Project 1

CDA 4102/CDA 5155: Fall 2021

Due: Oct 06, 11:30 pm

You are not allowed to take or give help in completing this project. **No late submission will be accepted**. Please include the following sentence on top of your source file: "**On my honor, I have neither given nor received any unauthorized aid on this assignment**".

In this project you will create a simple MIPS simulator which will perform the following two tasks. **Please develop your project in one** (C, C++, Java or Python) **source file** to avoid the stress of combining multiple files before submission and making sure it still works correctly.

- Load a specified MIPS text file[1] and generate the assembly code equivalent to the input file (**disassembler**). Please see the sample input file and disassembly output in the project assignment.
- Generate the instruction-by-instruction simulation of the MIPS code (**simulator**). It should also produce/print the contents of *registers* and *data memories* after execution of each instruction. Please see the sample simulation output file in the project assignment.

You do not have to implement any exception or interrupt handling for this project. We will use only valid testcases that will not create any exceptions. Please go through this document first, and then view the sample input/output files in the project assignment, before you start implementing the project.

Instructions

For reference, please use the MIPS Instruction Set Architecture PDF (available from the project1 assignment) to see the format for each instruction **and pay attention to the following changes.**

Your disassembler & simulator need to support the three categories of instructions shown in **Figure 1.**

Category-1	Category-2	Category-3
J, BEQ, BNE, BGTZ, SW, LW, BREAK	ADD, SUB, AND, OR, SRL, SRA, MUL	ADDI, ANDI, ORI

Figure 1: Three categories of instructions

The format of **Category-1** instructions is described in **Figure 2**. If the instruction belongs to **Category-1**, the first three bits (leftmost bits) are always "**000**" followed by **3 bits** Opcode. Please note that instead of using 6 bits opcode in MIPS, we use 3 bits opcode as described in **Figure 3**. The remaining part of the instruction binary is exactly the same as the original MIPS instruction set for that specific instruction.

000	Opcode (3 bits)	Same as MIPS instruction

Figure 2: Format of Instructions in Category-1

Please pay attention to the exact description of instruction formats and its interpretation in MIPS instruction set manual. *For example, in case of **J** instruction, the 26 bit instruction_index is shifted left by two bits (padded with 00 at LSB side) and then the leftmost (MSB side) four bits of the delay slot*

[1] This is a text file consisting of 0/1's (not a binary file). See the sample input file sample.txt in the project1 assignment.

*instruction are used to form the four bits (MSB side) of the target address. Since we do not use delay slot in this project, treat the address of the next instruction as the address of the delay slot instruction. Similarly, for **BEQ**, **BNE** and **BGTZ** instructions, the 16 bit offset is shifted left by two bits to form 18 bit signed offset that is added with the address of the next instruction to form the target address.* **Please note that we do not consider** *delay slot* **for this project**. *In other words, an instruction following the branch instruction should be treated as a regular instruction (see sample_simulation.txt).*

Instruction	Opcode
J	000
BEQ	001
BNE	010
BGTZ	011
SW	100
LW	101
BREAK	110

Figure 3: Opcode for Category-1 instructions

If the instruction belongs to **Category-2** which has the form "dest ← src1 op src2", the first three bits (leftmost three bits) are always "001" as shown in **Figure 4**. Then the following 5 bits serve as dest. The next 5 bits for src 1, followed by 5 bits for src2. The src1 is always register but src2 can be register (ADD, SUB, AND, OR, MUL) or immediate (SRL, SRA) depending on the opcode. The remaining bits are all 0's. The three-bit opcodes are listed in **Figure 5**.

001	opcode (3 bits)	dest (5 bits)	src1 (5 bits)	src2 (5 bits)	00000000000

Figure 4: Format of Category-2 instructions where both sources are registers

Instruction	Opcode
ADD	000
SUB	001
AND	010
OR	011
SRL	100
SRA	101
MUL	110

Figure 5: Opcode for Category-2 instructions

If the instruction belongs to **Category-3** which has the form "dest ← src1 op immediate_value", the first three bits (leftmost three bits) are always "010". Then 3 bits for opcode as indicated in **Figure 6**. The subsequent 5 bits serve as dest followed by 5 bits for src1. The second source operand is immediate 16-bit value. The instruction format is shown in **Figure 7**.

Instruction	Opcode
ADDI	000
ANDI	001
ORI	010

Figure 6: Opcode for Category-3 instructions

010	opcode (3 bits)	dest (5 bits)	src1 (5 bits)	immediate_value (16 bits)

Figure 7: Format of Category-3 instructions with source2 as immediate value

Once you look at the sample_disassembly.txt in the project assignment, it may be confusing for you to see that the last 16 bits of the following binary (offset) has the value of 9 but the assembly shows it as 36. This is a convention issue with MIPS. The binary always shows the actual offset (9 in this case) value. However, the assembly always shows the value shifted by 2 bits to the left (i.e., multiplied by 4).

0000010000100010 0000000000001001 276 BEQ R1, R2, #36

Please note there are also convention related confusion for other instructions. For example, in many binary format, the destination is the middle operand, whereas the destination always shows up as the leftmost operand in assembly instructions <opcode, dest, src1, src2>. Moreover, assume that all unassigned register and data memory locations are 0.

All signed numbers should be interpreted using 2's complement arithmetic. Note that the signed numbers can be in registers, data memories or inside an instruction (e.g., the immediate field is signed for ADDI). Most importantly, each location (register or data memory) can be treated differently based on the context. For example, an arithmetic instruction (e.g., ADD) will treat the content of a register as a signed number (in 2's complement arithmetic), whereas a logical operation (e.g., AND) will treat the same register content as an unsigned number (sequence of bits). Please go through mips.pdf to understand how each instruction treats its operands (signed or unsigned).

