Comprehensive MIPS Simulator Implementation Report

Introduction

This report provides a detailed analysis of the MIPS simulator implementation, covering both the compilation of MIPS assembly code into binary machine code and the subsequent execution simulation. The implementation successfully meets the requirements outlined in the assignment, demonstrating a robust understanding of MIPS architecture, instruction set, and execution pipeline.

The MIPS simulator comprises several components, including:

- Reading and parsing MIPS assembly code into a structured format that facilitates later processing.
- Converting parsed instructions into binary format for execution.
- Simulating the execution of binary instructions, which includes handling various instruction types and managing program control flow through branches and jumps.

Summary of the code

```
// Declare global variables for registers, memory, etc.
map<string, string> register_memory; // Register memory mapping
map<string, int> memoryAllocation; // Memory allocation for variables
map<string, int> labelAddressMap; // Label address mapping
map<int, string> instructionMemory; // Instruction memory
vector<int> atRegister; // Simulated memory for data storage

// Function prototypes for parsing and execution
void parseDataSection(ifstream &inputFile);
void parseTextSection(ifstream &inputFile);
void processInstruction(string instruction);
void executeRType(string funct, string rs, string rt, string rd);
void executeIType(string opcode, string rs, string rt, int immediate);
void executeJType(string opcode, string address);
void storeInRegister(const string &reg, const string &value);
void Processor(const string &filename);

// Main function to drive the simulation
int main(int argc, char *argv[]) {
    // Open input assembly file
    ifstream inputFile(argv[1]);

    // Parse sections
    parseDataSection(inputFile);
    parseTextSection(inputFile);

    Processor("output.bin");
```

In the following sections, we will delve into each of these components in detail, highlighting the data structures used, the parsing logic, instruction execution flow, and how the overall architecture of the simulator aligns with MIPS specifications.

Task 1: MIPS Compiler Implementation [50 points]

1.1 Reading Assembly Input

The first step in the MIPS compiler implementation involves reading MIPS assembly code from an input file. This is achieved using standard C++ file handling techniques, specifically with an `ifstream` object:

```
ifstream inputFile(argv[1]);
```

This approach allows the program to flexibly process different MIPS assembly files, enhancing its usability and versatility.

1.2 Parsing Instructions

The parsing process is divided into two primary sections: the **data section** and the **text section**. This separation is crucial for accurately interpreting and executing the assembly code.

1.2.1 Data Section Parsing

The `parseDataSection` function is responsible for handling the `.data` section of the MIPS assembly. It processes variable declarations and allocates memory for them, following these steps:

- A variable `dataMemoryAddress` is initialized to `0x10010000`, simulating the starting address for data memory in MIPS architecture.
- The function iterates through each line of the data section, identifying labels and directives, primarily supporting the `.word` directive at this stage.
- For each variable declaration, it stores the memory address and its initial value in the `memoryAllocation` map, effectively simulating memory allocation for these variables.
- The `varToRegMap` associates variables with registers, particularly mapping them to the `$at` (assembler temporary) register for easier manipulation during execution.

This function forms the backbone of memory allocation for variables, ensuring that the execution phase can accurately reference and manipulate these values.

1.2.2 Text Section Parsing

The `parseTextSection` function manages the `.text` section, which contains the executable MIPS instructions. It employs a two-pass algorithm for efficient instruction handling:

1. **First Pass**:
    - This pass captures labels and their corresponding addresses in the `labelAddressMap`, which is crucial for handling jumps and branches.
    - It specifically identifies the `main` label, marking the entry point of the program.
2. **Second Pass**:
    - During this phase, each instruction is parsed to determine its type (R, I, or J) and its components, including opcode, registers, and immediate values.
    - Each instruction is converted into its binary representation using a series of well-defined transformations.

1.3 Converting to Binary

The conversion from MIPS assembly to binary format is accomplished through three main functions, each tailored for a specific instruction type:

1. **`convertRType`**: Handles R-type instructions such as `add`, `sub`, `and`, `or`, and `slt`. It constructs the binary representation using bitwise operations to manipulate individual bits according to the MIPS instruction format.
2. **`convertIType`**: Handles I-type instructions like `addi`, `lw`, `sw`, and `beq`, following a similar approach to convert operands into their corresponding binary forms.
3. **`convertJType`**: Deals with J-type instructions, including `j` and `jal`, ensuring that the jump addresses are accurately represented in binary.

Each of these functions uses the `bitset` class for precise bit manipulation, ensuring that the final binary output adheres strictly to MIPS standards.

1.4 Output the Binary

Once the binary instructions are generated, they are written to a file named “output.bin”:

```
ofstream outputFile("output.bin");
```

This step is crucial as it creates a complete binary representation of the MIPS program, which can be later executed in the simulation phase.

Task 2: MIPS Execution Simulation [50 points]

2.1 Simulating the MIPS Datapath

The simulation of the MIPS datapath is implemented primarily within the `Processor` function. This function reads the binary instructions from the output file and simulates their execution in a systematic manner.

2.1.1 Instruction Fetch

Instructions are fetched from the `instructionMemory` map, which is populated during the initial reading of the binary file. The current instruction is accessed using the Program Counter (PC):

```
string currentInstruction = instructionMemory[PC];
```

This code simulates the instruction fetch stage of the MIPS pipeline, where the next instruction to execute is identified based on the current value of the PC.

2.1.2 Instruction Decode and Execute

The `processInstruction` function is responsible for both decoding and executing instructions. The execution process includes the following steps:

1. **Identify Instruction Type**: The function determines the type of instruction by examining the opcode.
2. **Call Appropriate Execution Function**: Based on the identified instruction type, the corresponding execution function is called:
    - `executeRType` for R-type instructions
    - `executeIType` for I-type instructions
    - `executeJType` for J-type instructions

Each of these functions decodes the specific fields of the instruction and performs the corresponding operation. This modular approach promotes code clarity and maintainability.

