Trevor Morris and Tristan Seroff
5/19/16

ECE 154B Project 4 Report

Introduction

Here we describe how we modified our MIPS architecture to be dual-issue superscalar.

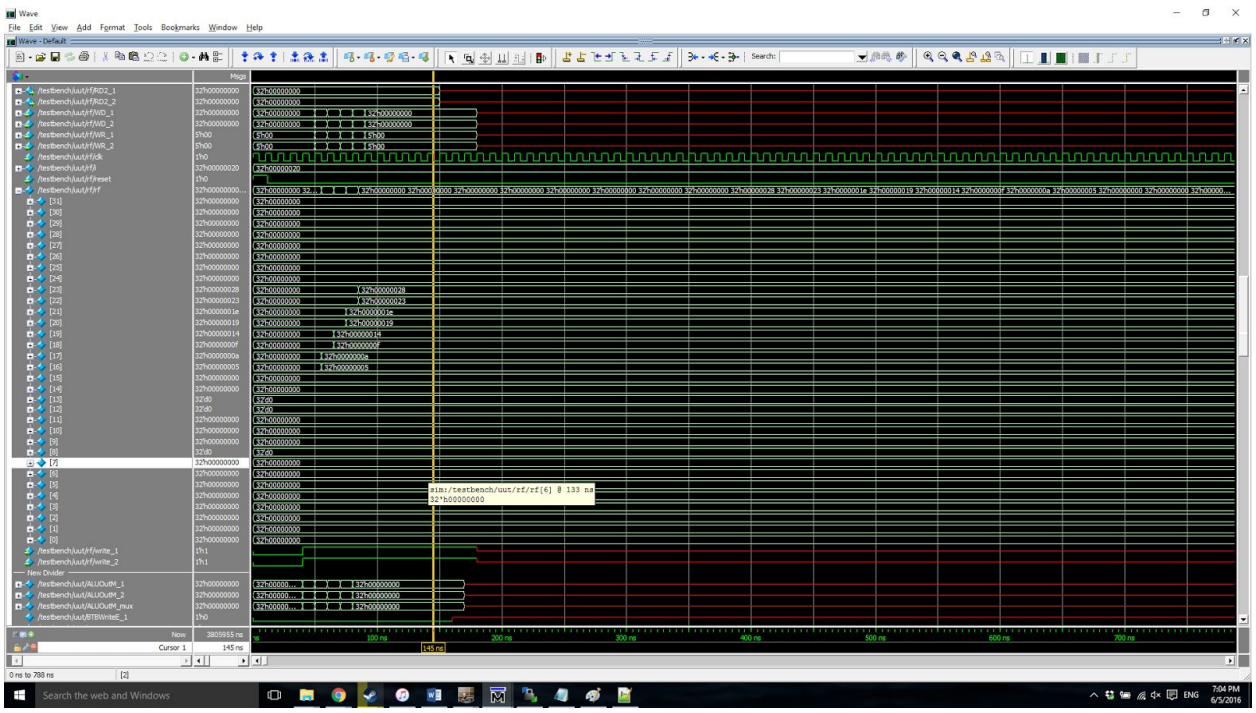
Design Methodology

Step 1) No Hazards

We refactored our code from the previous lab so all of the global structures (Register file, Data & Instruction memory, and BTB) were in our top module (‘mips.v’) and the rest of the pipeline is in a module ‘datapath’. To create a second pipeline, we simply instantiated datapath twice. We modified the fetch stage so that pipeline 1 gets the instruction at PC and pipeline 2 gets the instruction at PC+4. On every clock cycle the global (‘Master’) PC is incremented by 8.
We tested a simple program that contains no data hazards. The waveform showing the register file as the program runs is below.

