Project 1

CDA 4630/CDA 5636: Embedded Systems

Total: **10** points   **Due**: February **28**, 2024   **11:30** pm

This is an **individual** assignment. You are not allowed to take or give any help in completing this project. Please **strictly follow the submission instructions** (outlined at the end of this document) and submit your source code in eLearning website before the deadline. Late submission (by email attachment to umasaisree.avula@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. Please include the following sentence on top of your source code as a comment:

> "**I have neither given nor received any unauthorized aid on this assignment**".

Petri Net Simulator for a Simple Processor: In this project, you will create a Petri Net simulator for a simple processor. The model will use colored tokens (token with values) rather than the default Petri net. Your simulator should generate step-by-step simulation of the Petri net model of the processor described below. You can use **C**, **C++**, **Java**, or **Python** to implement this project. Please go through this document first, and then view the sample input/output files in the project assignment.
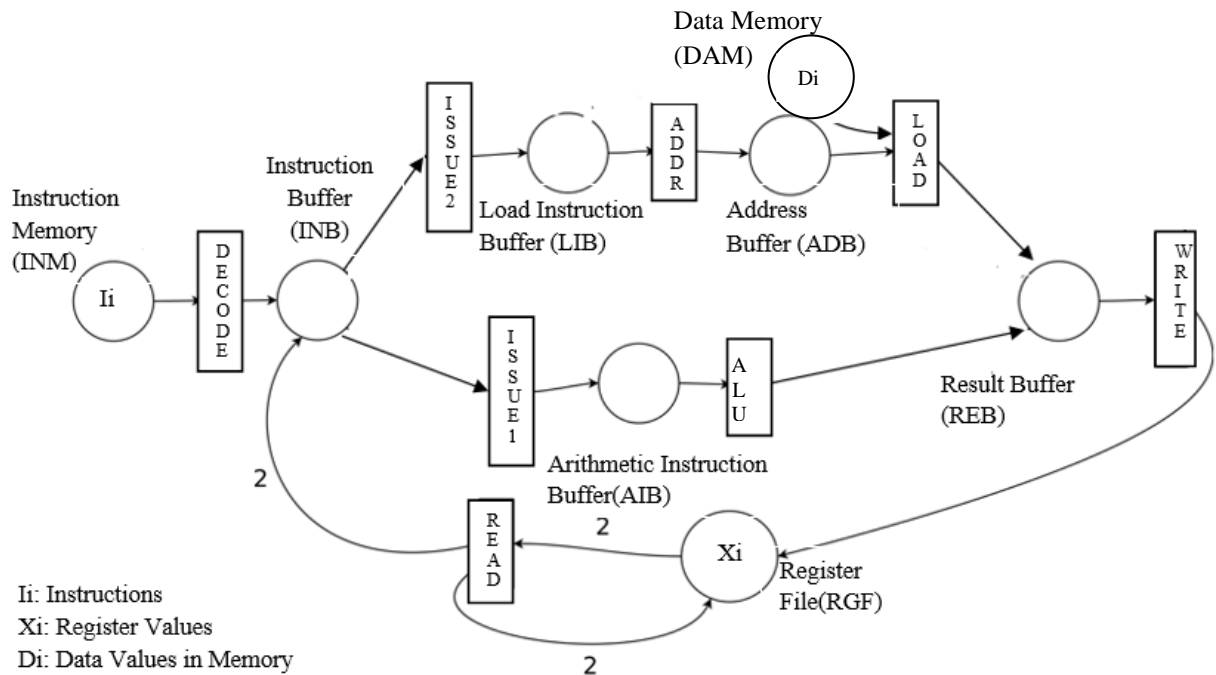

Figure 1. Petri Net Model of a MIPS Processor

Figure 1 shows the Petri net model. Here, circles are places and rectangles are transitions. We first describe three important places (Instruction Memory, Register File, and Data Memory) of the Petri net model. Next, we describe eight transitions. The remaining places can be viewed as buffers. Each arc carries (consumes) one token unless marked otherwise. **You do not have to worry about any hazards or exceptional scenarios. We will not provide any testcase that can produce any hazards or exceptions.**

THREE IMPORTANT PLACES

1. Instruction Memory (INM):

The processor to be simulated only supports five types of instructions: *add* (ADD), *subtract* (SUB), *logical and* (AND), *logical or* (OR), and *load* (LD). At a time step, the place denoted as Instruction Memory (INM) can have up to 16 instruction tokens. This is shown as **Ii** in Figure 1. We will provide an input file (instructions.txt) with up to 16 instruction tokens. It supports the following instruction format. Please note that both source operands are always registers.

