Ray Chang
Tim Kim
ECE 154B

Project 4: Dual-Issue MIPS Pipeline

Introduction

The goal of this project is to design a 32-bit 5-stage Dual-Issue MIPS pipelined processor that is capable of executing simple programs and resolving any potential hazards. To achieve this, the necessary core modules of the processor were adapted and modified from the pipelined processor in Project 1 to allow for 2 instructions to be executed during the same cycle. After confirming the performance of the initial circuit, more modifications were made to the hazard and datapath modules to resolve any potential hazards during program execution. After significant debugging and testing, we were able to achieve most of the design requirements of the project. This implementation is coded in Verilog and simulated in ModelSim.

It is important to note that we were unable to implement a fully working dual-issue pipeline processor that incorporated a cache system and branch predictor. In our previous projects, we have implemented relatively successful designs of both systems; however, implementing a complete processor involving all functionalities served to be a challenging task. We do come up with possible solutions and implementations for the cache systems and the branch predictor that work in theory. Further details are discussed in the design methodology section.

Block-Level Top Schematic

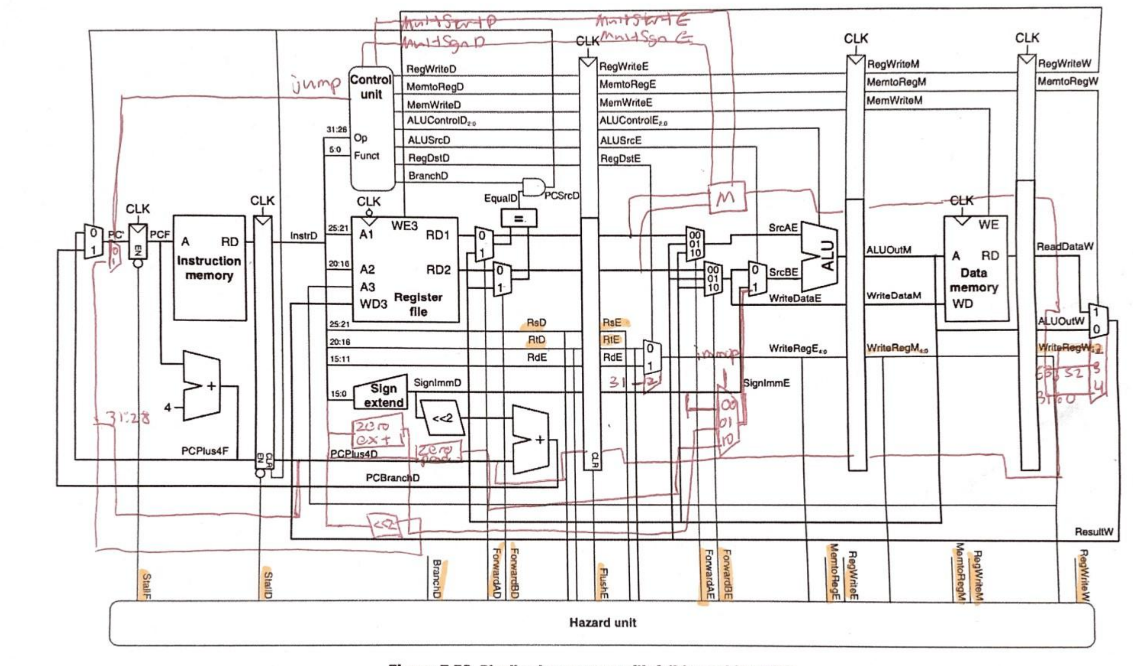


Figure 7.58 Pipelined processor with full hazard handling

Figure 1. Annotated Schematic of In-Order MIPS Pipeline

The schematic diagram above is the circuit of the pipeline processor that was implemented in project 1. Since a dual-issue pipeline begins two instructions at a time and executes them in the same clock cycle, we decided to design the processor by adding another layer of the original datapath. Below is an updated circuit diagram that is capable of executing two instructions simultaneously, allowing for a much more efficient pipeline processor.

