Electrical Engineering
& Computer Science
University of Missouri

ECE 4270: Computer Architecture, Spring 2021
LAB 1: MIPS Simulator

February 16, 2021

Samuel Bishop-Gutknecht

Wan Fatimah Wan Nawawi

Introduction

Microprocessor without Interlocked Pipeline Stages (MIPS) is a Reduced Instruction Set Computer (RISC) architecture that can access the memory using only the load and store instructions and transfer data between the main memory and the registers. The Arithmetic-Logic Unit (ALU) can perform on the registers only and not in the memory. There are three CPU instruction formats for MIPS and each format consists of a 32-bit. The three instructions are immediate (I-type), jump (J-type), and register (R-type). The CPU register consists of 32 general purpose registers, a program counter (PC) register, a high register for multiplication and division operations (HI), and a low register for multiplication and division operations (LO). The register at location zero (r0) stores the value 0 or to discard any results from running the instruction. While the register at location 31 (r31) stores the address from jump and link instruction.

The objective of this lab is to implement two functions in a given code that would simulate the 32-bit MIPS ISA simulation. The first function is the 'handle_instruction()' that would simulate each new instruction by identifying the type of instruction and executing the instruction and its operands. The second function 'print_instruction()' is to print each instruction after identifying each instruction in the register.

Implementation

handle_instruction()

For the instructions, we start by loading the instructions from the memory using the given function mem_read_32. After reading the instructions from the memory, we would mask the instructions, by obtaining and storing the opcode, rt, rd, rs, base, function, target and immediate value.

To retrieve the values stored in the address, we would assign the value '1' on the specific bits of the address for each register. The operand code (special) is located on bit 31-26. The register specifier (rs) is obtained by masking the address on bit 25-21. The temporary register (rt), is obtained by masking the address on bit 20-16. The destination register (rd) can be obtained by masking the address on bit 15-11. The shift amount (sa) is obtained on the 10-6. The function field (function) is obtained on bit 5-0. The target jump address (target) can be obtained in bit 25-0. The immediate value (immediate) is obtained on bit 15-0.

```
// Reading Instruction and Relevant Masks
uint32_t instruction = mem_read_32(addr);          // Get the 32-bit instruction from memory
uint32_t specialMask = 0xFC000000;                 // Mask for bits 26-31
uint32_t rsMask = 0x03E00000;                      // Mask for bits 21-25
uint32_t rtMask = 0x001F0000;                      // Mask for bits 16-20
uint32_t rdMask = 0x0000F800;                      // Mask for bits 11-15
uint32_t functMask = 0x0000003F;                   // Mask for bits 1 - 6
uint32_t immediateMask = 0x0000FFFF;               // Mask for bits 0 -16
uint32_t branchMask = 0x001F0000;                  // Mask for bits 16-20

// Variables for R-Type Instructions
uint32_t special    = (instruction & specialMask)   >> 26;  // Shifting to get correct digits
uint32_t rs         = (instruction & rsMask)        >> 21;
uint32_t rt         = (instruction & rtMask)        >> 16;
uint32_t rd         = (instruction & rdMask)        >> 16;
uint32_t function   = instruction & functMask;

// Variables for I-Type Instructions
uint32_t immediate  = instruction & immediateMask;

// Variables for J-Type Instructions
uint32_t branch     = instruction & branchMask;
```

Figure 1. Setting up masks and register variables

After determining the instruction, we would perform the specific instruction simulation based on the appropriate masking values obtained. We were able to understand how to obtain these instructions based on the MIPS R4000 Microprocessor User's Manual by Joe Henrich. On page 36 of the manual, it states that there are three types of CPU instruction formats that require different registers to perform their specific instruction. Based on this information, we know that we require eight variables for all three registers and we can determine the bit location of each register in order to do the masking. In order to determine the instructions, a switch case was used to compare the opcode (bit 31-26) of the instructions. If bit 31-26 are all zeros, they are considered special instruction. Therefore, bits 5-0 are being checked to determine their special instruction. If bits 31-26 are "000001" they are considered REGIMM instruction, therefore bits 20-16 will be checked to determine their REGIMM instruction. Each instruction is coded based on their operation from the manual. By referring to the manual, we were able to determine the conditions to satisfy the instruction.

```
// Register case code
case 0b000110:
    sprintf(returnString, "BLEZ $%d, %d\n", rs, immediate);
    offset = offset << 2;
    if(((offset & 0x00008000)>>15)){
        offset = offset | 0xFFFF0000;
    }
    if(((CURRENT_STATE.REGS[rs] & 0x80000000)>>31) || (CURRENT_STATE.REGS[rs] == 0x00)){
        jumpAmmount = offset;
    }
    break;

case 0b000001:
    switch (rt){
    case 0b00000:
        sprintf(returnString, "BLTZ $%d, %d\n", rs, immediate);
        offset = offset << 2;
            // sign extend (check if most significant bit is a 1)
            if(((offset & 0x00008000)>>15)){
                offset = offset | 0xFFFF0000;
            }
            if(((CURRENT_STATE.REGS[rs] & 0x80000000)>>31)){
                jumpAmmount = offset; // changed
            }
        break;
```