<Opcode>, <Destination Register>, <First Source Operand>, <Second Source Operand>

Sample instruction tokens and equivalent functionality are shown below:
<ADD, R1, R2, R3>	➔ R1 = R2 + R3
<SUB, R1, R2, R3>	➔ R1 = R2 – R3
<AND, R1, R2, R3>	➔ R1 = R2 & R3
<OR, R1, R2, R3>	➔ R1 = R2	R3
<LD, R1, R2, R3>	➔ R1 = DataMemory[R2+R3]

2. Register File (RGF):

This processor supports up to 8 registers (R0 through R7). At a time step, it can have up to 8 tokens. The token format is <registername, registervalue>, e.g., <R1, 5>. This is shown as **Xi** in Figure 1. We will provide an input file (registers.txt) with 8 register tokens that you can use to initialize the registers. You can assume that the content of a register can vary between $0 – 63$.

3. Data Memory (DAM):

This processor supports up to 8 locations $(0 – 7)$ in the data memory. At a time step, it can have up to 8 tokens. The token format is <address, value>, e.g., <6, 5> implies that memory address 6 has value 5. This is shown as **Di** in Figure 1. We will provide an input file (datamemory.txt) with 8 data tokens that you can use to initialize the data memory locations. You can assume that the content of a data memory location can vary between $0 – 63$.

EIGHT TRANSITIONS

1. READ:

The READ transition is a slight deviation from traditional Petri net semantics since it does not have any direct access to instruction tokens. Assume that it knows the top (in-order) instruction in the Instruction Memory (INM). It checks for the availability of the source operands in the Register File (RGF) for the top instruction token and passes them to Instruction Buffer (INB) by replacing the source operands with the respective values. For example, if the top instruction token in INM is <ADD, R1, R2, R3> and there are two tokens in RGF as <R2,5> and <R3,7>, then the instruction token in INB would be <ADD, R1, 5, 7> once both READ and DECODE transitions are activated. Both READ and DECODE transitions are executed together. Please note that when READ consumes two register tokens, it also returns them to RGF in the same time step (no change in RGF due to READ).

2. DECODE:

The DECODE transition consumes the top (in-order) instruction (one token) from INM and updates the values of the source registers with the values from RGF (with the help of READ transition, as described above), and places the modified instruction token in INB.

3. ISSUE1:

The ISSUE1 transition consumes one arithmetic/logical (ADD, SUB, AND, OR) instruction token (if any) from INB and places it in the Arithmetic Instruction Buffer (AIB).

4. ISSUE2:

The ISSUE2 transition consumes one load (LD) instruction token (if any) from INB and places it in the Load Instruction Buffer (LIB).

5. Arithmetic Logic Unit (ALU)

The ALU transition performs arithmetic/logical computations as per the instruction token from AIB, and places the result in the result buffer (REB). The format of the token in result buffer is same as a token in RGF i.e., <destination-register-name, value>.

6. Address Calculation (ADDR)

The ADDR transition performs effective (data memory) address calculation for the load instruction by adding the contents of two source registers. It produces a token as <destination-register-name, data memory address> and places it in the address buffer (ADB).

7. LOAD:

The LOAD transition consumes a token from ADB and gets the data from the data memory for the corresponding address. Assume that you will always have the data for the respective address in the data memory in the same time step. It places the data value (result of load) in the result buffer (REB). The format of the token in result buffer is same as a token in RGF i.e., <destination-register-name, data value>.

8. WRITE

The WRITE transition transfers the result (one token) from the Result Buffer (REB) to the register file (RGF). If there are more than one token in REB in a time step, the WRITE transition writes the token that belongs to the in-order first instruction.

Command Line and Input/Output Formats:

Command Line: The simulator should be executed with the following command line. Assume that the input files (with the following names) will be available in the same directory. You are also expected to produce the output file in the same directory.

