# Project 1 - Integer Pipeline - ECE463 - Luke Brown

## DATA STRUCTURES:

For the project I mainly used 2 data structures to represent some main parts of the code representation of the pipeline datapath. The data structures I used are an unsigned integer array and a struct that I created to represent the pipeline registers. The first data structure I used was an unsigned integer array to represent the 32 registers. I declared it as an unsigned regFile[NUM_GP_RESGISTERS]. I named it regFile as it represents the register file that would actually be used and I used the #define of NUM_GP_REGISTERS so that if some end user wanted to change the number of registers available for use they would not need to find my declaration and edit it they can just change the #define and the simulator will use that. The second data structure I used was a struct I created to represent the four pipeline registers called pipeReg. The struct had nine members that represented the IR using the instruction_t struct provided for instructions and the pc, npc, a, b, imm, ALUOut, cond, and LMD values with unsigned integers. I used the struct to create four pipeline registers and named them IFID, IDEX, EXMEM, MEMWB to represent the pipeline registers between each of the stages.

- Description of your approach to handle data, control and structural hazards;

## HAZARD HANDLING:

### DATA HAZARDS:

The only data hazard applicable to this pipeline is the Read-After-Write(RAW) hazard. The Write-After-Write(WAW) and Write-After-Read(WAR) are not applicable because the in order pipeline deletes the chances of WAR happening and the fact that all EXE stages take the same number of cycles deletes the chances of WAW happening. The way I handle RAW hazards in simple terms is to check the IFID pipeline register instruction and if either of the source(src1, src2) variables match the destination of the instruction before it then it sends forward two stalls in the form of NOPs and if either source matches the destination of the instruction two instructions before the current instruction then it stalls once so that the previous instruction can write the value before it reads preventing errors. The way I accomplish this in code is to check the values of IFID.IR.src1 and .src2 against the values of EXMEM.IR.dest and MEMWB.IR.dest. The reason I checked EXMEM and MEMWB instead of EXMEM and IDEX registers is because the simulator I wrote works backwards doing WB first and clearing it then MEM, EX, ID, IF. I did this so that the information in the later stages is not overwritten by the information in earlier stages moving forward. For example, if IF wrote to the IFID register before ID used the information in there from the previous instruction it would destroy/overwrite the information. If the destination matches either of the source, or certain instructions like LW only have one source so I only compare the one, then the ID backs the PC up in the program counter register(I called this PCR) to refetch the same instruction and then passes forward NOPs. I accomplished this using a specific version of the instruction_t struct provided where the opcode was set to NOP and everything else was set to empty or undefined. I called this special NOP instruction NOPI so that I could just set the entire IR for the passed instruction to NOPI. I did however later in the WB stage have to make sure to not count NOP in my instruction count(iCount in my program). The way I made sure stalls and NOPs got counted twice was by making one large if case that checked both if the instruction just before and the instruction two before the current instruction had RAW with the current instruction. What this caused is when an instruction came through directly before

an instruction causing RAW it would be counted once then in the EXMEM register and then once when it was in the MEMWB register which counted it twice and produced two NOPs. However, if the instruction causing RAW was only two before the current instruction meaning it would need one stall it would only trigger that if statement once and only produce one stall.

**CONTROL HAZARDS:**

The main control hazards I had to handle in this project are the branch instructions. As per the project specifications, branches cause the simulator to continue fetching the same instruction until the branch is evaluated and either taken or not. The way I achieved this was by writing over the instructions that have been stored into the IFID and IDEX stage registers with two NOP instructions and adding the stalls to the count accordingly. I also adjust the PCR so that it does not move forward while the branch is being handled. I handle all this in the EX stage when a branch instruction comes in. When the branch is executed, it always creates two stalls before the program can proceed while it waits for the branch to be evaluated. The way I make sure PCR does not move forward is by checking if the EXMEM.cond != UNDEFINED as the only way cond is changed from undefined is if there is a branch instruction. If this is true then I halt the PC.