Test program:

```
addi $s0, $zero, 5
addi $s1, $zero, 10
addi $s2, $zero, 15
addi $s3, $zero, 20
addi $s4, $zero, 25
addi $s5, $zero, 30
addi $s6, $zero, 35
addi $s7, $zero, 40
```



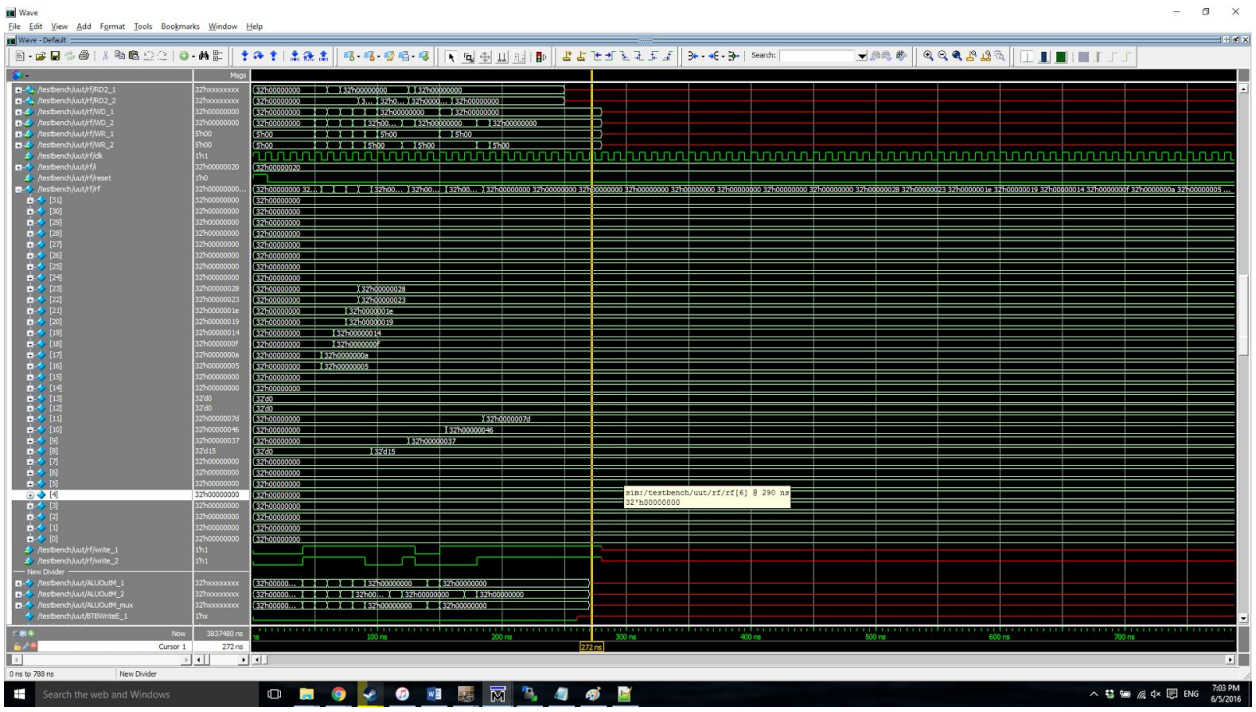
Step 2) With Data Hazards

We then added support for data hazards. In the decode stage we added logic to the hazard unit to determine when to stall in order to prevent out-of-order execution.

If the first pipeline needs something from the second pipeline in the Execute or Memory stage, the first pipeline is stalled in the decode stage while the second pipeline goes until the dependency is resolved. Or, if the first pipeline needs something from its own Execute or Memory stage, both pipelines are stalled at decode until the data dependency is resolved. During this time, both pipelines are not allowed to fetch new instructions into the decode stage. We added some dependent instruction to our test program. The waveform showing the register file as the program runs is below.

Test program:

```
addi $s0, $zero, 5
addi $s1, $zero, 10
addi $s2, $zero, 15
addi $s3, $zero, 20
addi $s4, $zero, 25
addi $s5, $zero, 30
addi $s6, $zero, 35
addi $s7, $zero, 40

add $t0, $s0, $s1   #t0 = 15
add $t1, $t0, $s7   #t1 = 55
add $t2, $t0, $t1   #t2 = 70
add $t3, $t2, $t1   #t3 = 125
```



Step 3) With Memory Access

If memory is being accessed, both pipelines are stalled until the memory access is complete. Because data memory only has one port, it cannot be accessed by both pipelines simultaneously. We implemented the following 3-state FSM in the hazard unit to allow both pipelines to take turns accessing memory in this case.