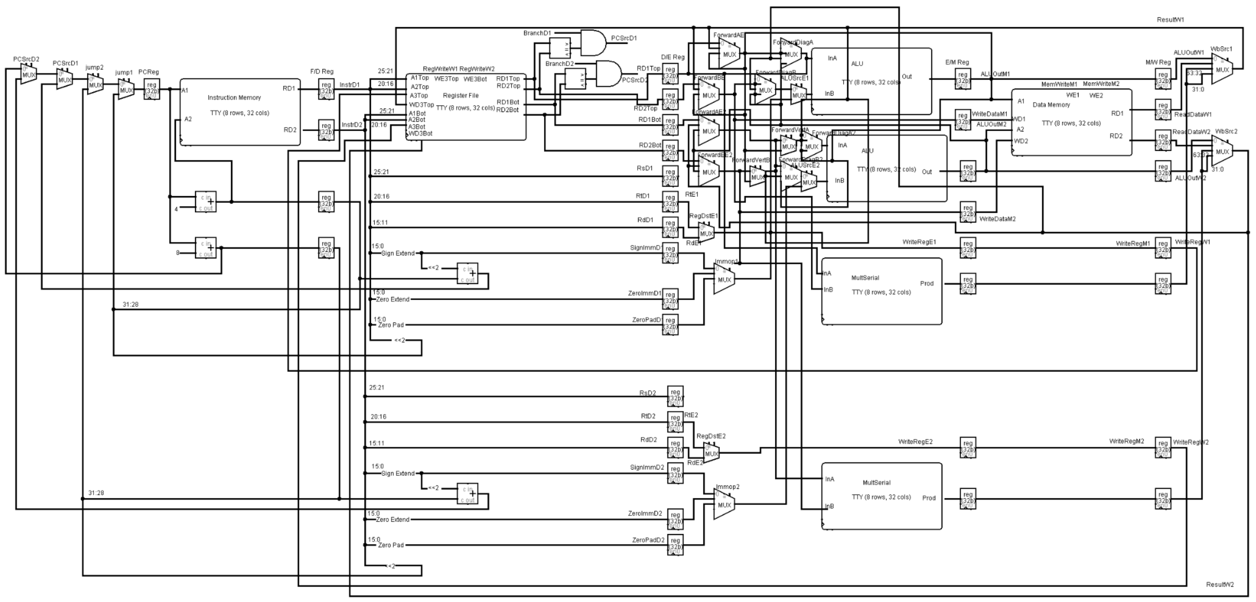


Figure 2. Schematic of In-Order Dual-Issue Pipeline

In this model of the pipeline, our target instruction set architecture (ISA) includes the following: add, addu, addi, addiu, sub, subu, and, or, xor, xnor, andi, ori, xori, slt, sltu, slti, sltiu, lw, sw, lui, j, bne, and beq. The instruction set is the same as the set that was used in project 1, but it excludes a few functions such as the multiplication and jump-and-link instructions.

Design Methodology

In order to ensure that the base model of the in-order pipeline functions correctly, the implementation that was designed for project 1 is used as a foundation. Then, the implementation is redesigned so that it supports dual-issue functionality. Some significant changes are made to the design, starting with doubling the number of input ports for certain modules of the schematic. In particular, the instruction memory, register file, data memory, control unit, and hazard unit modules are changed to allow for two sets of input and output data. Such a change would allow for each module to operate on two different instructions for each clock cycle. The datapath of the system is also significantly changed; the processor is modified so that it has another set of the original datapath that was implemented in the single-issue pipeline. This means that there is an additional multiplexer, sign-extend, zero-extend, zero-pad, adders, shifters, and AND gates for the processor.