**STRUCTURAL HAZARDS:**

Structural hazards are introduced in testcase5 for the integer pipeline with the introduction of a memory latency of 4. This causes the program to want to use the memory stage while its still being used by the previous memory instruction. The MEM stage is busy so the entire program need be halted until it is ready. The way I achieve this is with an if statement and a new flag I created called memFlag. memFlag goes is set to the data_memory_latency + 1. I then have an if statement with the IF, ID, and EXE stages inside of it. If the memFlag > 1 then I know memory is still being handled and these stages still need to be stalled. So the IF, ID, and EXE stages do not happen but clock cycles and stalls are incremented while memFlag is decremented to represent the number of cycles left before the stage is done. The reason I add 1 to the latency is so that I can use 0 as a sort of off state where if a memory instruction gets into the MEM stage it checks to be sure memFlag is off or 0. Then if it is off the memory stage realizes this is a new memory instruction and it needs to turn memFlag on and then it counts down to 1 and on memFlag=1 the MEM instruction like LW or SW moves forward and the if statement evaluates to true allowing the other stages to occur and the whole simulator begins moving forward again. The else case on the if also has the memFlag being set to 0 or off so that the simulator doesn't get infinitely stuck in a memory stall.

- Brief explanation of what works and what not in your implementation;

## WHAT WORKS AND WHAT DOESN'T:

**WHAT WORKS:**

Test Cases 1-5 work and report properly. The program can handle the RAW hazards, Control hazards from branches in test case 4-5, and Structural hazards in test case 5. I feel like I could handle the structural cases a bit better or more efficiently but they do work properly currently. It prints out all of the correct information in the correct order for all tests 1-5. I have been using CLion to code as it offers nice

features but have been uploading to grendel after every test I figure out and making sure the project compiles and runs properly there too. It has been compiling properly and outputting correctly.

**WHAT DOESN'T WORK:**

Test case 6 does not work. I compile and run properly on both CLion and Grendel but after the first few clock cycles the outputs become incorrect and the final clock cycles, instruction count, stalls, and IPC are incorrect. I believe it is something to do with the way I handle branches. In test 4-5 branch is followed by an EOP instruction which is a special case while in test case 6 the branch instructions are not followed by EOP but by other regular instructions. I am very curious to see the solution to the project to understand how I could have fixed this.

## MAKE FILE:

I did not alter the Makefile in any way.

## TESTCASE SCREENSHOTS:

```
[lpbrown2@grendel31 c++]$ diff mytest1.txt testcases/testcase1.out
[lpbrown2@grendel31 c++]$ ./bin/testcase2 > mytest2.txt
[lpbrown2@grendel31 c++]$ diff mytest2.txt testcases/testcase2.out
[lpbrown2@grendel31 c++]$ ./bin/testcase3 > mytest3.txt
[lpbrown2@grendel31 c++]$ diff mytest3.txt testcases/testcase3.out
[lpbrown2@grendel31 c++]$ ./bin/testcase4 > mytest4.txt
[lpbrown2@grendel31 c++]$ diff mytest4.txt testcases/testcase4.out
[lpbrown2@grendel31 c++]$ ./bin/testcase5 > mytest5.txt
[lpbrown2@grendel31 c++]$ diff mytest5.txt testcases/testcase5.out
[lpbrown2@grendel31 c++]$ ./bin/testcase6 > mytest6.txt
```

Test Cases 1-5 working. Diff command not producing anything so my test is the same as the correct out provided

```
[lpbrown2@grendel31 c++]$ diff mytest5.txt testcases/testcase5.out
[lpbrown2@grendel31 c++]$ ./bin/testcase6 > mytest6.txt
[lpbrown2@grendel31 c++]$ diff mytest6.txt testcases/testcase6.out
161c161
< PC = 268435484 / 0x1000001c
---
> PC = 268435480 / 0x10000018
163,164d162
< NPC = 268435484 / 0x1000001c
< Stage: EX
166c164
< IMM = 8 / 0x8
---
> Stage: EX
179c177
< PC = 268435484 / 0x1000001c
```

Test case 6 not working. Diff is producing output.(There is more output from diff for test case 6 this is just showing that 6 doesn't work)