PROGRAMMING MIDTERM
Total Time: 3 Hours

Important Notes:

- The Lab. 1 handout is included as a baseline; this explains the baseline MIPS pipeline that we will be simulating.
- Similarly, we have also included the MIPS Greensheet for your reference.
- For each question, start with the baseline code provided for that specific question's folder.
- The questions are NOT cumulative. For example, the solutions/code for Q2 does not have to include solutions for Q1. For each question, *do only what is required.*

Important Note: One of the statements in the Lab 1 solutions had a misleading comment. We have fixed the comment in the sample codes provided, i.e., the line of code has changed from:

```
newState.MEM.ALUresult = 0; //case of branch
```

to:

```
newState.MEM.ALUresult = 0; //default
```

Q1. [5 Points] The "Branch Taken Rate" (BTR) of a program is the fraction (between 0 and 1) of branches that are taken (# Taken Branches/Overall Branches). Recall that a Branch is Taken if we jump to the Branch Target instead of PC+4.

For the baseline Lab 1 simulator, print our the BTR to the command prompt.

To do: Update the baseline C++ code provided in the Q1 folder to output the BTR to the command prompt. The code already has a “cout” statement that initializes a *float* variable called “*branch_taken_rate*” and at the end of the simulation, prints it out to the command prompt.

Sample input files are provided. Running the baseline C++ code on these files will produce corresponding sample output files —- note, your updates to the baseline code should not change the sample output files. The only thing that will change is the BTR value printed to the command line.

Q2. [10 Points] BEQ Done Differently

In this question, you will update the pipeline to resolve BEQs in the EX stage instead of in the ID/RF stage. Recall from the Lab 1 handout that our BEQs actually act like BNEs. We will continue assuming this!

To do so, you will assume the following:

- Branches are always assumed to be NOT TAKEN. That is, when a BEQ is fetched in the IF stage, the PC is speculatively updated as PC+4 in the next clock (as in Lab 1), and PC+8 in the clock cycle after that. That is, the next 2 instructions are speculatively fetched!
- For BEQ instructions, the ID/RF stage passes on *read_data1*, *read_data2* to the EX stage instead of comparing *read_data1* and *read_data2* in ID/RF as we did in Lab 1.
- Also, ID/RF passes on *imm* to the EX stage for address computation instead of performing this computation in ID/RF as in Lab 1.
- In EX stage, *read_data1* and *read_data2* are compared and the effective address is computed.
- If the branch is NOT TAKEN, execution proceeds normally. However, if the branch is TAKEN, the speculatively fetched instructions from PC+4 and PC+8 are quashed in the ID/RF and EX stages,

respectively, and the next instruction is fetched from the effective branch address. Execution now proceeds normally.

- You can assume that our testbenches will ***not*** have RAW hazards where the beq is a dependent instruction. That is, the values that *beq* reads from the RF can be assumed correct.
- You can also assume that the 2 instructions immediately after a *beq* will never be a *halt.*

To do: Update the baseline C++ code provided in the Q2 folder to implement BEQ resolution in the EX stage as described above. Sample input files and corresponding sample output files are provided in the same folder.

Note that the baseline C++ code provided prints out 'X' for state results that we will not be grading (your code can output X or 0/1 for these). The state variables that are don't cares have been updated compared to Lab. 1 to reflect the fact that BEQ is being executed in the EX stage.

Q3 [10 Points] Registered Jump Instruction

In this question, you will update the pipeline to add support for the *jr* instruction. The format of the *jr* instruction is shown below and its semantics can be found in the MIPS GreenSheet.

Instruction	**Format**	**Op Code**	**Funct. (Hex)**
jr	R-Type	00 0000	08

The *jr* instruction is decoded and resolved in the ID/RF stage by computing its target address using the value of register rs (*read_data1*). In the mean time, an instruction from PC+4 is fetched speculatively.

In the next clock cycle, the PC is set to the target address, and at the same time, the speculatively fetched instruction is quashed in its ID/RF stage (this happens regardless of whether the target address is PC+4 or not.)

You can assume that our testbenches will ***not*** have RAW hazards where the *jr* is a dependent instruction. That is, the values that *jr* reads from the RF can be assumed correct.

You can also assume that the instructions immediately after a *jr* will never be a *halt.*

To do: Update the baseline C++ code provided in the Q3 folder to implement the *jr* instruction. Sample input files and corresponding sample output files are provided in the same folder.

Note that the baseline C++ code has been updated to print out the relevant state results for *jr* instructions in each stage. The code prints out X for state results that we will not be grading (your code can output X or 0/1 for these).

Note that we chose to print out read_data1 in the EX stage for jr instructions, although is not actually necessary since the jr is resolved in ID/RF. Therefore, in your implementations, please make sure to correctly pass on read_data1 from ID/RF to EX even for *jr* instructions.

WHAT TO SUBMIT:

Please submit *ONLY* your C++ codes in a single zipped folder on NYUClasses. The folder should contain

MIPS_pipeline_base_code_Q1.cpp
MIPS_pipeline_base_code_Q2.cpp
MIPS_pipeline_base_code_Q3.cpp

If for any reason you depart from this, please provide a README.txt file in your folder that explains how to run your code.