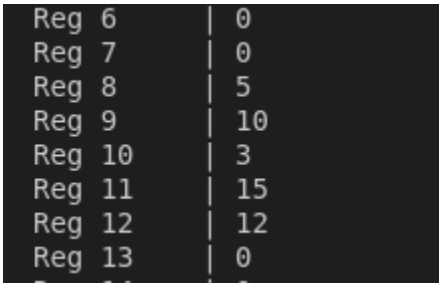
Since, most basic instructions are already done so we would focus on the important portions of the program.



Values loaded inside registers Reg8, Reg9 and Reg10.



Add instruction performed.



Sub instruction performed (15 - 3 = 12)

```
PC:       24
Instruction:10101100000011000001001110010100

ALU operation with immediate: rsVal + immediate = 5012

Store Word Instruction execute:
Stored 12 at  address 5012 in the Memory.
```

Store word performed

```
PC:       28
Instruction:00001000000000000000000000000111

Control Signal Values:
---------------------------
Signal         Value
---------------------------
RegDst         0
ALUSrc         0
MemToReg       0
RegWrite       0
MemRead        0
MemWrite       0
Branch         0
Jump           1
ALUOp[2]       0 0
ALUControl     0
---------------------------
```

Jumped to **end**

- **Code test_code_4_mips_sim.asm:**

Assembly Code:

test_code_4_mips_sim.asm

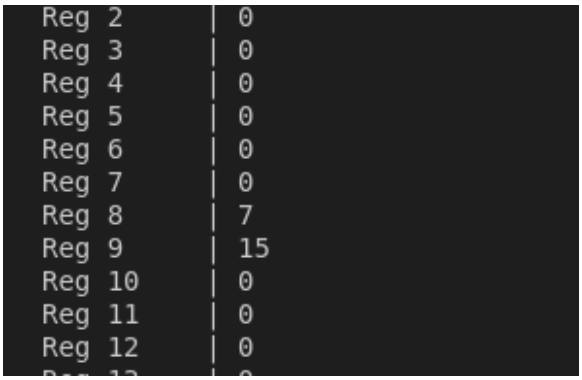
```
1   .data
2   num1:   .word 7
3   num2:   .word 15
4   result: .word 0
5
6   .text
7   main:
8       lw  $t0, num1
9       lw  $t1, num2
10
11      slt $t2, $t0, $t1
12
13      sw  $t2, result
14
15      j   end
16
17  end:
```

Instruction Compilation:

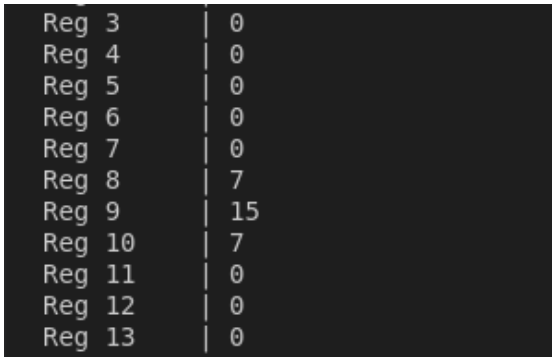
```
lw  $t0 num1
Instruction code: 10001100000010000001001110001000

lw  $t1 num2
Instruction code: 10001100000010010001001110001100

slt $t2 $t0 $t1
Instruction code: 00000001000010010101000000101010

sw  $t2 result
Instruction code: 10101100000010100001001110010000

j   end
Instruction code: 00001000000000000000000000000101
```

Outputs:

Since, most basic instructions are already done so we would focus on the important portions of the program.



Values loaded inside registers Reg8($t0) and Reg9($t1).



‘Slt’ performed and since 7 < 15, thus Reg10($t2) is assigned the value 7.

```
PC:       16
Instruction:10101100000010100001001110010000

ALU operation with immediate: rsVal + immediate = 5008

Store Word Instruction execute:
Stored 7 at  address 5008 in the Memory.

Control Signal Values:
---------------------------
```

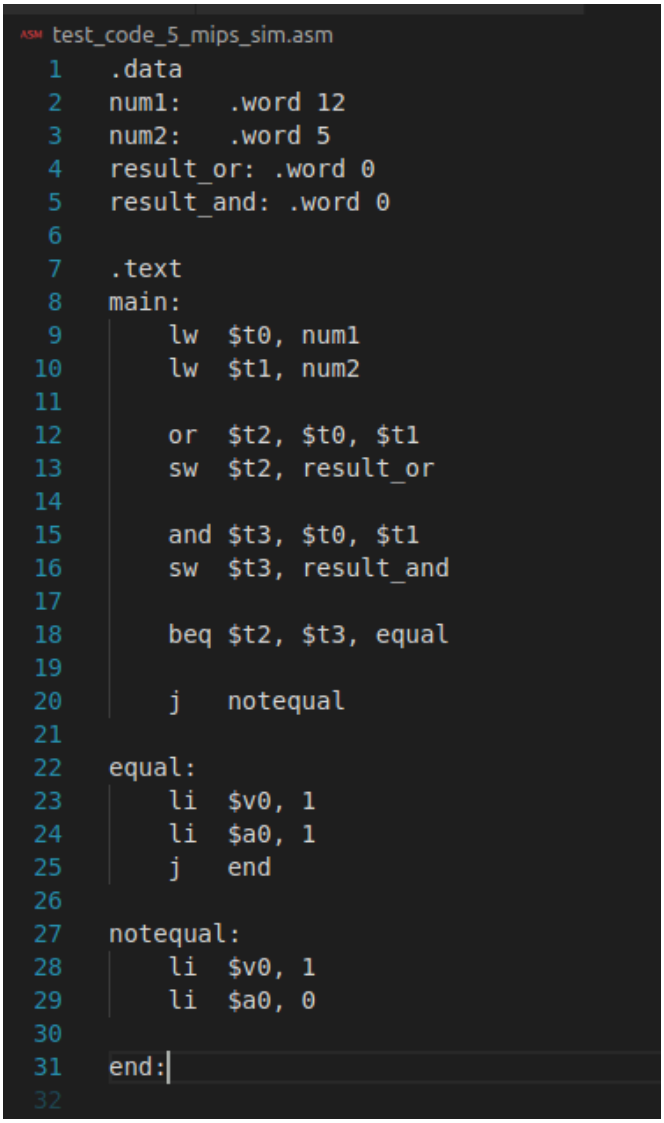
Store word performed and result is updated

```
PC:       20
Instruction:00001000000000000000000000000101

Control Signal Values:
---------------------------
Signal         Value
---------------------------
RegDst         0
ALUSrc         0
MemToReg       0
RegWrite       0
MemRead        0
MemWrite       0
Branch         0
Jump           1
ALUOp[2]       0 0
ALUControl     0
---------------------------
Register  |Value
---------------------------
```

