Project 1

CDA 4102/CDA 5155: Fall 2025

Due: October 21, 11:30 pm

You are not allowed to take or give help in completing this project. Submit the source file in eLearning before the deadline. Late submission (by email attachment to nkim2@ufl.edu) is allowed (up to 24 hours) with a 20% penalty (irrespective of whether it is late for 10 minutes or 10 hours). No grades for late submissions after 24 hours from the deadline. *Use of AI (e.g., ChatGPT) in any form is prohibited to do the project*. Please include the following sentence on top of your source file (as a comment).

On my honor, I have neither given nor received any unauthorized aid on this assignment.

In this project you will create a simple RISC-V 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 RISC-V 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 RISC-V 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

You can refer to RISC-V Instruction Set Architecture (riscv-ISA.pdf in the course website) to see the format for each instruction **and pay attention to the following changes.** For example, we introduced a **break** instruction, modified the opcode format, etc. In other words, you should exactly follow the details from riscv-ISA.pdf except for the changes outlined in this document. In this project, we will be using the instruction format from Figure A.23 and A.24 in the book (slide 54 of instruction.pptx).

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

Category-1	Category-2	Category-3	Category-4
beq, bne, blt, sw	add, sub, and, or	addi, andi, ori, sll, sra, lw	jal, break

Figure 1: Four categories of instructions

The format of **Category-1** instructions is described in **Figure 2**. It has the same format as the S-type instruction in slide 54 in instruction.pptx except the rightmost two bits. If the instruction belongs to **Category-1**, the rightmost two bits (least significant bits) are always "**00**" preceded by **5 bits** Opcode. Note that instead of using 7 bits opcode in RISC-V, we use 5 bits opcode as described in **Figure 3**. Assume func3 as "000".

imm[11:5]	rs2	rs1	func3	imm[4:0]	Opcode (5 bits)	00

[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.

Figure 2: Format of Instructions in Category-1

Please pay attention to the exact description of instruction formats and their interpretation in RISC-V instruction set. *For example, in case of **jal** instruction, the 20-bit offset is shifted left by one bit (padded with 0 at LSB side), sign extended to form 32 bits, and then added to the address of the jal instruction to form the target address. Similarly, for **beq**, **bne** and **blt** instructions, the 12-bit offset is formed by concatenating bits in [31:25] with bits in [11:7], and then the 12-bit offset is shifted left by one bit, sign extended to form 32-bits, and added to the address of the current instruction to form the target address.* **Please note that we do not consider** *delay slot* **for this project**.

Instruction	Opcode
beq	00000
bne	00001
blt	00010
sw	00011

Figure 3: Opcode for Category-1 instructions

If the instruction belongs to **Category-2** which has the form "dest ← src1 op src2", the rightmost two bits (least significant bits) are always "01" as shown in **Figure 4**. It has the same format as the R-type instruction in slide 54 in instruction.pptx except the rightmost two bits. Then the preceding 5 bits serve as opcode as listed in **Figure 5**. Assume func3 as "000" and func7 as "0000000".

func7	rs2	rs1	func3	rd	Opcode (5 bits)	01

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

Instruction	Opcode
add	00000
sub	00001
and	00010
or	00011

Figure 5: Opcode for Category-2 instructions

If the instruction belongs to **Category-3** which has the form "dest ← src1 op immediate_value", the rightmost two bits (least significant bits) are always "10". It has the same format as the I-type instruction in slide 54 in instruction.pptx except the rightmost two bits. Then 5 bits for opcode as indicated in **Figure 6**. The instruction format is shown in **Figure 7**. Assume func3 as "000".

Instruction	Opcode
addi	00000
andi	00001
ori	00010
sll	00011
sra	00100

lw	00101

Figure 6: Opcode for Category-3 instructions

imm[11:0]	rs1	func3	rd	Opcode (5 bits)	10

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

If the instruction belongs to **Category-4**, the rightmost two bits (least significant bits) are always "11". Then 5 bits for opcode as indicated in **Figure 8**. The instruction format is shown in **Figure 9**. It has the same format as the U-type instruction in slide 54 in instruction.pptx except the rightmost two bits. Also note that the U-type format in the slide shows "imm[31:12]" but we show it as "imm[19:0]" – both means the same as 20-bit immediate value. Finally, we use the full functionality of "jal" (i.e., not assuming rd as x0).

