University of Missouri
College of Engineering

ECE 4270/7270 Computer Organization

LAB 1: MIPS Simulator

Section A: TA Yasmin Kassim

September 11, 2018

Group 5

Benjarit Hotrabhavananda, Nic Padilla, John Walter

Work distribution:

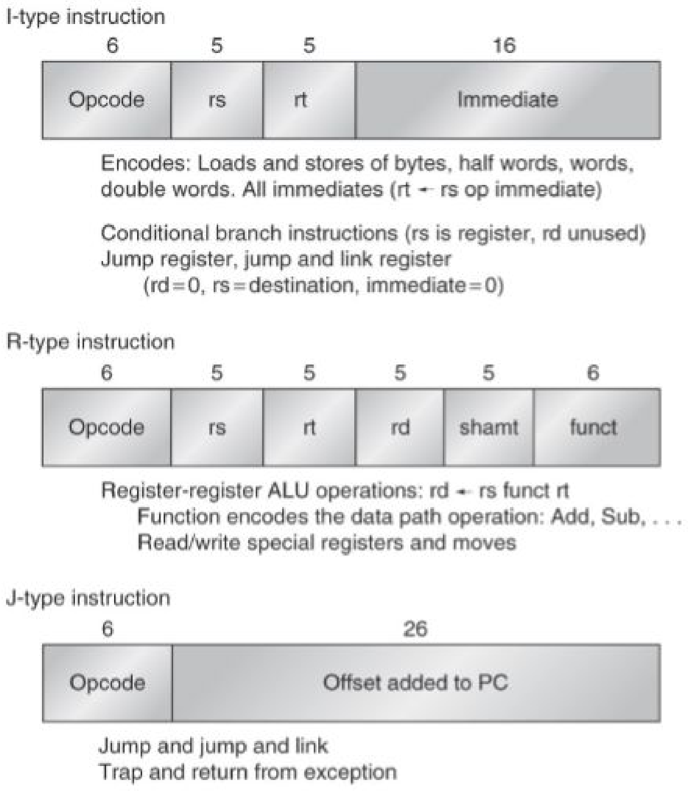
The workload distribution between the members of group 5 was allocated as evenly as possible. Each group member was given a certain aspect of the program to focus on, while also providing help to the other group member. The distribution was as follows. Nic's main focus of this lab was to develop the code implemented in the print_instruction() function, while also contributing to some of the ALU instruction. John's main focus was on the first half of the ALU instructions along with the Load/Store instructions. Finally, Ben's focus was on the second half of the ALU instructions along with the Control Flow instructions. Each instruction was also peer reviewed by the other group members to verify correct implementation. Once completed, the provided test files were simulated in the program and also by hand to check for correctness.

Objective:

The purpose of this lab was to complete a provided skeletal framework for a 32-bit MIPS ISA simulator written in the C programming language. This simulator's purpose was to mimic the functional behavior of the MIPS architecture and produce the desired output. This lab was broken down into two main parts: the development of the handle_instruction() function that would decipher the given hexadecimal input instruction and then simulate the corresponding MIPS functionality, and also the development of the print_instruction() function that would also decipher the given input hexadecimal instruction loaded in memory and then output the instructions translated into the MIPS assembly language. Once completed, this lab was then tested against three provided test files to evaluate the simulator's functionality.

Implementation Decisions:

The basic structure of the handle_instruction() function was the use of a switch statement to house every instruction block. The MIPS assembly language consists of three different types of instructions: I-Type, J-Type, and R-Type. Each type determines the formatting of the 32-bit instruction breakdown. This also means that the instruction code (ADD, SUB, MULT, etc) can be located in different sections of the 32-bits. To accommodate this, the function uses an if-else statement to locate the instruction type and then shifts the instruction bits into the least significant bits of an empty 32-bit integer. This is done so that the switch statement cases are based upon similar constraints. However, because some instructions have the same binary instructional code once shifted into the least significant bits, flags were created to determine where the instruction type was located (left, right, middle) and ultimately which instruction it is. For example, the operation code for ADD and LB are both 0x20, however ADD is located in the least significant bits and LB is located in the most significant bits.