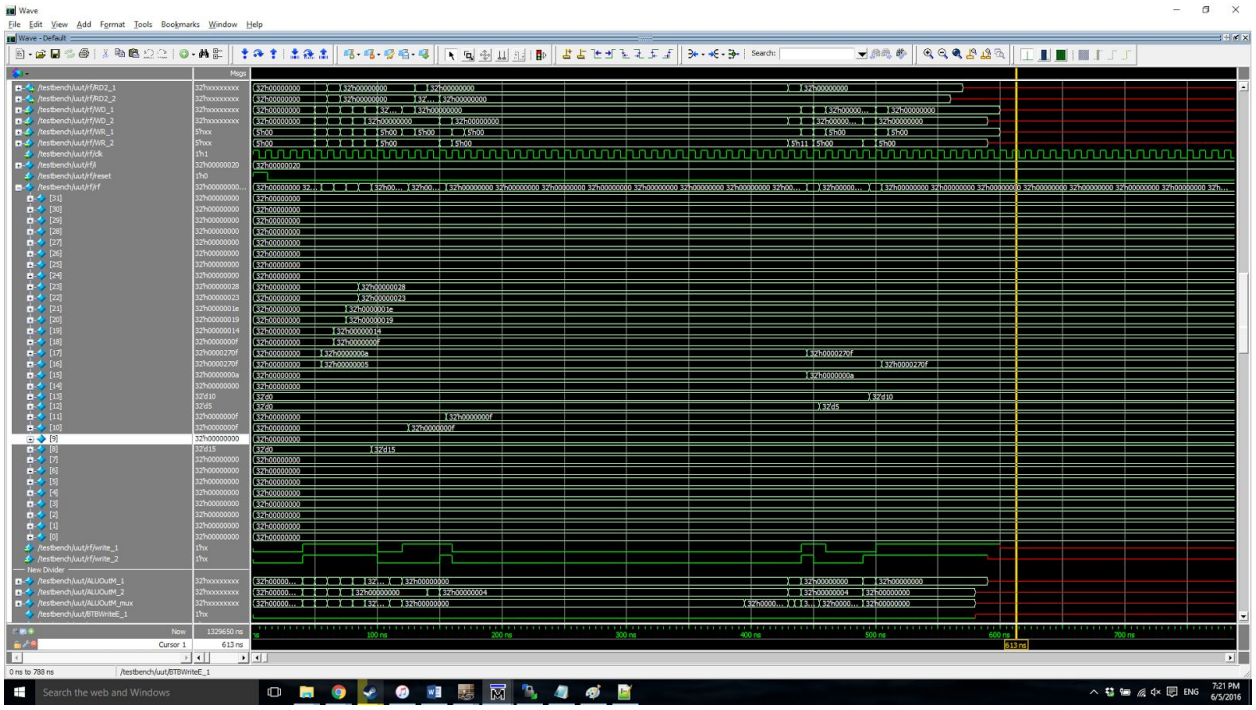
State	Output	Transition
State 0(Default)	Stop stalling E1, E2, M1, M2 stages, Stop flushing M1 and W2.	If both pipelines are trying to access memory in the same cycle, go to State 1
State 1(Pipeline 1's turn)	Set MemoryUser to pipeline 1(This controls a MUX which chooses which pipeline's addr, write, and write_data go to the data memory) Start stalling E1, E2, M1, M2, Start flushing M1 and W2 (stops both memory operations from being duplicated as the pipeline is stalled)	If the memory is done being accessed, go to State 2
State 2(Pipeline 2's turn)	Set MemoryUser to pipeline 2	If the memory is done being accessed, go to State 0

We added a few memory accesses to our test program.

```
addi $s0, $zero, 5
addi $s1, $zero, 10
addi $s2, $zero, 15
addi $s3, $zero, 20
addi $s4, $zero, 25
addi $s5, $zero, 30
addi $s6, $zero, 35
addi $s7, $zero, 40

add $t0, $s0, $s1  #t0 = 15
add $t1, $t0, $s7  #t1 = 55
add $t2, $t0, $t1  #t2 = 70
add $t3, $t2, $t1  #t3 = 125

sw $s0, 0($zero)
sw $s1, 4($zero)
```

```
lw $t7, 4($zero)
addi $s1, $zero, 9999

lw $t4, 0($zero)
lw $t5, 4($zero)

addi $s0, $zero, 9999
```

In the waveform below you can see the register file as the program executes. The large break in the middle is the stall as the memory is accessed, but subsequent accesses then use the cache.