After implementing a base model capable of executing a simple set of dual-issue instructions without taking into account any hazards or issues, a sufficient test program is run to examine the functionality of the pipeline processor. It is concluded that the simple dual-issue program ran successfully without any observable problems, meaning that the processor is able to handle two instructions simultaneously. Now, additional changes are made to the model to allow for it to handle hazards that would likely happen in more advanced programs.

First, an extensive list of hazards is established in order to understand what problems are being solved in the next implementation. A certain type of hazard is known as the read-after-write hazard, in which the second instruction of a pair of related instructions is dependent on the result of the first instruction. Listed below are some examples of read-after-write (RAW) hazards that are examined:

a. Read from register and then write register value into data memory address
    i. LW $s0 0x0000 $zero
       SW $s0 0x0004 $zero
b. Store then load same register
    i. SW $s0 0x0000 $zero
       LW $s1 0x0000 $zero
c. add/sub/or/xor involving same register
    i. ADD $s0 $t1 $zero
       SUB $s1 $t2 $s0

To tackle the three types of RAW hazards, a clever solution is implemented involving data forwarding and stalling strategies. It is determined that there are “horizontal”, “vertical”, and “diagonal” read-after-write hazards. A “horizontal” hazard involves data forwarding on the same parallel level of two instructions, a “vertical” hazard involves stalling to allow for a second dependent instruction to work given a related first instruction, and a “diagonal” instruction involves data forwarding on instructions that are on different parallel levels and different cycles.

After much consideration, it is established that a write-after-write hazard is likely not possible due to how the data memory module is implemented. A write-after-write hazard occurs when data is stored on the same memory address consecutively. The data memory module that was implemented in our design takes care of this issue as it does exactly what is intended to happen; the second instruction overwrites the data that was written by the first instruction in a particular data address. No matter what the case, a second instruction will always overwrite data written by the first in a write-after-write sequence of instructions; thus, our data memory module is sufficient to take care of this hazard.

A third type of hazard that may occur results from jump instructions. Our implementation of the dual-issue pipeline is expected to handle a pair of instructions at a time, meaning that the first instruction is an even counter and the second is odd. If a jump instruction occurs, then there is a possibility that the target instruction is an odd instruction, which would result in a pair of instructions that are odd then even. In order to solve such an issue, a NOP instruction is inserted if the jump instruction jumps to an odd instruction, allowing for the processor to stabilize and execute instructions in an even-to-odd fashion. Another issue may be a set of jump instructions; if the first instruction is a jump, regardless of what the second instruction is, the target address is taken and the second instruction is flushed. If the second instruction is a jump, and the first is not, then a normal jump instruction is executed, and any instructions that begin throughout the process are stalled.

