Chuck Richardson

ECE 563

Dr. Michela Becchi

2/21/2023

**Project 1-Floating-Point Pipeline Report**

# **Objective**

The objective of this project was to implement a 5-stage MIPS floating-point pipelined processor in c++. This pipelined processor contains five stages including Fetch(IF), Decode(ID), Execute(EXE), Memory(MEM), and Writeback(WB). This design also includes 32 integer general-purpose registers(R0-R31) as well as 32 floating-pointer general-purpose registers (F0-F31). This MIPS processor needed to handle basic integer register arithmetic instructions such as ADD, SUB, XOR, as well as memory operations such as LOAD and STORE. For the floating-point operations, it needed to handle instructions such as basic floating point arithmetic operations such as ADDS, SUBS, MULTS, and DIVS and memory operations such as LWS and SWS. This pipelined processor also needed to handle control/branching instructions such as BEQZ, BNEZ, BLTZ, etc. This pipelined processor also had to handle hazards such as structural hazards pertaining to resource conflicts, data hazards pertaining to dependencies between instructions, and control hazards pertaining to branches and other control flow instructions. During this project the following metrics were kept track of such as the IPC(Instructions per Clock Cycle), Total Clock Cycles, Total Instructions, and Number of Stalls.

# **Data Structures**

The MIPS pipelined processor needed special purpose pipelined registers, integer general-purpose registers, floating-point general-purpose registers, and an instruction pipelined register. For the integer general-purpose registers, I used an array with the size of the number of general-purpose registers. These general-purpose registers have a struct of type int\_file\_t, in which it contains two integer values to hold the value and to check if it is busy being written to. This is shown in Figure 1.

Text

Description automatically generated

Figure 1: Integer General-Purpose Register Implementation

For the floating-point general-purpose registers, I used an array with the size of the number of general-purpose registers. These general-purpose registers have a struct of type fp\_file\_t, in which it contains one floating-point value to hold the value and an integer value to check if it is busy being written to. This is shown in Figure 2.

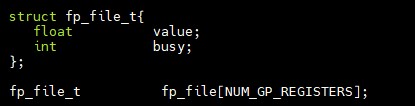


Figure 2: Floating-Point Register Implementation

For the special purpose pipelined registers, I had to use a two-dimensional array in which the first index was the number of stages and 2nd index was the number of special purpose registers. This is shown in Figure 3.



Figure 3: Special Purpose Registers Implementation

For the instruction pipelined register, I used an array with the size of the number of stages to hold the instructions for each collective stage. This instruction register array has struct of type instruction\_t to hold the opcode, src1, src2, etc. This is shown in Figure 4.



Figure 4: Instruction Pipelined Register Implementation

# **Hazard Handling**

There are three types of hazards to be handled in the MIPS pipeline including structural hazards, data hazards, and control hazards.

## **Data Hazards**

## **RAW Hazards**

In the MIPS Pipeline there exists data dependencies between instructions. For RAW(Read-After-Write) Hazards we must stall the decoding of the instruction until the value for a specified register that is needed to be written back has completed in the writeback stage. We check if the general-purpose registers are busy being written to and if they are then we stall. Once the value is written back, the instruction from decode stage can then be decoded and propagated to the execute stage. This is shown in Figure 5.

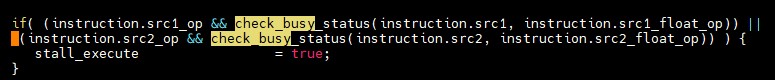


Figure 5: Data Hazard Checking

## **WAW Hazards**

For WAW(Write-After-Write) Hazards, I first check if the destination register in the decode stage and the destination register in the execute stage are being used or/are going to be presently in use. Then following that I check if the two instruction destination registers are equal. Then I check if the latency of the current instruction will be less than or equal to the instruction inside the execute units’ current latency, thus determining if the instruction in the decode stage will finish before the instruction being executed. All these factors combined I determine if there will be a WAW Hazard. This is shown in Figure 6.

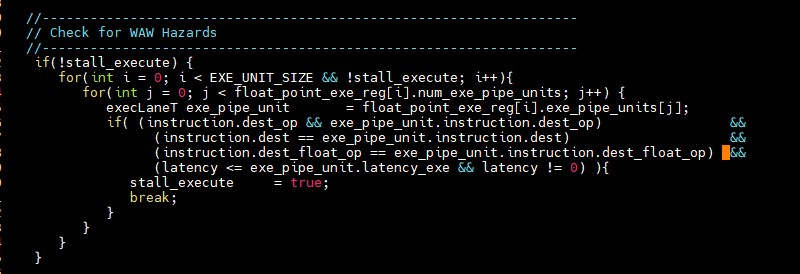


Figure 6: WAW Hazard Checking

## **Control Hazards**

Control Hazards are caused by conditional and unconditional branching such as BEQZ,BLTZ, JUMP, etc. This detection happens in the decode stage. This will also change what instruction is being fetched, whether the branch condition is true or false. At the decode stage we will check if the instruction being executed at the execute stage is a branch if it is then we stall the decode stage as well as the fetch stage. Then once the target of the branch is calculated we will then execute the next instruction whether it be the instruction at the target of the branch or the next instruction in program order. This is checking is shown in Figure 7.

