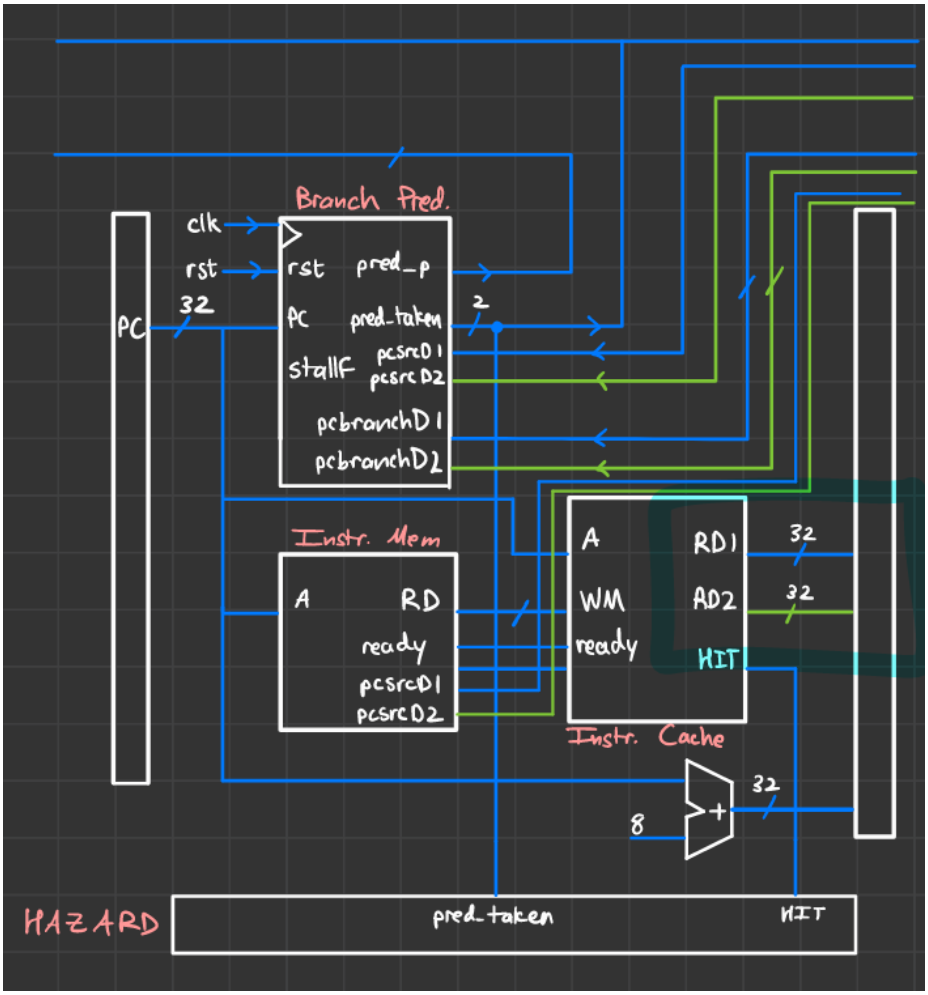
Project 04: Dual-Issue Superscalar
Brian Lee, Arjun Vinod
ECE154B W22

Overview

The dual issue processor allows us to achieve a CPI of less than one by sending two instructions down the pipeline simultaneously. However, we found that this architecture achieves this feat at the cost of extreme complexity that makes the designer compromise between limiting the scope of programs that this processor can execute and handling every hazard that comes its way. We made several design choices along the way to modify our original single-issue pipelined processor to support dual-issue, and this report outlines the modifications at each stage.

Design Specification

Fetch



The fetch stage consists of three core modules: instruction cache, instruction memory, and a local branch predictor. These modules were relatively easy to modify to support dual-issue because instructions will typically be read in sequential order.

Instruction Memory

This memory already has a wide bus that brings multiple instructions at a time to the cache on a cache miss. The only modification we had to make was to check if the decoded instruction in the decode stage was a branch taken. If so, we flush the two instructions just fetched.

