Project 2

CDA 3101: Spring 2022

Due Date: April 13, 11:30 pm **Total Points:** 10 points (5% of overall score)

Submit your source code in e-Learning (Canvas) before the deadline. **No late submission will be accepted**. You are not allowed to take or give help in completing this project. Include the following sentence on top of your source file:

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

You **MUST** verify that your submission in Canvas is successful by downloading your submission from Canvas and successfully testing it using the procedure outlined in the **Submission Policy** (see Section 3). This will ensure that you have uploaded the right file in eLearning and the upload was successful. This will avoid zero score in Project2.

In this project, you will implement (using **C**, **C++**, **Java,** or **Python**) an ARM simulator to perform the following two tasks:

- Load a specified ARM 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 2 assignment webpage** at

  https://ufl.instructure.com/courses/448395/assignments/5067305

- Generate the instruction-by-instruction simulation of the ARM 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 class assignments page.

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 (including **submission policy**), and then view the sample input/output files in the project assignment, before you start implementing the project.

1. Instructions

For reference, please use the ARM LEGv8 Reference Data to see the format for each instruction **and pay attention to the following changes.** Also, look at the ARM instruction set (PDF) in the project 2 assignment webpage. Your disassembler and simulator need to support the four categories of instructions shown in **Figure 1.**

Category-1	Category-2	Category-3	Category-4
CBZ, CBNZ	ADDI, SUBI, ANDI, ORRI, EORI	ADD, SUB, AND, ORR, EOR, LSR, LSL	LDUR, STUR

Figure 1: Four 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 "**001**" followed by **5 bits** Opcode. Please note that instead of using 8 bits opcode in ARM, we use 5 bits opcode as described in **Figure 3**. Note that you need to left shift the branch offset by two bits and add to the current PC to determine the actual branch target (new PC address if the branch is taken).

001	Opcode (5 bits)	Src1 (5 bits)	Branch Offset (19 bits)

Figure 2: Format of Instructions in Category-1.

Instruction	Opcode
CBZ	**10000**
CBNZ	**10001**

Figure 3: Opcode for Category-1 instructions

[1] This is a text file consisting of 0/1's (not a binary file). See the sample input file (sample.txt) in the Project 2 assignment.

If the instruction belongs to **Category-2**, which has the form "dest ← src1 op immediate_value", the first three bits (leftmost three bits) are always "**010**". The 7 bits for opcode is indicated in **Figure 4**. The subsequent 5 bits serve as dest followed by 5 bits for src1. The second source operand is immediate 12-bit value. The instruction format is shown in **Figure 5**.

Instruction	Opcode
ORRI	1000000
EORI	1000001
ADDI	1000010
SUBI	1000011
ANDI	1000100

Figure 4: Opcode for Category-2 instructions

010	opcode (7 bits)	dest (5 bits)	src1 (5 bits)	immediate_value (12 bits)

Figure 5: Format of Category-2 instructions with source2 as immediate value

If the instruction belongs to **Category-3**, which has the form "dest ← src1 op src2", the first three bits (leftmost three bits) are always "**011**" as shown in **Figure 6**. The next eight bits denote opcode. Then the following 5 bits serve as dest. The next 5 bits for src1, followed by 5 bits for src2. Dest, src1 as well as src2 are registers. The remaining bits are all 0's. The three bit opcodes are listed in **Figure 7**. Assume that for **LSR** and **LSL** instructions, shift amount is the **least significant 5 bits** of src2.

011	opcode (8 bits)	dest (5 bits)	src1 (5 bits)	src2 (5 bits)	000000

Figure 6: Format of Category-3 instructions where both sources are registers

Instruction	Opcode
EOR	10100000
ADD	10100010
SUB	10100011
AND	10100100
ORR	10100101
LSR	10100110
LSL	10100111

Figure 7: Opcode for Category-3 instructions

If the instruction belongs to **Category-4**, which has the form "srcdst ← Memory [src1 + immediate_value]" or "Memory [src1 + immediate_value] = srcdst. The first three bits (leftmost three bits) are always "**100**". Then 8 bits for opcode as indicated in **Figure 8**. The subsequent 5 bits serve as dest followed by 5 bits for src1. The second source operand is immediate 12-bit value. The instruction format is shown in **Figure 9**.

Instruction	Opcode
LDUR	10101010
STUR	10101011

Figure 8: Opcode for Category-4 instructions

100	opcode (8 bits)	srcdst (5 bits)	src1 (5 bits)	immediate_value (11 bits)

Figure 9: Format of Category-4 instructions with source2 as immediate value

2. Sample Input/output Files

Your program will be given a text input file (see sample.txt in project2 assignment webpage). This file will contain a sequence of 32-bit instruction words which begin at address **"64"**. The final instruction in the sequence of instructions is always a DUMMY instruction ("10100000000000000000000000000000"), and there will be only one such DUMMY instruction. Immediately after the DUMMY instruction, 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 ARM simulator (with executable name as **ARMsim**) 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). You can 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".*

ARMsim inputfilename.txt

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 project 2 assignment webpage.

1. The text (e.g., 0's and 1's) string representing the 32-bit instruction word.
2. The address (in decimal) of that instruction
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 (", "). See **disassembly.txt** for reference.

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 (*last line may have less than 8 data values*) >

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.

The project2 assignment webpage 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.