Step 4) Branch support

We created the following chart to determine what should happen for every case of having branches in both instructions.

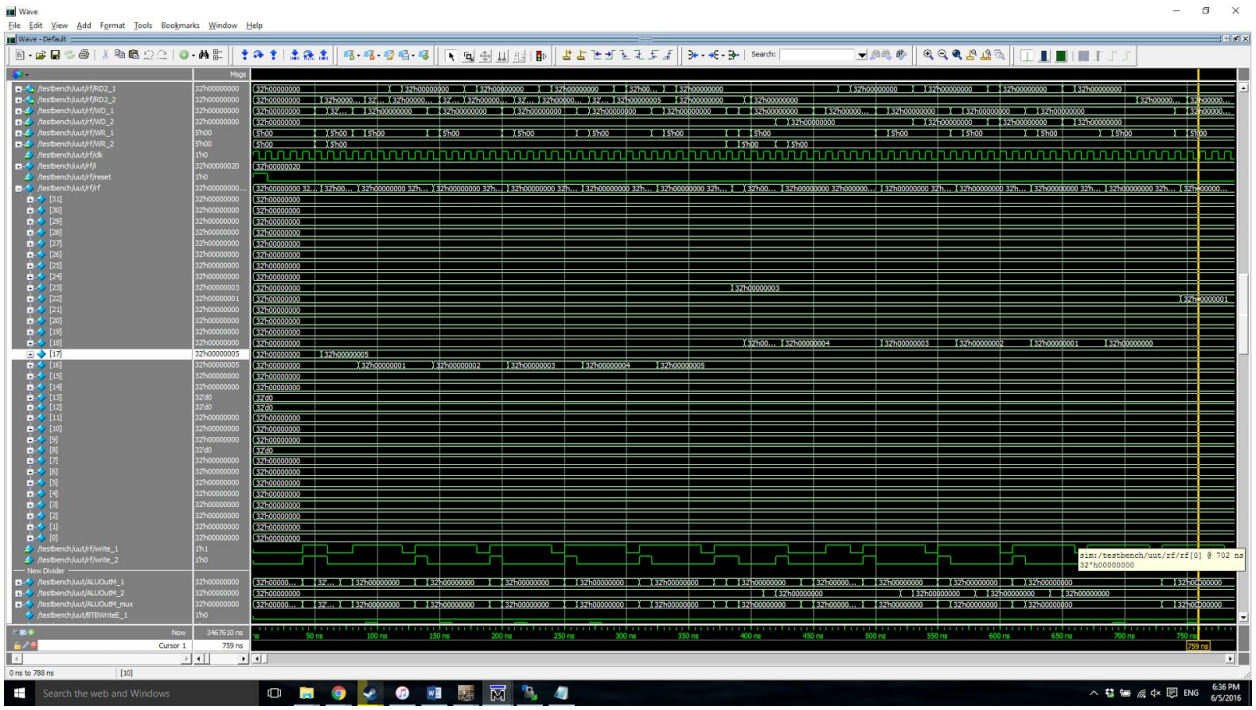
Inst 1	Inst 2	PCNext_1 (_2 is +4)	1st Branch Misses?	2nd Branch misses?	Control Ops
Branch predict taken	Anything	PCPredicted_1	Flush D,E, PC = PCPlus8E	N/A	Flush instruction 2 to a nop
Branch predict not taken	Not branch	PCPlus8	Flush D,E, PCF1 = PCBranch1,	N/A	Continue normally unless there