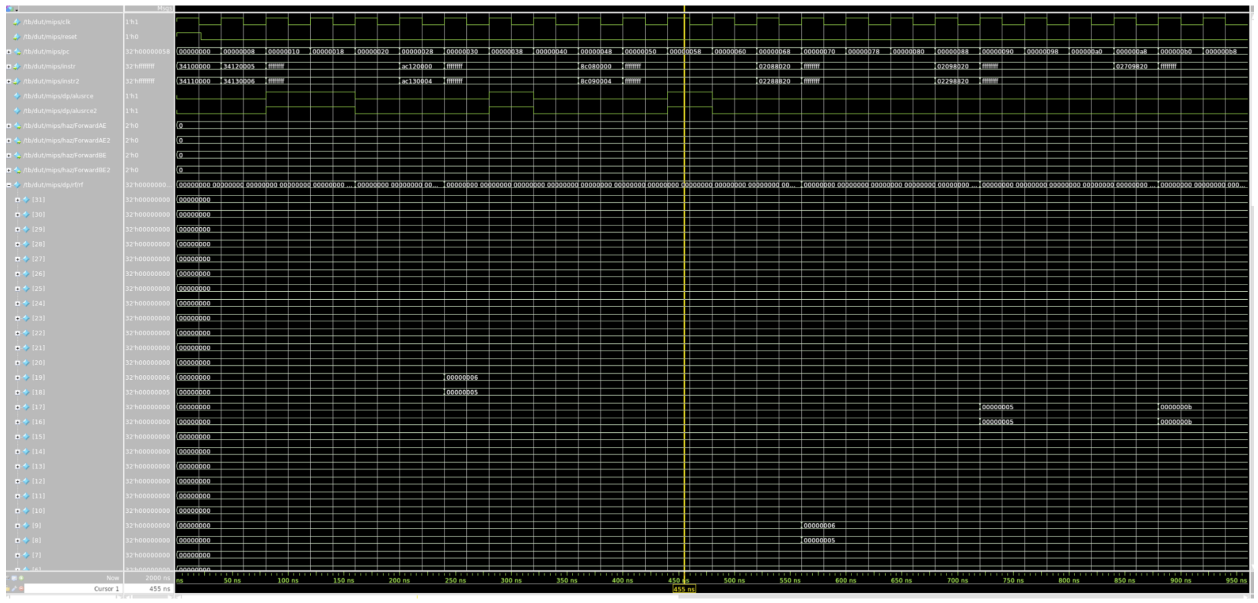
Finally, hazards caused by different cases of branch predictions are examined. If both instructions of a set are not taken, then both instructions are executed as intended without any critical issues. If the first instruction is predicted taken, a NOP instruction is inserted instead of executing the second instruction. Depending on the correct result of branching, the second instruction is either executed or the branch is taken to the target address. In another case where the first prediction is predicted not taken, but the second is predicted taken, a NOP is inserted to wait for whether the branches are actually taken or not; if the first is actually taken, then the second instruction is flushed, and if the prediction of the first is correct, then the second instruction is executed. In order to calculate the different possibilities of the branch predictions, we often relied on NOPs to wait a cycle until we knew the correct result of branches; then, we flushed instructions or allowed normal execution to happen depending on the outcome.

Modules Designed

1. Instruction Memory
   The instruction memory module is updated so that it takes two 32-bit program counters as inputs and two 32-bit instruction addresses as outputs. Instead of accessing the array of instructions once as before, it is accessed twice in the same clock cycle, allowing for the module to access an address at both PC and PC + 4. It is important to note that two PC values are inputted into the module instead of one. This design choice is made, because the corresponding PC values are calculated in the datapath instead of inside of the module.
2. Register File
   The register file module is updated so that there are two sets of inputs, which include the register source of A, register source of B, register source that is being written to, data for the destination register, and the write enable signal. Thus, four registers are accessed, and two registers are written to for each clock cycle. The module also outputs the contents of two sets of registers A and B.
3. Data Memory
   The data memory module takes as inputs two data addresses, two 32-bit data bits, and two write enable signals, each one corresponding to the top and bottom instructions. If the signals are enabled, then data is read from the top and bottom data addresses.