3. 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 whole project in only one source file. **Please add ".txt" at the end of your filename**. Your file name must be ARMsim (e.g., **ARMsim.c.txt** or **ARMsim.cpp.txt** or **ARMsim.java.txt** or **ARMsim.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 */. Please do not worry about the version numbers that gets added to your filename by eLearning if you upload the source file more than once. Also, please do not upload any executable or object file (**your full project should be only one source file**).
2. Please test your submission. These are the exact steps we will follow too.
    - Download your submission from eLearning (ensures that your upload was successful).
    - Remove ".txt" extension (e.g., ARMsim.c.txt should be renamed to ARMsim.c)
    - Login to any CISE linux machine (e.g., **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.cise.ufl.edu. To avoid this time waste, we strongly recommend that you should test your program on storm.cise.ufl.edu.
    - Please compile to produce an executable named **ARMsim**.
        - gcc ARMsim.c –o ARMsim **or** javac ARMsim.java **or** g++ ARMsim.cpp –o ARMsim **or** g++ -std=c++0x ARMsim.cpp –o ARMsim
    - 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 the correct/provided ones
        - ./ARMsim inputfilename.txt **or** java ARMsim inputfilename.txt **or** python3 ARMsim.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 (giving help or taking help in completing this project). We were able to establish violation in several cases - those students received "0" in the project and their names were reported to UF Dean of Students office. This year we will also impose one additional letter grade penalty. Someone could potentially lose two letter grade points (e.g., "A-" grade to "B" grade) – one for getting 0 score in the project and then another grade penalty on top of it. Moreover, the names of the students will also be reported to UF 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 repeated academic honesty violation (can lead to deportation for international students).*

4. Important Clarifications

- When we grade your project, we will use additional testcases (not only sample.txt) that you will not have access prior to your submission deadline. Therefore, please create your own testcases to test your implementation. It is okay if several students develop or share multiple new testcases (as long as it does not directly help in the implementation of the project). In other words, it is not cheating to share testcases (sample, disassembly and simulation files), but it is cheating if one student helps another student to debug/modify his/her project implementation irrespective of the circumstances (e.g., helping me to debug code because my code failed while running testcase developed by you – still cheating since it directly helped me in debugging the project).

- Please do not assume anything that we did not explicitly state in the project document. For example, we did not provide any information in terms of the number of instructions or memory locations (data values). It will depend on the new testcase that we will develop for grading the projects. The list of instructions will start with address "64" and continue until you reach the DUMMY instruction. The memory locations will start immediately after the DUMMY instruction and continue until the end of the file. Therefore, the address of the first memory location would be the address (PC value) of the DUMMY instruction + 4.

- Please do not hardcode the input file. Take it as a command line argument, so that we can invoke your program with new testcases e.g., ARMsim new.txt or ARMsim test.txt or ARMsim sample.txt

- Sample.txt did not include all the opcodes in Project 2 description. Please create your own testcases to test your implementation with other opcodes and other valid scenarios based on this document. We will not test any scenario that causes any types of exceptional behavior not covered by the project document.

- After you upload your submission in eLearning, please download and test it using the procedure outlined the Project 2 document. In some cases in the past, students argued that they uploaded it for sure, but it is not there in the eLearning website. Also, some students argued that it works in their machine (PC or MAC) but you will not get a grade unless it works on **storm.cise.ufl.edu**.

- Assume that all registers are initialized to 0. Note that X31 (XZR) is a read only register which is always 0.

- There is no entry for ORRI in ARM instruction set - it is ORR with immediate Operand2.

- Signed or unsigned operand comes from the ARM instruction set manual. All logical operations (e.g., ANDI, ORRI, EORI, ORR, EOR, LSR, LSL) treat their operands as unsigned numbers, whereas all arithmetic operations as well as branch offset (all other opcodes in Project 2) treat their operands as signed numbers. If an operand is a signed value, it should be treated as a 32-bit 2's complement number, otherwise the result will be incorrect. For example, you need to sign-extend correctly when you convert an 11-bit immediate value to a 32-bit number. On the other hand, if it is an 11-bit unsigned number, you should always extend with 0's. Since you are implementing a simulator (not an emulator), it is okay if you can produce correct result without sign-extend, etc. However, you have to carefully operate in decimal domain with signed and unsigned values depending on the opcode.

5. How to Implement – Where to Start?

Implementing and debugging may take a lot of time depending on your programming background. It is better to submit your project before the deadline, even if it is not complete or does not compile, since you may get partial points. If you do not submit anything, you will get 0 points - there will be no extension even if your project is almost ready and needs one more hour of debugging to make it perfect.

If you know exactly what to do, please ignore the following notes. If you are not sure where to start, here are few steps that may be helpful.

- Please choose between C, C++, Java or Python based on your comfort and expertise (no other language is allowed).
- Please browse through the project2.pdf to understand which instructions need to be implemented and their formats. I hope you have gone through LEGv8 Reference Card as well as ARM instruction set earlier so that you have rough idea about these instructions.
- Look at the first 32-bit binary in sample.txt, and try to manually decode it using the format in project2.pdf. If you do it right, it should match with disassembly.txt. Once you are comfortable decoding 32-bit instructions, please write a program to decode binaries in sample.txt. Of course, your decode function should be able to decode any instructions (with any dest/src operands) in the project2.pdf (all of them are not in sample.txt, but they may appear in the new testcase that we will use for grading your submission).
- Once you are done decoding (or as you decode one instruction), please print the decoded instructions in disassembly.txt. Compare (diff -w -B) to make sure it is identical to sample_disassembly.txt
- For each disassembled instruction, figure out whether it modifies any registers or memories. Check the sample_simulation.txt to make sure your understanding is correct. Once you are comfortable, please write a program to simulate each of the disassembled instructions, and print the results in simulation.txt. Compare (diff -w -B) to make sure it is identical to sample_simulation.txt.