Project Specification

In this project, you will work in groups of 1-2 students to implement a Multi Cycle Pipeline simulator.
The maximum students per group will be 2.

Deliverables:

You can use one of the following languages to implement the program:

- C
- C++
- Python
- Java
- Verilog
- VHDL

You will also need to:

- Write clean code
- Make sure your code is readable even for someone not versed in the particular language
- Comment your code where necessary (too much commenting of code sometimes makes it hard to read)
- **<u>Submit a user manual for your code detailing how to run it; step by step instructions as well as what OS to use and how to compile the source code.</u>**
    - Your instructor has access to the following OS:
        - a Mac OS Catalina (unix) platform
        - a Linux via linux.gl.umbc.edu
        - Windows 10
    - I would suggest you test your code on one of these platforms to ensure it runs as you expect.
- Make sure that it is possible to run your code without needing to purchase any additional software. (I count Trials as a purchase)

You will need to submit a zip file named lastname_firstname_project.zip containing:

- The program files and source code.
- A text file (README) detailing how to run your simulator and recompile it if needed.
- **Only one submission per group. You will need to make sure all your names are shown in the README as well as in the program code.**

Your program should take as an input a text file with MIPS code (see Instructions section), and output a table similar to one used in class showing:

- When each instruction completed each stage
- The values in **<u>all the</u>** registers at the end of execution
- The values in **<u>memory</u>** at the end of execution
- See input and output sections for further details

Due Date:
December 12th 2021 at 11.59pm

Grading:

The code and program will be graded according to the following:

10% -if the program works. It does not have to be perfect for you to get this credit

50%- Implementation
- How close is your simulator to mimicking a true scoreboard
- How well does it meet the specifications in this document
- Do values in the register match what is expected
- How does it handle corner cases

40%-Well written code:
- use of good variable names
- code is easy to follow an understand and is well organized
- Good instructions on how to run/compile your code

Late submissions will incur a 10 point deduction for every 6 hours they are late.

This project follows the UMBC Academic Integrity Policy and any work submitted must be your own. In addition to that policy, Collaboration between teams should only involve discussion of ideas but no sharing of code.
Any work copied from online resources MUST be cited.

Processor Configuration:

The algorithm will model a processor with the following configuration:

- Pipelined FP adder/s for FP adds and subtracts -2cycles
- Pipelined FP multiplier/s for FP multiplies -10cycles
- Pipelined FP divider/s for FP divides -40cycles
- Integer Unit/s for Loads, Stores, Integer Adds and Subtracts -1cycle

32 FP registers and 32 integer registers.

- <u>All registers will be initialized to 0 at the beginning.</u>

Full forwarding is possible and it is possible to write and read the register in the same cycle.

Control Instructions:

We will assume a non correlating 1 bit predictor (0,1) for the branch instructions in the simulator. All branches, conditional and unconditional, will assume *"taken"* to start and the prediction per branch will be updated when resolved only if it is wrong. Branch resolution will happen in the execute stage. If the prediction was wrong then all instructions fetched after the branch will seize execution and the correct instruction will be fetched on the corresponding cycle. The prediction information should also be updated at this point. If the prediction is correct then the pipeline will continue execution without any interruptions.
The label for control instructions will always be the first word on the line and end in a colon, for example: "*Loop:"*

Data Memory:

The data memory hierarchy will be one level 1 cache and a main memory.
Data memory is not pipelined, thus you cannot have more than one instruction in the mem stage.

L1 Data Cache:
The L1 cache will be empty at the start of the simulator. It will have four blocks and will take a direct mapped approach for block placement and a write back strategy.
Reads from the L1 cache will take 1 cycle for a hit, however, if there is a miss, then a read to main memory will take 2 cycles for a total of 3 cycles (first cycle for the search in cache and 2 cycles for search in main memory). In the case of a miss, the data will be brought into the cache in addition to bringing it into the pipeline.
A miss will lead to stalls.
In addition to data, each block brought into the cache should have information regarding the block's original address in main memory. This will be used to track whether blocks exist in the cache for a hit or miss and where to write the data back.

Main Data Memory:
You can initialize your data memory to the following values:

Memory Location	Value in Memory
0	45
1	12
2	0
3	92
4	10
5	135
6	254
7	127
8	18
9	4
10	55
11	8
12	2
13	98
14	13
15	5
16	233
17	158
18	167

The memory location will be the index of the value and we will use that index as the actual address for loads and stores.
For simplicity the value at the memory location is what gets loaded into a register and if the memory location is offset then it loads whatever value is at that offset.
Example:
LD F2, 1(17) would load 167 into register F2

In some cases, the address will come from an integer register for example:
```
LI   $2, 17
LD F2, 0($2)
```

This sequence would load 158 into register F2. The addresses wrap back around so, address 19 would be location 0 in main memory.

Instructions:

You can program your algorithm for the following instructions:

Memory Instructions:

L.D Fa, Offset(addr)	Load a floating point value into Fa
S.D Fa, Offset(addr)	Store a floating point value from Fa
LI $d, IMM64 -Integer Immediate Load	Load a 64 bit Integer Immediate into $d
LW $d, Offset(addr)	Load an integer value into $d
SW $s, Offset(addr)	Store an integer from $s

ALU Instructions:

ADD $d, $s, $t - Integer add	$d = $s + $t
ADDI $d, $s, immediate – Integer Add with Immediate	$d = $s + immediate
ADD.D Fd, Fs, Ft – Floating Point Add	Fd = Fs + Ft
SUB.D Fd, Fs, Ft – Floating Point Subtract	Fd = Fs - Ft
SUB $d, $s, $t -Integer Subtract	$d = $s - $t
*MUL.D Fd, Fs, Ft -Floating Point Multiply	Fd = Fs X Ft
DIV.D Fd, Fs, Ft – Floating Point Divide	Fd = Fs ÷ Ft

*You can assume that Fd is big enough to hold the value of the result.

Control Instructions:

BEQ $S, $T, OFF18 - Branch to offset if equal	IF $S = $T, PC += OFF18±
BNE $S, $T, OFF18 - Branch to offset if not equal	IF $S ≠ $T, PC += OFF18±
J ADDR28 - Unconditional jump to addr	PC = PC31:28 :: ADDR28∅

$s/$d/$t are integer registers and Fa/Fd are floating point registers

Inputs

Your program should take the following inputs (how it takes them is up to you):

1. Text file with a program written using MIPS instructions outlined above
    - You do not have to account for MIPS instructions not outlined in this document

Outputs

Your program should output the following:

1. The pipeline showing the different stages for the different instructions (similar to how we have been doing it in class).
    - For the the different execute stages you should use the following:
        i. FP ADD/SUB: A# (where # stands for number)
        ii. FP DIV: D# (where # stands for number)
        iii. FP MUL: M# (where # stands for number)
        iv. Integer Units: E (this includes branches)
2. Final values in the FP and Integer registers as well as the memory locations.