Instruction Cache

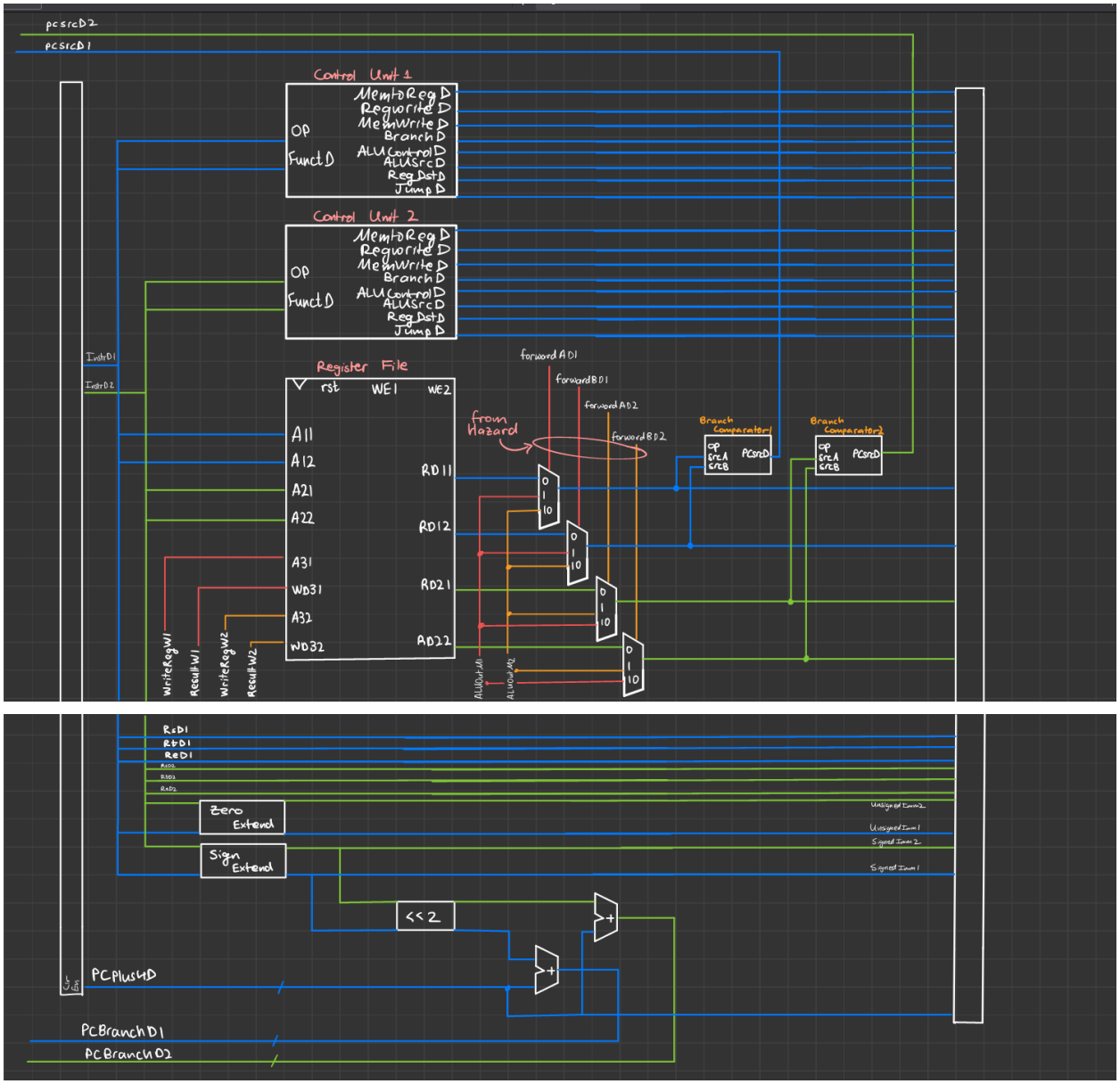
Again, since instructions are read sequentially, we will output the instruction corresponding to the inputted program counter and the program counter + 4. The PCPlus4F signal sent to the decode stage is now program counter + 8 since two instructions are processed at a time.