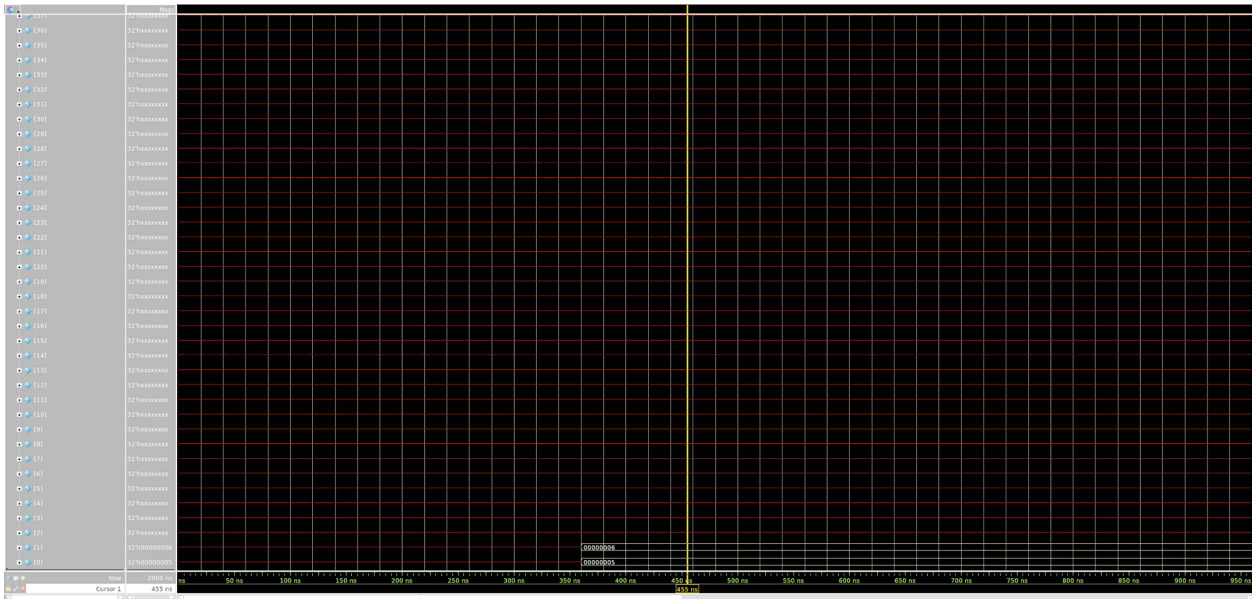
4. Control Unit
   The control unit is updated so that it performs two logical operations at once, allowing for two instructions to be decoded simultaneously. The unit takes two sets of opcode and function units, and it outputs corresponding control signals that are used throughout the datapath.
5. Hazard Unit
6. Register Units
   The register units are crucial for storing data in between the different stages that occur throughout an instruction's process. The register modules are updated so that it has double the amount of input and output ports.
7. MIPS/Top
   In the MIPS module and the tops module, wires were rearranged so that correct numbers of ports matched for the updated modules mentioned previously.