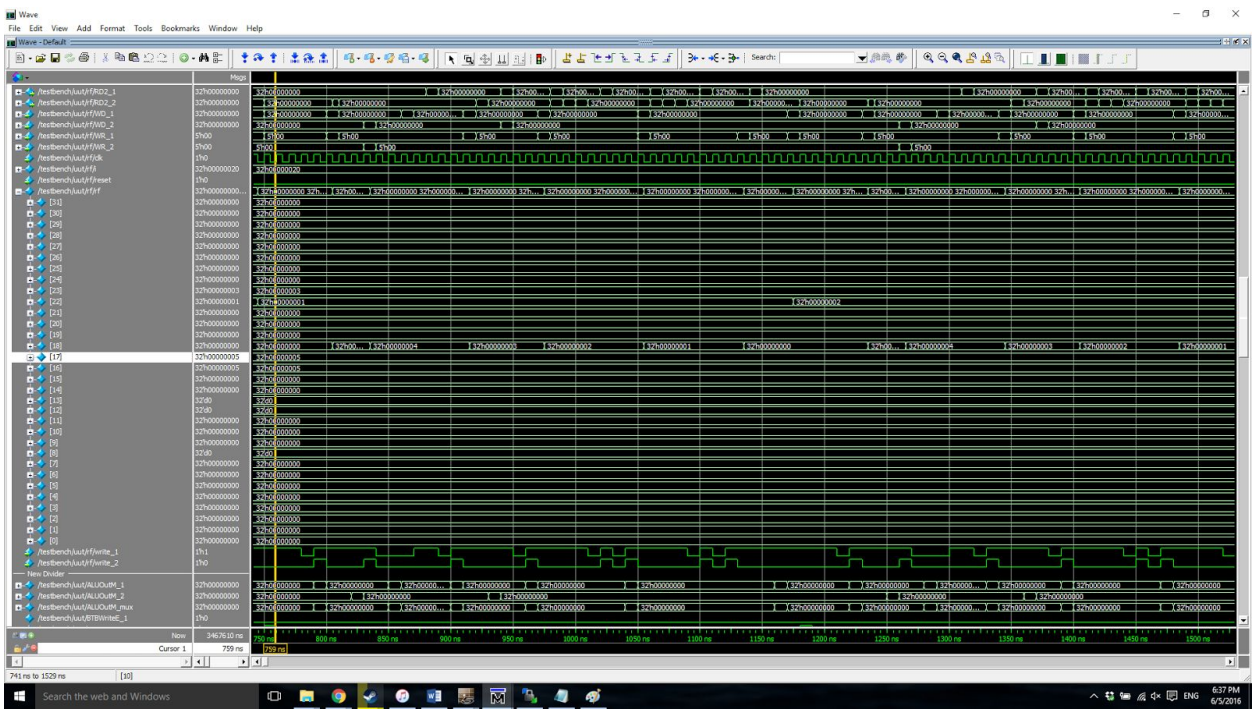
			PCF2 = PCBranch1+4		is a misprediction
	Branch predict taken	PCPredicted_2	Same as above case ^	Flush D,E, PCF1 = PCPlus4E_2 PCF2 = PCPlusPlus4E_2+4	Stall until 1st branch resolves (StallBothAtFetch, StallTwoAtDecode until instruction 1 finishes)
	Branch predict not taken	PCPlus8	Same as above case ^	Flush D,E, PCF1 = PCBranchE_2 PCF2 = PCBranchE_2+4	Stall until 1st branch resolves (StallBothAtFetch, StallTwoAtDecode until instruction 1 finishes)
Not branch	Not branch	PCPlus8	Flush D,E, PCF1 = PCBranchE_1, PCF2 = PCBranchE_1+4	N/A	Continue normally
	Branch predict taken	PCPredicted_2	N/A	Flush D,E, PCF1 = PCPlus4E_2 PCF2 = PCPlusPlus4E_2+4	StallBothAtFetch, StallTwoAtDecode until instruction 1 finishes
	Branch predict not taken	PCPlus8	N/A	Flush D,E, PCF1 = PCBranchE_2 PCF2 = PCBranchE_2+4	StallBothAtFetch, StallTwoAtDecode until instruction 1 finishes