Branch Predictor

This is where we introduce the dependency flag. This flag's value is set in the decode stage, but it helps us determine whether or not the second instruction has a read-after-write hazard with the first instruction. If this flag is true, then we do not update the state of the branch predictor for the instruction corresponding to program counter + 4. The reasoning behind this will be clear once we go over the behavior of the decode stage if there's a dependency. If there's no dependency, then the state of the finite state machine of both instructions is updated for the branch predictor and the branch target address is updated if the decode stage indicates it's a branch taken.

If the first instruction is a branch predict taken, then we need to ignore the second instruction that was fetched and complete the branch. If the second instruction is a branch predict taken, then we should allow the first instruction to propagate through the pipeline and take the branch after it. We achieve this behavior using a two-bit predict_takenF flag that is 10 when the second instruction is predict taken and 01 when the first instruction is predict taken. In the former case, the first instruction should pass through the pipeline so we do not need to change anything for this to happen. In the latter case, the hazard unit stalls the memory and execute stage so that the instruction that needs to be skipped over does not make it past the decode stage.

Decode



Handling RAW Hazards

The decode stage is where we handle the processor's issuing of instructions when there's a read-after-write conflict due to the second instruction reading from a register that the first one writes to. To solve this, we break the two instructions into one instruction + one NOP per cycle. The forwarding circuitry handles the rest of the hazards presented by issuing these instructions consecutively. The first instruction is sent with a NOP down the pipeline, followed by a NOP with the second instruction. This process takes three cycles: one cycle to detect the hazard, and a cycle for each instruction to be issued. This alone presents a large cost to the CPI of the processor if a program has consecutive instructions that depend on the last instruction.

This hazard is detected in the execute stage by checking the Rs and Rt registers of the second instruction against the WriteReg register of the first instruction. We also need to make sure that this register is actually being written to (RegWrite is high). If there's a match, then we need to send the instructions independently down the pipeline. We keep track of this by defining a dependency flag. The hazard unit catches this case and all stages are stalled except for the decode stage that issues the first instruction alone in the first cycle. In the second cycle, only the fetch stage is stalled to allow the pipeline to process the first instruction before the second one is issued on its own. In the final cycle, the pipeline processes the second instruction in the execute stage and the hazard has been handled.

We also changed the forwarding logic so that the control signals are now three bits to cover the two cases. We may need to forward data from the ALU output from either ALU. The hazard unit generates these signals by comparing every combination of matching read-registers in the decode stage and write-registers in the memory stage.

Handling WAW Hazards

To consider write-after-write hazards, we add one more condition that sets off the dependency flag. We check if the WriteReg registers of both instructions match and that they're being written to (RegWrite is high). If this is the case, we want to send the instructions down the pipeline independently as well.

We thought about omitting the first instruction that writes to the same register since it ends up getting overwritten anyways. If the first instruction requires many cycles, we may end up saving a significant amount of time by doing so. However, we thought it warranted a deeper analysis into the ramifications of omitting an instruction on this basis so we didn't implement this idea.

Handling Branches with a RAW Hazard

If a branch is in the second instruction and depends on a register that's being written to by the first instruction, the dependence logic takes priority. So the branch instruction and instruction that it depends on are sent down the pipeline separately. Since the instruction the branch depends on is only one cycle ahead, the hazard unit inserts a bubble so that the instruction is two cycles ahead. When the instruction is two cycles ahead, it is in the memory stage while the branch is the decode stage. Our forwarding logic then correctly forwards the ALU output that the branch in the decode stage needs to determine if the branch is taken or not.

