A Simple VM for MIPS

Harshavardhan Miryala
Taanya Nithya Anand
Bhavye Jain

1 Introduction

Virtual machines (VMs) are software-based emulations of physical computers that enable multiple operating systems and applications to run simultaneously on a single physical machine, providing isolation, flexibility, and efficient resource utilization. Through this project, we aim to develop a VM for the MIPS32 architecture.

The MIPS architecture has seen widespread adoption in industry and still remains relevant with applications in embedded systems, networks and IoT. Due to its simplicity, MIPS has been used in education for understanding techniques like pipelining and instruction decoding. The power efficiency, high performance, reliability and scalability of MIPS will drive its deployment for the foreseeable future and that motivates building a VM to simulate MIPS based programs.

Such a solution would enable legacy software to be tested and run once again, and would also aid in testing & debugging of new solutions before full-scale deployment. The inherent simplicity and scalability of MIPS also allows the VM to scale freely and flexibly to suit the requirements of a user.

2 Problem Statement

The aim of this project shall be to develop a virtualization solution for the MIPS 32-bit architecture that supports the execution of programs written in a subset of the C++ language defined as follows:

1. Data types: integer (signed), string
2. Operators: arithmetic, logical, relational
3. Iteration: loops, nested loops, recursion
4. Functions
5. Pointers and arrays
6. Console I/O

NOTE: We do NOT aim to provide support for C++ libraries, linking and system interrupts.

3 Background

MIPS (Microprocessor without Interlocked Pipeline Stages) is a RISC (Reduced Instruction Set Computer) architecture that emphasizes simplicity and efficiency in instruction

execution. All instructions are of a fixed 32-bit size and can be perceived as being divided into 3 categories: register (R) type, immediate (I) type and jump (J) type instructions, based on their bytecode format.

MIPS provides a set of 32 registers, each 32 bits in size which allows for efficient data manipulation. Memory operations can only be performed through load and store instructions. Data must be loaded from memory into registers, manipulated within registers, and then stored back into memory.

The memory model in MIPS architecture assumes a flat address space, where both instructions and data are stored. The architecture supports byte addressing, meaning each byte in memory has a unique address. The MIPS processor utilizes a classic five-stage instruction pipeline: instruction fetch, instruction decode, execute, memory access, and write back.

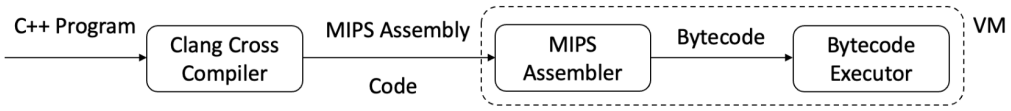
4 Solution

Components and high level flow

The system involves 3 main components.

- Clang cross compiler
- MIPS assembler
- Byte code executer

The input to the virtual machine system is a C++ program which is converted to MIPS assembly code using the Clang 13.0 cross-compiler with no optimizations. The resulting MIPS assembly code is assembled by our MIPS assembler which generates the byte code in accordance with the MIPS32 architecture instruction set. The bytecode is then executed to simulate a run of a MIPS based processor.



Our solution is written in approximately 1200 lines of Python code and was thoroughly tested on 40 programs of varying complexity. (GitHub Repository)

MIPS Assembler

The assembly instructions in MIPS are divided into 3 main categories: *R-type* instructions where the operands are registers, *I-type* instructions with 16 bit immediate value as one of the operands and *J-type* instructions containing the jump label. Every register is associated with a number whose value is used while representing them in the byte code.

A dictionary `REGISTER_MAPPING` maps register names with their corresponding values. `INSTRUCTION_MAPPING` maps operations to their corresponding opcodes and funct numbers. `INSTRUCTION_CLASSIFICATION` that classifies the instructions to various subcategories as listed before.

The assembly instructions are first pre-processed. During pre-processing, the assembler looks for `.data` and `.rodata` directives to check for data that has to be stored in memory. With the `.asciz` and `.4byte` directives, we assign the memory location to the relevant data objects using `free_memory_pointer`. The memory mappings are stored in a dictionary named `memory_mapping`.

Next, the labels are processed based on the presence of a colon (`:`) and their offset(s) are stored in the dictionary `LABEL_MAPPING`. For a jump instruction that refers to a label, the label is replaced by the stored offset in the bytecode.

During assembly, an index `program_counter` is maintained to build a dictionary `program_instructions` which maps the instruction index to its byte code. The instruction-opcode and memory mappings are utilized to stitch together the binary representation of the assembly code line-by-line. The final result of assembly is a hexadecimal byte code which is ready to execute.

Bytecode execution

The byte code execution module utilizes `program_instructions`, `label_mapping` and `memory_mapping` generated by the assembler. The execution begins with the initialization of registers. General Purpose registers are initialized to `0`. `REG_DATA` holds the values of the registers. `REG_DATA["$sp"]` points to the value of the stack pointer and similarly, `REG_DATA["$fp"]` points to the stack frame pointer.