Jumped to end

- **Code test_code_5_mips_sim.asm:**

  Assembly Code:



Instruction Compilation:

```
lw  $t0 num1
Instruction code: 10001100000010000001001110001000

lw  $t1 num2
Instruction code: 10001100000010010001001110001100

or  $t2 $t0 $t1
Instruction code: 00000001000010010101000000100101

sw  $t2 result_or
Instruction code: 10101100000010100001001110010000

and $t3 $t0 $t1
Instruction code: 00000001000010010101100000100100

sw  $t3 result_and
Instruction code: 10101100000010110001001110010100

beq $t2 $t3 equal
Instruction code: 00010001010010110000000000001000

j   notequal
Instruction code: 00001000000000000000000000001011

li  $v0 1
Instruction code: 00100000000001000000000000000001

li  $a0 1
Instruction code: 00100000000001000000000000000001

j   end
Instruction code: 00001000000000000000000000001101

li  $v0 1
Instruction code: 00100000000001000000000000000001

li  $a0 0
Instruction code: 00100000000001000000000000000000
```

Outputs:

Since, most basic instructions are already done so we would focus on the important portions of the program.

```
Reg 3     | 0
Reg 4     | 0
Reg 5     | 0
Reg 6     | 0
Reg 7     | 0
Reg 8     | 12
Reg 9     | 5
Reg 10    | 13
Reg 11    | 0
Reg 12    | 0
```

Values loaded inside various registers.

```
PC:       16
Instruction:10101100000010100001001110010000

ALU operation with immediate: rsVal + immediate = 5008

Store Word Instruction execute:
Stored 13 at  address 5008 in the Memory.

Control Signal Values:
--------------------------
```

Performed the Store Word instruction.

```
  Reg 4      | 0
  Reg 5      | 0
  Reg 6      | 0
  Reg 7      | 0
  Reg 8      | 12
  Reg 9      | 5
  Reg 10     | 13
  Reg 11     | 4
  Reg 12     | 0
  Reg 13     | 0
  Reg 14     | 0
```

And instruction preformed (12 and 5 = 4)

```
PC:       28
Instruction:00010001010010110000000000001000

ALU operation with register values: rsVal + rtVal = 9
Control Signal Values:
--------------------------
Signal         Value
--------------------------
RegDst         0
ALUSrc         0
MemToReg       0
RegWrite       0
MemRead        0
MemWrite       0
Branch         1
Jump           0
ALUOp[2]       1 0
ALUControl     0
--------------------------
Register  |Value
```

Branch not taken as ALU Result came out to be 9 != 0.

```
Reg 31    | 0
PC:       52
Instruction:00100000000000100000000000000001

ALU operation with immediate: rsVal + immediate = 1
Control Signal Values:
--------------------------
Signal         Value
--------------------------
RegDst         0
ALUSrc         1
MemToReg       0
RegWrite       1
MemRead        0
MemWrite       0
Branch         0
Jump           0
ALUOp[2]       0 0
ALUControl     0
--------------------------
Register  |Value
```

Jump statement executed and jumped on to the **not-equal** label

```
----------------------
Reg 0     | 0
Reg 1     | 0
Reg 2     | 1
Reg 3     | 0
Reg 4     | 0
Reg 5     | 0
```

Value 1 loaded inside Reg2 or $v0 and 0 loaded in Reg4 or $a0.

---

10. Conclusion

The MIPS instruction execution process involves a series of well-defined stages, from compiling high-level code into machine code to executing instructions within the processor. Understanding this process is essential for grasping the fundamental concepts of computer architecture, enabling students and engineers to develop optimized code and design efficient systems.

This report has provided a comprehensive overview of the MIPS architecture's instruction handling, including the role of control signals, ALU operations, and specific

instruction executions, facilitating a deeper understanding of how MIPS operates internally.

---

11. Contributions

- **`Akarsh Katiyar`**

  Worked on the compilation part of the code and the Report. Did Research on various opcodes and the format of various types of Instruction (R-Type, I-Type and J-Type) and implemented them to do the compilation of the code. Also trimmed the input code, removed commas etc.

  Also worked along-side Vaibhav for making the Report.

- **`Vaibhav Gupta`**

  Worked mainly on the processing part. Researched on the ways to generate the control signals and how to implement and use Registers, Data Memory and Instruction Memory. And thus performed the processing of R-Type, I-Type and J-Type instructions. Further worked along with Akarsh on Data Labeling and Instruction Labeling for ‘beq’ and ‘j’.

  Also worked along-side Akarsh for making the Report.