./Psim or **java Psim** or **python3 Psim.py**

Please hardcode the input and output files as follows:

       Instructions (input): instructions.txt
       Registers (input): registers.txt
       Data Memory (input): datamemory.txt
       Simulation (output): simulation.txt

File Formats:

We will provide inputs in the specific format as listed below:

Input Register File Format: (see registers.txt for example)

```
<register name,value>
…
```

Input Data Memory File Format: (see datamemory.txt for example)

```
<memory address,data value>
…
```

Input Instruction Memory File Format (see instructions.txt for example):

```
<opcode,dest,src1,src2>
…
```

Step-by-step Snapshot Output File Format (see simulation.txt for example): **Please note the following comments are not part of the output format**.

```
STEP 0:
INM: I1,I2,I3,…        # Where Ii are comma separated instruction tokens.
INB:                   # Comma separated tokens with source values.
AIB:                   # Comma separated arithmetic instruction tokens
LIB:                   # Comma separated load instruction tokens
ADB:                   # Comma separated address tokens
REB:                   # Comma separated result buffer tokens
RGF:RF1,RF2,…          # Comma Separated register file tokens.
DAM:D1,D2,…            # Comma Separated data memory tokens.
<blank_line>
STEP 1:
…
```

Continue until the end of simulation. End of simulation is determined when none of the transitions can be fired in a time step.

Additional Notes: (refer to sample input and output files for ease of understanding)

- STEP 0 values represent initial states of all the places. STEP 1 represents the tokens of all the places at the end of first time step. In general, STEP **i** values should reflect the tokens of all the places at the end of time step **i**. In each time step, you are supposed to execute each transition exactly once (if it has required input tokens at the beginning of that cycle).

- When there are more than one tokens in a place, please print them in instruction order (**in-order**) except for DAM and RGF. The token for DAM and RGF should be printed in the sorted order based on memory address or register name (starting with the smallest), respectively.

Submission Policy:

Please follow the submission policy outlined below. There will be up to **20% score penalty** based on the nature of submission policy violations. 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.

1. Please develop your project in one source file since the submission website accept only one source file. **Please add ".txt" at the end of your filename**. Your file name must be Psim (e.g., Psim.c.txt **or** Psim.cpp.txt **or** Psim.java.txt **or** Psim.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" using the comment feature of that language.

2. Please test your submission. These are the exact steps we will follow too.
    o   Download your submission from eLearning (ensures your upload was successful).
    o   Remove ".txt" extension (e.g., Psim.c.txt should be renamed to Psim.c). **We know that eLearning adds a number at the end of the file if you submit multiple times.** My script will take care of it (so do not worry about that number inserted by eLearning).
    o   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 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.
    o   Please compile to produce an executable named **Psim**.
        ▪   gcc Psim.c –o Psim **or** javac Psim.java **or** g++ Psim.cpp –o Psim   **or** g++ -std=c++17 Psim.cpp -o Psim
    o   **Please do not print anything on screen.**
    o   Execute to generate simulation file and test with the correct one.
        ▪   ./Psim **or**   java Psim **or** python3 Psim.py
        ▪   diff –w –B simulation.txt correct_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.** 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). This year, I will include one grade level penalty (in addition to getting 0 in project 1), e.g., if your final grade is B, it will be B-. If the department of university impose additional penalty (e.g., getting a failing grade in the class) that will be imposed. 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).

Grading Policy

The project assignment has links to the sample input and output files. Correct handling of the sample input (with possible changes of register and data values) will be used to determine 60% of credit awarded. **The register and data memory values will be between 0 and 63**. The remaining 40% will be determined from other input test cases that you will not have access prior to grading. The other test cases can have different number or order of instructions as well as different register and data memory tokens. It is recommended that you construct your own sample input files with which to further test your simulator. **Note that the new test case will NOT test any hazards, exceptions, or scenarios that are not described in this document**.

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

Please ignore this section if you know how to use CISE servers to run programs.

1. Register for a CISE account using your Gatorlink account and password.

    https://register.cise.ufl.edu/

2. Download PuTTY or similar shell to run your project using Linux. If you are using MAC, you can use the Terminal client.

3. Download WinSCP or similar file transfer mechanism. You can use sftp command from your MAC terminal client.

4. 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.

5. 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).

6. 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).

7. 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.