Current instruction to be executed is referenced by `REG_DATA['PC']`, and `REG_DATA['IR']` is set to the byte code of the instruction pointed by the program counter. Having known the PC, fetching instructions is straightforward due to the constant 32-bit size.

The module proceeds to decode the obtained instruction and starts with the opcode. Bits 26-31 are checked for equality with 0. If 0, it is an R-type instruction. If not, `bgez` (bit 17) and `bltz` (bit 25) are checked to determine the instruction via reverse instruction mapping.

To execute *R-type* instructions, operand and destination registers are fetched by accessing specific bits in the 32 bit word. Opcode is 0. and since the negative values are stored in 2s complement form, the signed operand values are to be explicitly calculated from the operand register values. Once the operation is formed fully, the corresponding operation is performed on the signed operand values and saved in the `REG_DATA[rd]`, where rd is the name of the destination register. For division and modulo operations, the quotient is stored in `REG_DATA["LO"]` and the remainder is stored in `REG_DATA["HI"]`.

The *I-type* instructions include a 16 bit immediate value. Since signed values are used, they would have been stored in 2s complement form and have to be explicitly fetched. Destination register value is set based on the operation performed on data values of source register and the immediate value.

For operations that involve memory access such as `lwl`, `lwr`, `lw` etc., the memory address is calculated by adding the source register value and immediate value. Based on the operation the value from the memory is fetched and the destination register is updated. A similar series of steps is carried out for `sw`, `sh` and `sb`. For branch instructions such as `beq` and `bgez`, the value of `REG_DATA["PC']` is updated as appropriate.

The execution of *J-type* instructions involves using the data structures `REG_DATA`, `MEMORY_MAPPING`, `STDOUT_VALUES`, and `LABEL_MAPPING`. There are 2 jump instructions: `j` and `jal`. For the `j` instruction, `REG_DATA['PC']` is updated to the instruction based on the offset value represented by the last 26 bits of the 32 bit word. The `jal` instruction is used for function calls where the `$ra` register stores the return address. When the `jal` instruction is encountered, `$ra` is set to `REG_DATA['PC']` and the program counter is updated to the instruction offset represented by the last 26 bits of the instruction. This is where our custom I/O implementation comes into focus.

Handling I/O

Our VM currently does not support any libraries, hence, we cannot use the standard C++ I/O library `iostream`. We implement a custom I/O interface in C++ with input and print functions for char, string, int and int array. The interface defines functions like `_input_char()`, `_intput_int()`, `_print_string()`, and `_print_int()` to name a few. These functions appear with their name in the assembly code without any body.

During assembly, the offsets for these functions are stored in `label_mapping`. These addresses are retrieved and used in processing of J-type (`jal`) instructions. While processing jal instructions (used for functions), we check if the address in jal instruction matches with the address of either of the I/O interface functions. If there is a match, then an input value has to be processed.

The inputs can be given at run time in an interactive manner or all inputs for a program can be passed while calling `run_vm()` function. In this the inputs are stored in a variable called `file_inputs`. If the `file_inputs` array is empty (i.e inputs are not passed all at once), then the module tries to get an input from the user.

The input processed is stored in memory. The register `$a0` stores the value of the first argument to the function. We process the input as a function with pass by reference, hence `$a0` will have the memory address of the input variable.

The output functions are processed in a similar manner to the input functions. The address in the `jal` instruction is checked for equality with the address of labels `_print_int()`, `_print_char()`, etc. If equal, we fetch the value in the memory mapping for register `$a0`. For integers, we fetch the signed value from 2's complement. For characters, we retrieve

the char from ascii value. The results are printed and stored in the `STDOUT_VALUES` array when the inputs are passed all at once.

Tests and Results

To test our VM, we wrote a suite of around 40 tests, ranging from simple addition of 2 integers to algorithms like merge sort. We test functionality for each operation and data type individually and progressively compose the tests to create more complex programs.

In the 'easy' suite, we test for basic integer arithmetic, I/O, while loops and nested conditions. In the 'medium' suite, we test for character arithmetic, for loops, nested loops, pointers, function calls and recursion. The 'hard' suite tests for array integers, arithmetic on arrays, printing array outputs and working with strings.

Our VM is successfully able to support all data types, operations and constructs listed out in §2. We demonstrate that interesting programs written in the defined subset of C++ can be executed on our system with user interaction:

- Fibonacci series
- Merge sort
- Binary search
- Factorial calculator
- KMP algorithm
- String reversal

Our virtual machine meets the goals we set out with, but there is a huge scope of improvement and building upon what has already been implemented. A GUI is in development to make the visualization of line-by-line execution of a program available. We realize the plethora of possibilities that surface with creating a basic VM, such as designing and building debuggers, enabling system interrupts, adding support for static and dynamic linking thereby supporting libraries. We look forward to taking this project ahead and making it a more capable MIPS32 virtualization platform.