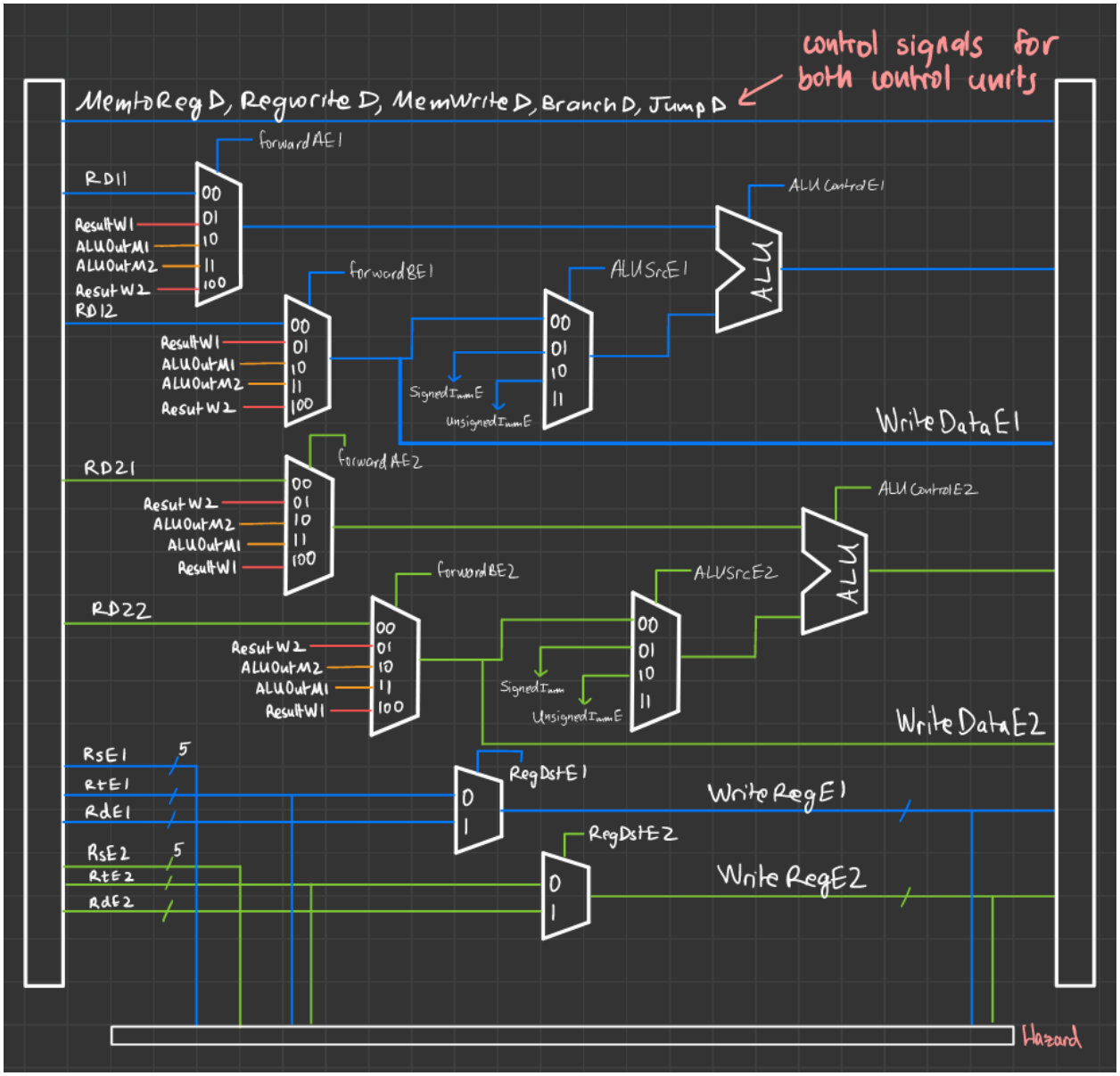
Register File

The register file only required one relatively simple modification: it needed to be able to take in double the inputs from the original design and output double the signals as well.

Control Units

Since the control units operate independently, we simply instantiated two control units to handle each instruction.