Test Programs



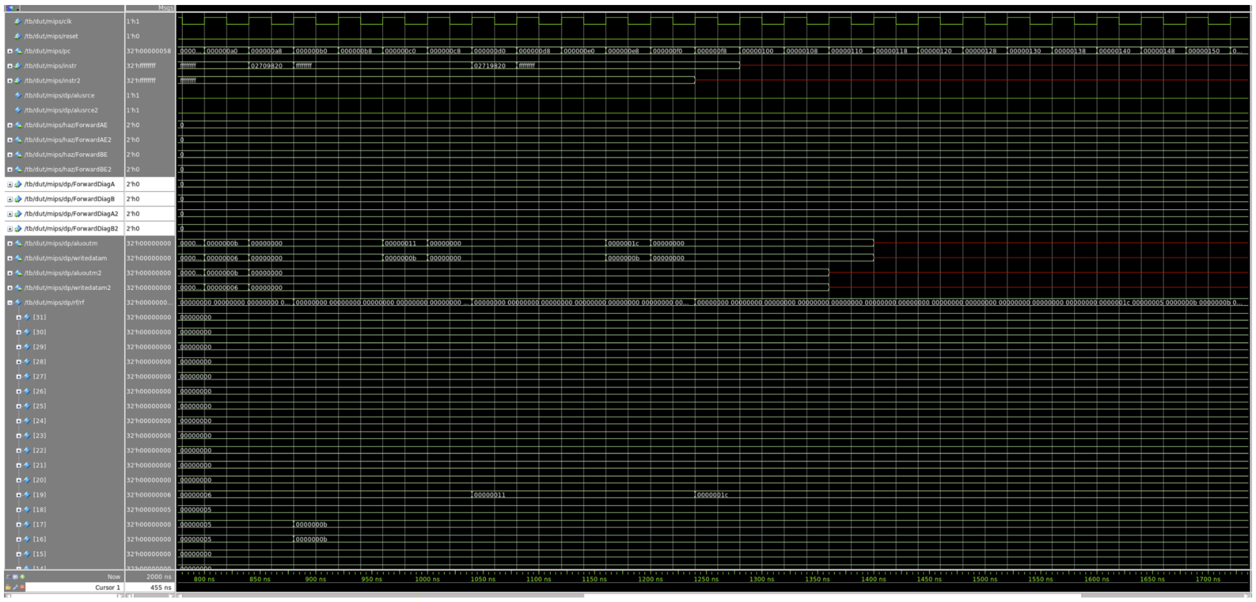
Waveform of memfile9.dat Part 1

The above waveform shows the correct outputs in the register file after each instruction. Register s2 is initialized to 5 and s3 is initialized to 6. A few instructions later, s0 and s1 are both set to 5. Next, they are increased by 6 due to the values in t0 and t1 so s0 and s1 are now 0xb. The values of t0 and t1 appear due to the load word instructions.



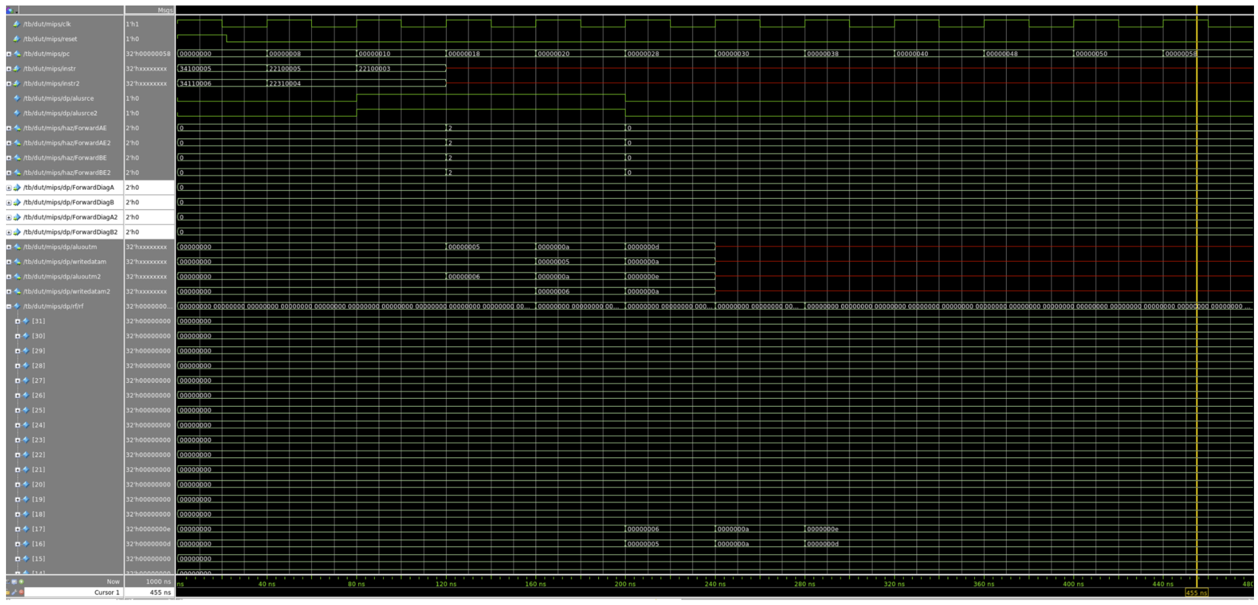
Waveform of memfile9.dat Part 2

This waveform shows the result of the store word instructions, as the first two addresses of the data memory store the values 5 and 6.