Sample Input/output Files

Your program will be given a text input file (see sample.txt). This file will contain a sequence of 32-bit instruction words starting at address **"260"**. The final instruction in the sequence of instructions is always BREAK. There will be only one BREAK instruction. Following the BREAK instruction (immediately after BREAK), there is a sequence of 32-bit 2's complement signed integers for the program data up to the end of the file. The newline character can be either "\n" (linux) or "\r\n" (windows). Your code should work for both cases. ***Please download the sample input/output files using "Save As" instead of using copy/paste of the content.***

Your MIPS simulator (with executable name as **MIPSsim**) should accept an input file (**inputfilename.txt**) in the following command format and produce two output files in the same directory: **disassembly.txt** (contains disassembled output) and **simulation.txt (**contains the simulation trace). Please hardcode the names of the output files. ***Please do not hardcode the input filename***. *It will be specified when running your program. For example, it can be "sample.txt" or "test.txt".*

MIPSsim inputfilename.txt

Correct handling of the sample input file (with possible different data values) will be used to determine 60% of the credit. The remaining 40% will be determined from other valid test cases that you will not have access prior to grading. It is recommended that you construct your own sample input files with which to further test your disassembler/simulator.

The disassembler output file should contain 3 columns of data with each column separated by one tab character ('\t' or char(9)). See the sample disassembly file in the project1 assignment.

1. The text (e.g., 0's and 1's) string representing the 32-bit data word at that location.
2. The address (in decimal) of that location
3. The disassembled instruction.

Note, if you are displaying an instruction, the third column should contain every part of the instruction, with each argument separated by a comma and then a space (", ").

The simulation output file should have the following format.

20 hyphens and a new line

Cycle < cycleNumber >:< tab >< instr_Address >< tab >< instr_string >

< blank_line >

Registers

R00: < tab >< int(R0) >< tab >< int(R1) >...< tab >< int(R7) >

R08: < tab >< int(R8) >< tab >< int(R9) >...< tab >< int(R15) >

R16: < tab >< int(R16) >< tab >< int(R17) >...< tab >< int(R23) >

R24: < tab >< int(R24) >< tab >< int(R25) >...< tab >< int(R31) >

< blank_line >

Data

< firstDataAddress >: < tab >< display 8 data words as integers with tabs in between >

..... < continue until the last data word >

The instructions and instruction arguments should be in capital letters. Display all integer values in decimal. Immediate values should be preceded by a "#" symbol. **Note that some instructions take signed immediate values while others take unsigned immediate values**. You will have to make sure you properly display a signed or unsigned value depending on the context.

Because we will be using "**diff –w -B**" to check your output versus the expected outputs, please follow the output formatting. Mismatches will be treated as wrong output and will lead to score penalty.

The project assignment contains the following sample programs/files to test your disassembler/simulator.

- sample.txt : This is the input to your program.
- sample_disassembly.txt : This is what your program should produce as disassembled output.
- sample_simulation.txt : This is what your program should output as simulation trace.

Submission Policy:

Please follow the submission policy outlined below. There can be up to **10% score penalty** based on the nature of submission policy violations.

1. **Please develop your project in one source file**. In other words, you cannot submit your project if you have designed it using multiple source files. **Please add ".txt" at the end of your filename**. Your file name must be MIPSsim (e.g., MIPSsim.c.txt or MIPSsim.cpp.txt or MIPSsim.java.txt or MIPSsim.py.txt). On top of the source file, please include the sentence: "/* On my honor, I have neither given nor received unauthorized aid on this assignment */".
2. Please test your submission. These are the exact steps we will follow too.
    - Download your submission from eLearning (ensures your upload was successful).
    - Remove ".txt" extension (e.g., MIPSsim.c.txt should be renamed to MIPSsim.c)
    - Login to any CISE linux machine (e.g., **thunder**.cise.ufl.edu or **storm**.cise.ufl.edu) using your Gatorlink login and password. Then you use **putty** and **winscp** or other tools to login. Ideally, if your program works on any Linux machine, it should work when we run them. However, if you get correct results on a Windows or MAC system, we may not get the same results when we run on storm or thunder. To avoid this headache and time waste, we strongly recommend that you should test your program on thunder or storm server.
    - Please compile to produce an executable named **MIPSsim**.
        - gcc MIPSsim.c –o MIPSsim **or** javac MIPSsim.java **or** g++ -std=c++17 MIPSsim.cpp –o MIPSsim
    - Please do not print anything on screen.
    - Please do not hardcode input filename, accept it as a command line option. You should hardcode your output filenames. Execution should always produce **disassembly.txt** and **simulation.txt** irrespective of the input filename.
    - Execute to generate disassembly and simulation files and test with correct/provided ones
        - ./MIPSsim inputfilename.txt **or** java MIPSsim inputfilename.txt **or** ./MIPSsim.py inputfilename.txt **or** python3 MIPSsim.py inputfilename.txt
        - diff –w –B disassembly.txt sample_disassembly.txt
        - diff –w –B simulation.txt sample_simulation.txt
3. *In previous years, there were many cases where output format was different, filename was different, command line arguments were different, or e-Learning submission was missing, etc. All of these led to un-necessary frustration and waste of time for TA, instructor and students.* **Please use the exactly same commands as outlined above to avoid 10% score penalty.**
4. You are not allowed to take or give any help in completing this project. *In the previous years, some students violated academic honesty. We were able to establish violation in several cases - those students received "0" in the project, and their names were reported to Dean of Students Office (DSO). If your name is already in DSO for violation in another course, the penalty for second offence is determined by DSO. In the past, two students from my class were suspended for a semester due to repeat academic honesty violation (implies deportation for international students).*