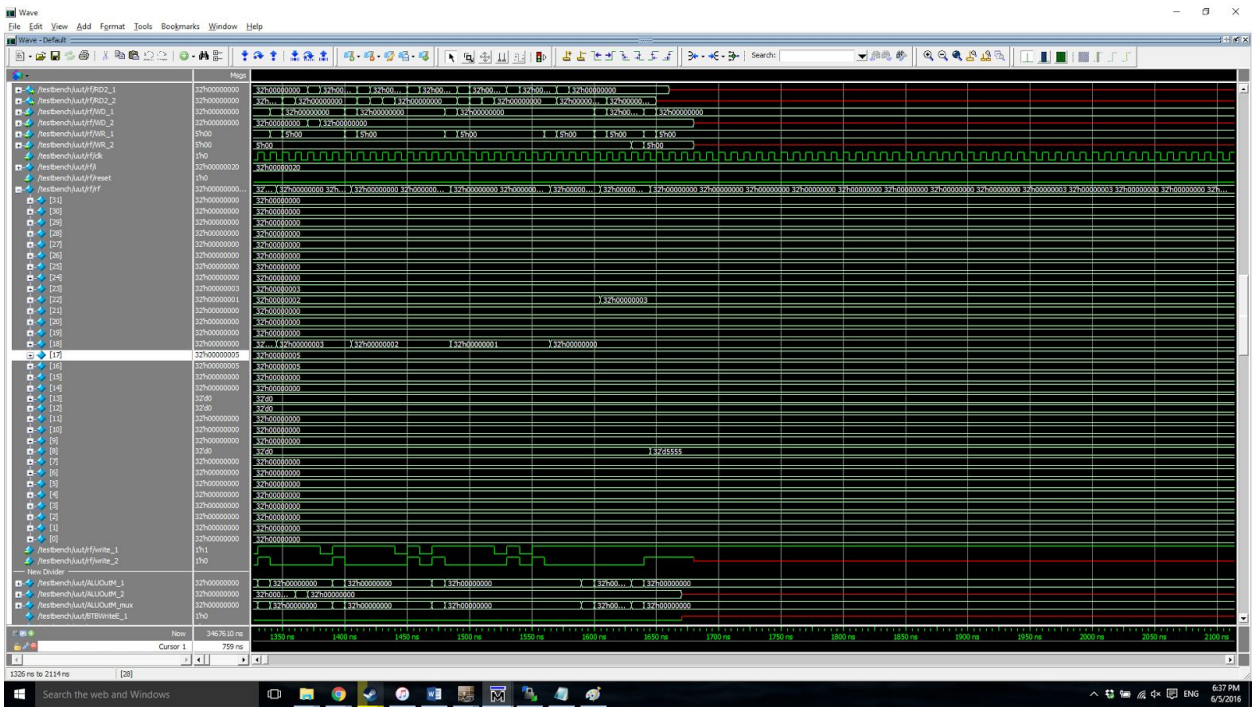
We used our nested loop branch program from Project 3 to test this step.

```
#TEST 1
#FOR LOOP

addi $s1, $zero, 5
addi $s0, $zero, 0
loop1:
addi $s0, $s0, 1
bne $s0, $s1, loop1 #BRANCH1 = 0x0000c
# should add to buffer, predict taken x 3, mispredict taken


#TEST 2
#NESTED FOR LOOP

addi $s7, $zero, 3
addi $s6, $zero, 0
loop3:


addi $s2, $zero, 5
loop2:
addi $s2, $s2, -1
beq $s2, $zero, done #BRANCH2 = 0x000020
j loop2
done:

addi $s6, $s6, 1
bne $s6, $s7, loop3 #BRANCH3 = 0x00002c


addi $t0, $zero, 5555

#first run of loop3:

#branch 2 is not taken and not added to buffer x4
#on last iteration is taken and added as weakly taken

#branch 3 is added as weakly taken


#second run of loop3

# branch 2 is mispredicted taken, then updated to predict weakly
not taken
# correctly predict not taken x 3, now strongly not taken
```

```
# mispredict not taken, now weakly not taken

#branch 3 is predicted taken, now strongly taken


#third run of loop3

#branch 3 is predicted not taken, now strongly not taken
#branch 3 predict not taken x3
#mispredict not taken, now weakly not taken

#branch 3 is mispredicted taken
```