Figure 2. Example of handle_instruction switch logic for the BLEZ and BLTZ instructions

print_instruction()

Performing the similar loading instructions from memory using the given function mem_read_32 and similar methods previously in the handle_instruction to mask in order to obtain the opcode, rt, rd, rs, base, immediate, base and other required values. Similarly to the handle_instruction(), after retrieving the required values, using a switch case to compare the opcode and printing the instructions based on the format of the instruction by referring from the manual.

```
switch (special)
{
// Special case code
case 0b000000:
    switch (function)
    {
    case 0b100000:
        sprintf(returnString, "ADD $%d, $%d, $%d\n", rd, rs, rt);
        break;
    case 0b100001:
        sprintf(returnString, "ADDU $%d, $%d, $%d\n", rd, rs, rt);
        break;
    case 0b100010:
        sprintf(returnString, "SUB $%d, $%d, $%d\n", rd, rs, rt);
        break;
    case 0b100011:
        sprintf(returnString, "SUBU $%d, $%d, $%d\n", rd, rs, rt);
        break;
```

Figure 3. Printing the associated instruction and register values with a switch case

Work Distribution:

Work Assigned	Assigned To
Location of Instructions	Zach
Switch Statement Logic for print_instruction() R-type Instructions	Zach
Switch Statement Logic for print_instruction() I-type Instructions	Zach
Switch Statement Logic for print_instruction() J-type Instructions	Zach
Figuring out the increment of program counter	Zach
Creation of Bit Masks	Sam
Switch Statement Logic for handle_instruction() R-type Instructions	Sam
Switch Statement Logic for handle_instruction() I-type Instructions	Sam
Switch Statement Logic for handle_instruction() J-type Instructions	Sam

Milestones and Implementation Decisions:

1. Creation of the bit masks for relevant CPU registers.
2. Application of bit masks onto the instruction to get register values.
3. Switch case logic to create individual cases for each specified MIPS instruction.
4. Application of print statements for each MIPS instruction in the print_instruction() function.
5. Copying the print_instruction code into the handle_instruction() function and application of opcode concept.
6. Implementation of code to put values within the registers in the data structures CURRENT_STATE and NEXT_STATE.

Results

Upon compiling the program using the mu-mips.c, mu-mips.h files, and the Makefile a mu-mips.exe executable was generated. The executable was then run using the test1.in file which contains 56 (32-bit) words containing MIPS instructions for interpretation by our program.

At the boot of our program the program indicates that it has received all 56 instructions, and it writes them into 4 byte long memory locations as shown below.

```
writing 0x2410000a into address 0x004000ac (4194476)
writing 0x1c40fff3 into address 0x004000b0 (4194480)
writing 0x2412000a into address 0x004000b4 (4194484)
writing 0x0c100021 into address 0x004000b8 (4194488)
writing 0x2414000a into address 0x004000bc (4194492)
writing 0x03c0f809 into address 0x004000c0 (4194496)
writing 0x00e0b027 into address 0x004000c4 (4194500)
writing 0x32d67fff into address 0x004000c8 (4194504)
writing 0x02cab826 into address 0x004000cc (4194508)
writing 0x0000c012 into address 0x004000d0 (4194512)
writing 0x02e00013 into address 0x004000d4 (4194516)
writing 0x2402000a into address 0x004000d8 (4194520)
writing 0x0000000c into address 0x004000dc (4194524)
Program loaded into memory.
56 words written into memory.
```

Figure 4. The executable acknowledging it has received the instructions from the test1.in file

Then the user is prompted with the MU-MIPs program help menu with options to select the sim, run, rdump, reset, input, mdump, high, low, print, ?, quit. Typing “sim” executes the program to completion. Typing “run <instruction number>” simulates the program for a set number of instructions. Rdump dumps all register values into the terminal. Reset clears all register/memory and re-loads the program. Input <reg> <val> sets the gpr register to value. Mdump <start> <stop> dumps memory from <start> until <stop> address. High <val> sets the HI register to that value. Low <val> sets the LO register to that value. Print prints the program loaded into memory. ? displays the help menu. Quit exits the simulator.