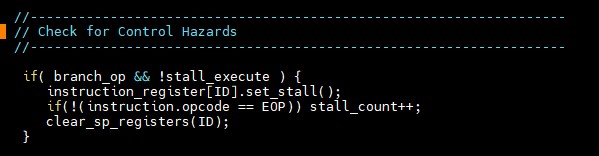


Figure 7: Control Hazard Checking

## **Structural Hazards**

## **Memory Latency**

One source of a Structural Hazards is memory latency. This issue is caused whenever a load or store instruction reaches the memory stage. If the instruction at the memory stage is a load or a store the other stages will be stalled such as IF, ID, and EXE until the memory stage has been completed its operation. Note we also must stall the Writeback Stage in this case due to the ordering of our stages in the run function. The implementation of data memory latency and stalls are shown in Figure 8.

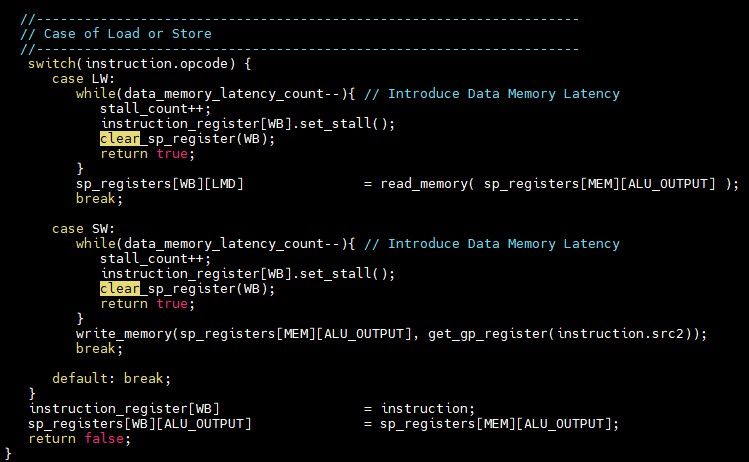


Figure 8: Structural Hazard Implementation

## **Floating-Point Execute Structural Hazard**

Another source of a structural hazard might occur due to execute unit contention. For example, say let’s say we have a series of instructions shown below:

DIV F1 F2 F3

DIV F3 F4 F5

These series of instructions might cause contention for the floating-point divider since they both need to use the divider during the execute stage, so therefore we will have to stall the execution of the second instruction until the divider finishes executing the first instruction. This checking is shown in Figure 9.

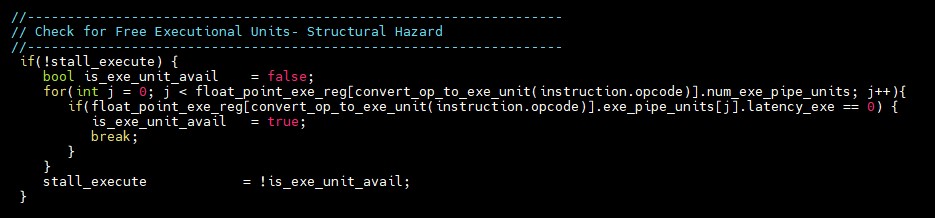


Figure 9: Structural Hazard at Execute Checking

# **Conclusion**

In conclusion, my MIPS Pipeline works for all testcases. I used Clion as well as valgrind to debug possible issues such as incorrect values for each respective stage and segmentation faults. My implementation for checking hazards works extremely well as well and passes all testcases. Shown below in Figure 10, is a sample of my output from testcase6 for the integer pipeline. Shown in Figure 11 is my output from my floating-point pipeline for testcase\_fp5.

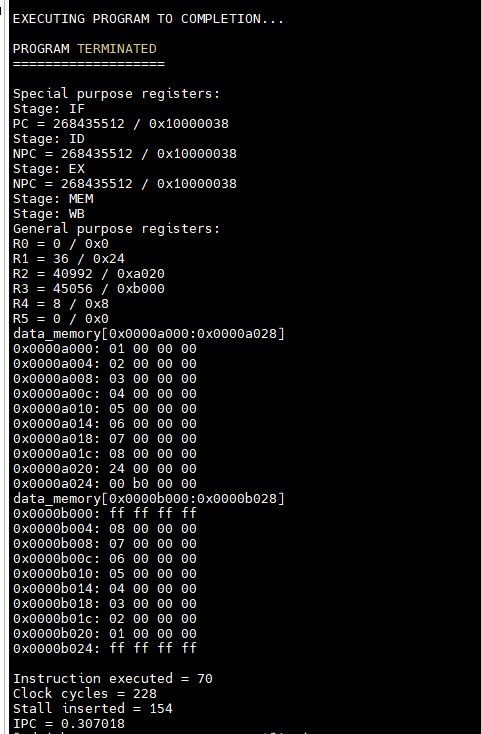


Figure 10. Testcase 6 Output

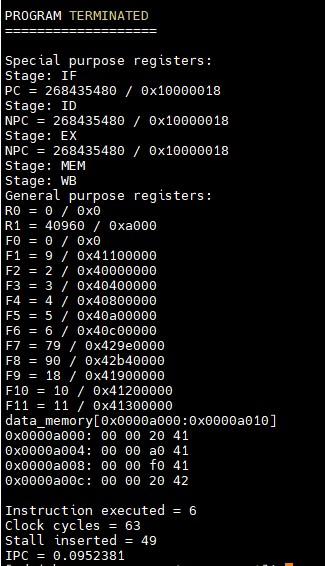


Figure 11: Testcase\_fp5 Output