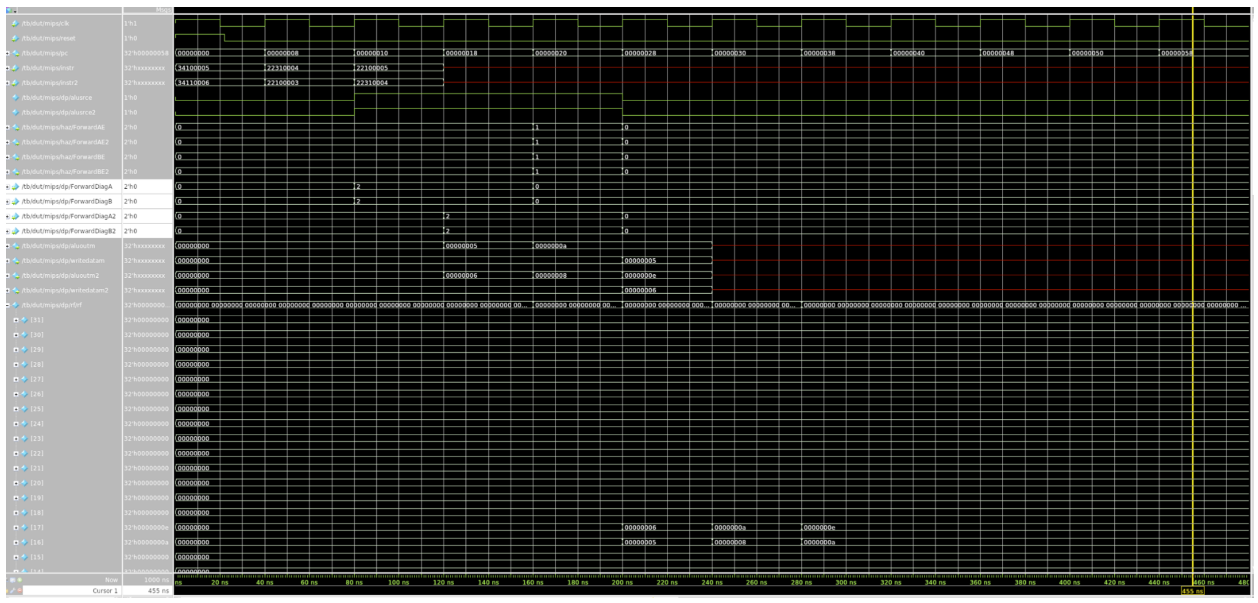
Waveform of memfile9.dat Part 3

This waveform shows s0 increasing to 0x11 and then 0x1c, which is the expected final result. This program contains nops between certain instructions, which is a limitation of this implementation, and this problem would be resolved by taking care of the hazards.



Waveform of memfile10.dat

This program tests the horizontal forwarding capabilities of this implementation. Instructions for reading and writing to s0 are always the even instructions, while instructions for reading and writing to s1 are always the odd instructions. Register s0 has values 0x5, then 0xa, then 0xd, while register s1 has values 0x6, 0xa, 0xe, which are the expected results. No nops were used in this program, which shows that the horizontal forwarding is successful.



Waveform of memfile11.dat

This program tests the diagonal forwarding of this implementation. Diagonal forwarding occurs when RAW instructions happen in consecutive cycles but are on different level pipelines. In other words, there's a bottom to top forwarding and a top to bottom forwarding. The fact that s1 goes from 0x6 to 0xa shows that the bottom to top forwarding is working, and the fact that s0 goes from 0x5 to 0x8 shows that the top to bottom forwarding is working. However, the third value of s0 is incorrect, so this program may introduce a new hazard that is not resolved with this diagonal forwarding.

Conclusion

In this project, we implement an in-order MIPS pipeline processor that is able to handle dual-issue instructions, meaning that our processor is able to execute two instructions simultaneously to provide a significant advantage in efficiency. We first create modules that are capable of handling two sets of inputs and outputs at the same time, and we also update our control and hazard units to give correct signals to the datapath of the instructions. Our original datapath was duplicated in order to handle another whole set of signals. After a raw implementation of our dual-issue processor, we attempt to debug any hazards and issues that came along with the design process.