The two functions that we implemented in this program are called print_instruction() and handle_instruction() which are triggered by the commands in print and run respectively within the help menu. We believe to have implemented the print_instruction command correctly and the results are as follows.

```
MU-MIPS SIM:> print
[0x400000]      LUI $2, 4097($0)
[0x400004]      LUI $3, 16($0)
[0x400008]      SW $3, 0($2)
[0x40000c]      SRL $0, $3, $0
[0x400010]      SRA $0, $4, $4
[0x400014]      SH $4, 4($2)
[0x400018]      ADDI $2, $2, 8
[0x40001c]      ORI $4, $4, 1
[0x400020]      ADDI $4, $4, 1
[0x400024]      SB $4, 0($2)
[0x400028]      ADDI $3, $2, 0
[0x40002c]      LB $5, 0($3)
[0x400030]      SLL $0, $5, $2
[0x400034]      SUB $0, $2, $5
[0x400038]      LW $7, 0($6)
[0x40003c]      LH $8, 4($6)
[0x400040]      XORI $9, $4, 5
[0x400044]      OR $0, $7, $4
[0x400048]      DIVU $7, $9
[0x40004c]      MFHI $0
[0x400050]      MFLO $0
[0x400054]      ADDI $2, $2, 32766
[0x400058]      MULTU $2, $3
```

Figure 5. The result of running the print command which cycles the print_instruction() function

The other main function that we implemented was the handle_instruction function which depends on using a switch statement similar to the one used in the print_instruction function and

then implementing the logic that would simulate those instructions. Our code for those simulated instructions can be found in our attached code file or on our github repository.

```
------------------------------------
Dumping Register Content
------------------------------------
# Instructions Executed : 55
PC      : 0x00400000
------------------------------------
[Register]      [Value]
------------------------------------
[R0]    : 0x00000000
[R1]    : 0x00000000
[R2]    : 0x0000000a
[R3]    : 0x00000000
[R4]    : 0x00000000
[R5]    : 0x00000000
[R6]    : 0x00000009
[R7]    : 0x00000000
[R8]    : 0x00000000
[R9]    : 0x00000005
[R10]   : 0x00000000
[R11]   : 0x00000000
[R12]   : 0x0000000a
[R13]   : 0x0000000a
[R14]   : 0x0000000a
[R15]   : 0x0000000a
[R16]   : 0x0000000a
[R17]   : 0x0000000a
[R18]   : 0x0000000a
[R19]   : 0x0000000a
[R20]   : 0x0000000a
[R21]   : 0x0000000a
[R22]   : 0x00007fff
[R23]   : 0x00007fff
[R24]   : 0x00000000
[R25]   : 0x00000000
[R26]   : 0x00000000
[R27]   : 0x00000000
[R28]   : 0x00000000
[R29]   : 0x00000000
[R30]   : 0x00000001
[R31]   : 0x00400008
```

```
MU-MIPS SIM:> mdump 0 55
-------------------------------------------------------------
Memory content [0x00000000..0x00000055] :
-------------------------------------------------------------
        [Address in Hex (Dec) ] [Value]
        0x00000000 (0) :        0x00000000
        0x00000004 (4) :        0x00000000
        0x00000008 (8) :        0x00000000
        0x0000000c (12) :       0x00000000
        0x00000010 (16) :       0x00000000
        0x00000014 (20) :       0x00000000
        0x00000018 (24) :       0x00000000
        0x0000001c (28) :       0x00000000
        0x00000020 (32) :       0x00000000
        0x00000024 (36) :       0x00000000
        0x00000028 (40) :       0x00000000
        0x0000002c (44) :       0x00000000
        0x00000030 (48) :       0x00000000
        0x00000034 (52) :       0x00000000
        0x00000038 (56) :       0x00000000
        0x0000003c (60) :       0x00000000
        0x00000040 (64) :       0x00000000
        0x00000044 (68) :       0x00000000
        0x00000048 (72) :       0x00000000
        0x0000004c (76) :       0x00000000
        0x00000050 (80) :       0x00000000
        0x00000054 (84) :       0x00000000
```

Figure 6. The rdump(left) and rdump(right) results after running the simulator which cycles the handle_instruction() function

Conclusion

Our group did end up completing the lab in the allotted amount of time and we feel confident with our c code and how the simulator performs. Implementation of each and every instruction was somewhat tedious because understanding each instruction required referencing the MIPS R4000 Microprocessor manual for each instruction to understand how execution of the instruction is determined and which resources it utilizes. However, conceptually this lab was somewhat easy to understand. All it required was understanding the segmentation of MIPS

instructions for each resource and then being able to mask the address string provided with a series of hexadecimal values. Once the resource values were obtained it was as easy as printing those values out or adding logic within the code to perform the given instruction's operation.

References

Shen, John Paul, and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, 2013.

Heinrich, Joe. *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies, 1994.