Instruction	Opcode
jal	00000
break	11111

Figure 8: Opcode for Category-4 instructions

imm [19:0]	rd	Opcode (5 bits)	11

Figure 9: Format of Category-4 instructions

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 32-bit 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 riscv-ISA.pdf to understand how each instruction treats its operands (signed or unsigned). **Assume that all unassigned register and data memory locations are 0.**

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 **"256"**. 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 RISC-V simulator (with executable name as **Vsim**) 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".*

Vsim inputfilename.txt

Correct handling of the sample input file (with possible different data values) will be used to determine 40% of the credit. The remaining 60% 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. It is okay to share your new testcases with other students in the class as long as it does not lead to similarity in the project source code.

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

x00: < tab >< int(x0) >< tab >< int(x1) >...< tab >< int(x7) >

x08: < tab >< int(x8) >< tab >< int(x9) >...< tab >< int(x15) >

x16: < tab >< int(x16) >< tab >< int(x17) >...< tab >< int(x23) >

x24: < tab >< int(x24) >< tab >< int(x25) >...< tab >< int(x31) >

< blank_line >

Data

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

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

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 **20% 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 Vsim (e.g., Vsim.c.txt or V.cpp.txt or V.java.txt or Vsim.py.txt).
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., Vsim.c.txt should be renamed to Vsim.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 see the notes below on accessing CISE machines.
    - Please compile to produce an executable named **Vsim**.
        - gcc Vsim.c –o Vsim **or** javac Vsim.java **or** g++ -std=c++17 Vsim.cpp –o Vsim
    - 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
        - ./Vsim inputfilename.txt **or** java Vsim inputfilename.txt **or** ./Vsim.py inputfilename.txt **or** python3 Vsim.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 20% score penalty.**
4. **You are not allowed to take or give any help in completing this project.** Use of AI (e.g., ChatGPT) to develop the code, irrespective of whether it is a few lines or full project, will be treated as a violation of academic honesty. *In 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). The penalty would be higher this year - I will give a failing score in the class. 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 violations (imply deportation for international students).*

Accessing CISE Servers (storm or thunder) to run Project 1

Please ignore this section if you know how to use CISE servers (storm or thunder) to run programs.

5. Register for a CISE account using your Gatorlink account and password.
   https://register.cise.ufl.edu/
6. Download PuTTY or similar shell to run your project using Linux. If you are using MAC, you can use the Terminal client.
7. Download WinSCP or similar file transfer mechanism. You can use sftp command from your MAC terminal client.
8. Once you register (step 1), you will get an email from do-not-reply@cise.ufl.edu indicating "your account has been created". Unless you get this email, the subsequent steps will not work.
9. Open PuTTY and login to any of the CISE servers (storm.cise.ufl.edu or thunder.cise.ufl.edu) using your Gatorlink account and password. Create a directory for your project 1 (e.g., mkdir project1).
10. Open WinSCP and login to any of the CISE servers (storm.cise.ufl.edu or thunder.cise.ufl.edu) using your Gatorlink account and password. Transfer files from your Windows directory to the directory you created (e.g., project1).
11. Go to PuTTY again and run the command from that directory (e.g., cd project1, g++ ...., diff ...., etc.). If your implementation is correct, 'diff' command (e.g., diff -w -B simulation.txt correct_simulation.txt) should not provide any output.

If your project 1 runs perfectly on VSCode, but it produces mismatches on CISE servers (storm or thunder), here is a potential fix.

12. Login to Storm or Thunder, go to the testing directory.
13. Perform dos2unix on all three input files. Make sure you have the write permissions for these three files (not read only).
    > dos2unix sample.txt
    > dos2unix sample_disassembly.txt
    > dos2unix sample_simulation.txt
14. Run “diff -w -B simulation.txt sample_simulation.txt” to make sure there are no differences.

If the above steps fixed the differences, your code is fine (submit it). This issue relates to how Windows and Linux look at lines in a file if you implemented using C or C++. Specifically, in one case it ends with one character “**\n**”, and in another case it ends with “**\r\n**”. If you have used Java or Python, you may not face this problem.