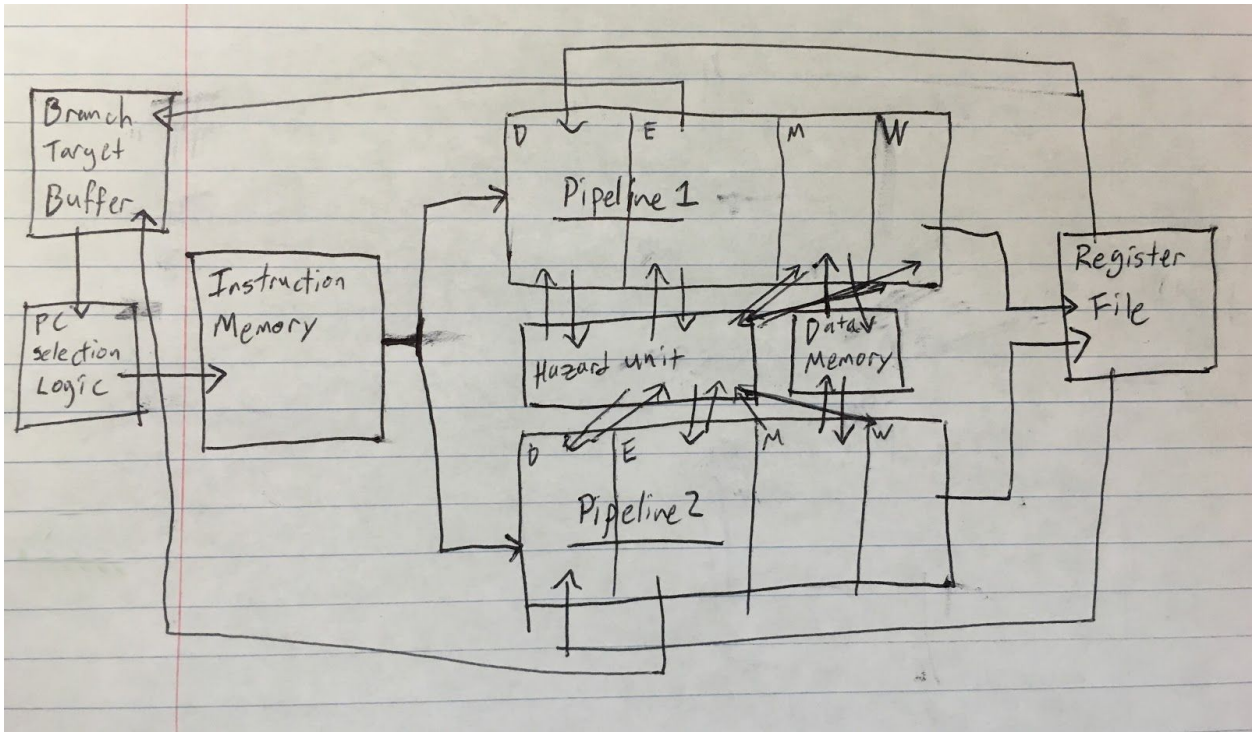


Step 5) Jump

We re-added support for jumping operations to the processor by adding a multiplexer in the PC control logic to choose between the PC from regular operation or a branch instruction, and the PC as calculated from the immediate in a jump instruction. Another multiplexer chooses between the calculated target PC as output by pipeline 1 and pipeline 2, and is controlled by which pipeline the jump instruction is actually located in.

When a jump instruction occurs, the unified fetch stage is flushed and advanced to the new program counter. Additionally, if pipeline 1 has a jump instruction, the instruction in the decode stage of pipeline 2 is flushed, as it should not be executed due to the jump.

Overview Diagram of Design



- There are two main pipelines, each with decode through writeback stages, and control units (not shown)
- There is a single unified fetch stage, consisting of the branch target buffer, PC selection logic (multiplexers, a clocked buffer, etc), and the instruction memory.
- There is a single data memory, consisting of a cache and main memory, with a multiplexer connecting its single port to either pipeline, but not both simultaneously.
- There is a single dual-ported register file which both pipelines can use simultaneously.
- There is a single large hazard unit which coordinates stalling and flushing of instruction stages due to data hazards, memory latency, or branching/jumping actions.

Conclusion

The lab took around 15 hours to complete. The largest amounts of time were spent debugging controlling memory stall actions (stalling and flushing appropriate pipeline stages in exactly the correct cycle timing), and re-integrating the branch target buffer with the superscalar setup and performing branch misprediction corrections appropriately while maintaining correct program execution.