2.2 Generating and Applying Control Signals

While control signals are not explicitly modeled as separate entities, the flow of control is implicitly handled within the execution functions. For instance, in `executeRType`, different operations are performed based on the function code:

```
if (funct == "100000") {  // add
    result = value1 + value2;
} else if (funct == "100010") {  // sub
    result = value1 - value2;
}
// ... other operations
```

This segment of code effectively simulates the control signals that would dictate ALU operations in a physical MIPS processor, demonstrating the seamless integration of control flow within execution logic.

2.3 Simulating ALU Operations

Arithmetic and logical operations are simulated within the execution functions. For instance, in the `executeRType` function:

```
int value1 = stoi(register_memory[rs], nullptr, 2);
int value2 = stoi(register_memory[rt], nullptr, 2);
int result = 0;
// ... perform operation based on instruction
```

In this code, values are fetched from the specified registers using their binary representations, the appropriate arithmetic or logical operation is performed, and the result is calculated based on the instruction's specification.

2.4 Memory Access Simulation

Memory access is specifically handled for load and store operations within the `executeIType` function. For example, the load word (lw) instruction is processed as follows:

```
result = atRegister[immediate / 4];
storeInRegister(reverse_mips_registers[rt], bitset<32>(result).to_string());
```

This simulates loading a value from memory (represented by the `atRegister` vector) into a specified register. The careful indexing by dividing the immediate value by 4 accounts for word alignment in MIPS.

2.5 Handling Branching

Branching operations are implemented in the `executeIType` function for `beq` and `bne` instructions:

```
if (opcode == "000101") {  // bne
    int val = stoi(register_memory[rt], nullptr, 2);
    if (value != val) {
        PC += immediate * 4;  // Jump to the address specified by immediate
    } else {
        PC += 4;  // Continue to the next instruction
    }
}
```

This code snippet compares values stored in registers and updates the Program Counter (PC) accordingly, effectively simulating the conditional branching behavior characteristic of MIPS instructions.

2.6 Register and Memory Simulation

Registers are simulated using the `register_memory` map, which holds the values of all 32 MIPS registers. The `storeInRegister` function is utilized to update the values stored in the registers:

```
void storeInRegister(const string &reg, const string &value) {
    register_memory[mips_registers[reg]] = value;
}
```

This structure allows for direct access and manipulation of register values throughout execution, reflecting changes made by executed instructions.

2.7 Program Counter (PC) Simulation

The Program Counter is simulated using an integer variable `PC`. It is initialized to `0x00400000` (the conventional starting address for the MIPS text segment) and updated after each instruction:

```
PC += 4;  // Move to the next instruction for most cases
// or
PC = address;  // For jump instructions
```

This simulation accurately reflects the behavior of the PC in the MIPS architecture, enabling precise tracking of instruction flow.

Task 3: Testing and Reporting

3.1 Testing

The implementation incorporates a basic testing mechanism that allows users to execute the compiled code immediately after compilation:

```
string wantToExecute;
cout << "Want to execute\nTo execute type yes\n";
cin >> wantToExecute;
if(wantToExecute == "yes") {
    Processor("output.bin");
}
```

This interaction provides users with immediate feedback and the ability to verify the functionality of their compiled MIPS program, facilitating debugging and validation.

3.2 Output and Reporting

At the end of the execution, the final state of all registers is printed, providing a comprehensive view of the program's effect on the processor state:

```
for(const auto& ele: registerVals) {


    cout << "Register " << ele.first << ": " << ele.second << endl;
}
```

This feature enhances the usability of the simulator, enabling users to analyze the results of their MIPS programs and assess performance.

Conclusion

The MIPS simulator implementation effectively demonstrates the essential processes of compiling and executing MIPS assembly code while adhering to the architecture's specifications. The project begins with a robust parsing mechanism that divides the MIPS assembly code into data and text sections, allowing for accurate instruction and memory management. By implementing a two-pass algorithm for instruction parsing, the simulator efficiently captures labels and generates a precise binary representation of each instruction. This meticulous attention to detail in parsing ensures that the generated binary is both accurate and executable, forming the foundation for the subsequent execution phase.

During the execution phase, the simulator successfully replicates the MIPS datapath by incorporating instruction fetch, decode, and execution stages. The modular approach to handling different instruction types—R, I, and J—facilitates clear separation of concerns, enhancing code maintainability and readability. The ability to simulate ALU operations, memory access, and branching logic showcases a deep understanding of MIPS architecture and its operational characteristics. The use of a `Processor` function to manage the execution flow, combined with accurate updates to the Program Counter and register values, provides an authentic simulation of how a real MIPS processor would execute instructions.

In summary, the MIPS simulator project not only meets the assignment's requirements but also serves as an invaluable learning tool for understanding MIPS architecture and assembly language programming. The comprehensive approach taken in developing this simulator highlights key concepts in instruction parsing, binary translation, and execution simulation. This implementation sets a solid groundwork for further exploration into advanced features such as handling more complex instruction types, incorporating pipelining, and implementing additional control logic. The experience gained from this project will be instrumental in fostering a deeper appreciation for computer architecture and the intricacies of low-level programming.

Group Members

- Sri Ganesh Thota (B22CS054)
- Trivedi Aditya Bhargavkumar (B22CS055)
- Jateen (B22CS026)

References

- MIPS Architecture Reference Manual
- MIPS Instruction Set Overview
- C++ Programming Language Documentation