I-type instruction
6
5
5
16
Opcode
rs
rt
Immediate
Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)
Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd=0, rs=destination, immediate=0)
R-type instruction
6
5
5
5
5
6
Opcode
rs
rt
rd
shamt
funct
Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves
J-type instruction
6
26
Opcode
Offset added to PC
Jump and jump and link
Trap and return from exception

There was also the obstacle of obtaining the register and immediate values within each instruction; these being RS, RT, RD, offset, base, and immediate. To reduce redundant code, each value, whether it belongs to that specific instruction or not, is extracted from the instruction statement by the use of bit masking and bit shifting during every call of the handle_instruction() function. This provided easy access to the specific registers and values needed for each case in the switch statement.

The general flow of the program consists of interaction with the CURRENT_STATE and the NEXT_STATE. Each state contains the program counter and the register values. Therefore, each instruction takes values from the current state or memory, performs the instruction's functionality, and then updates the next state's values or writes to the memory. It is important to mention that the simulated memory reserved for program functionality resides between the memory addresses 0x1001000 to 0x7FFFFFFF. All other addresses are reserved for different aspects of the simulator and thereby cannot be used by the program. The input program is loaded into the memory addresses 0x04000000 to 0x0FFFFFFF, and the current state program counter initially points to 0x04000000. This means that to traverse through the program loaded in memory, the NEXT_STATE's program counter is set to the CURRENT_STATE's program counter plus 4. The operation code to stop/finish the simulation corresponds to 0xC . When this instruction is reached, the register $v0 (register 2 of the simulator) is evaluated to determine if its contents is equal to the decimal value 10 (0xA), and if this is true then the RUN_FLAG is set to false to tell the runALL() function that the simulation has completed.

The implementation of the ALU Instructions consisted of basic logic and arithmetic instructions. Several of the instructions contained overflow exceptions, however, all exceptions were not considered in the implementation per lab instruction guidelines, excluding MULT, MULTU, DIV, and DIVU. Similar to regular division abstract, dividing by zero is inclusive. Therefore, both DIV and DIVU verify that the contents of general register RT is not equal to zero. Also, there exists the condition that if the instructions MFHI or MFLO are followed by MULT, MULTU, DIV, or DIVU, then the results of these following instructions are undefined. Therefore, each of these subsequent instructions check to verify that the previous instruction was not equal to MFHI or MFLO before performing its functionality.

The implementation of the Load/Store instructions generally consisted of implementing the mem_read_32() and mem_write_32() functions to interact with the reserved memory for the program. One important issue that arose during testing with the provide test files occurred from and instance where the program was trying to write to non-permitted memory. In the test1.in file, the first statement, 3C031000, is a LUI operation that stores the address 0x10000000 into a register. Several statements later a SW operation is called that tries to write a value to this address, and eventually a LW instruction reads the memory at that address and stores the contents into a register. However, upon examination of memory and register values, only the value 0x0 was present. The issue was eventually found to be due to the previously mentioned concept that the program is only allowed memory access to the memory addresses 0x1001000 to 0x7FFFFFFF, and therefore the address 0x10000000 cannot be utilized.

To implement the Control Flow instructions, we created a new variable, called jmpby, that would hold the amount of bytes that the program counter needs to jump in order to get to the designated address. Instructions BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ are control flow instructions that require checking for a certain conditions to be met first before the program actually can do the jump. For instructions J, JR, JAL, JALR, they are jump instructions without conditions. We need to calculate the address to jump, and then we need to put those calculated address into the program counter, however, these instructions do not add the program counter by calculated address, so we have to do some subtraction of calculated address by the current program counter address to get the offset, then we can put that offset to our jmpBy variable then add it with the program counter.

Similar to the handle_instruction() function, print_instruction() utilizes two embedded switch statements to determine which instruction was to be printed. The instructions are formatted appropriately into a buffer using sprintf() within their respective switch cases. Given that the print function has one argument (the address of the instruction to be interpreted), following the switch statements, simply printing what is in the buffer is required. To be safe, the buffer is initialized to a null terminator at the beginning of each call.