Execute



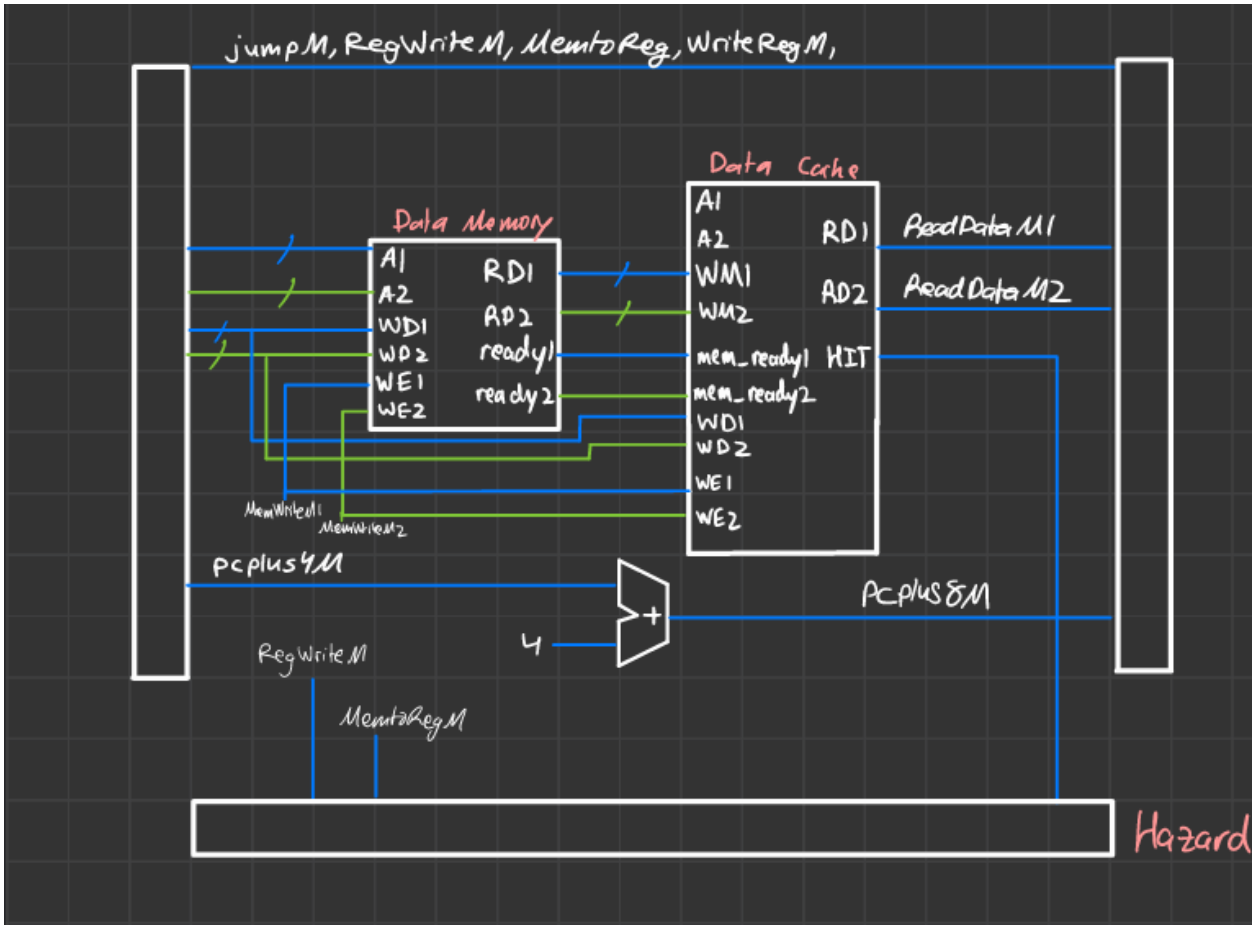
In our pipeline, two instructions flow simultaneously. This adds one more case that we need to consider when forwarding data. We already considered the cases where one instruction depends on another instruction in the same cycle. We also considered how to handle forwarding data from either ALU to the read-registers in the decode stage. Lastly, we must consider data forwarding in the execution stage.

Forwarding

We now need to add the ALU output in the memory stage or the resultW signal in the writeback stage for both instructions as inputs to the MUXs that lead to the ALUs of both instructions in the execute stage. To do so, we increase the number of bits of the forwarding control signal to three bits to consider each of these four forwarding cases. Similar to the single-issue processor's

forwarding logic here, we are checking the ALU input's registers against registers written to down the pipeline by both instructions in each stage down the pipeline.

Memory



Like the fetch stage's memory and cache, this stage required minimal modifications to support dual-issue. However, since data from the cache/memory doesn't need to be accessed sequentially like instructions, we need to implement a dual-address input to the data memory. Since we handled WAW hazards already, we don't need to consider the case where two places in memory are being written to at the same time.

Data Memory

Data memory now accepts two input memory locations and outputs the data for both after 20 cycles.

Data Cache

For the most part, we duplicated the logic in the single-issue data cache design to create the dual-issue cache design. Now, the cache accepts two memory locations as input and can output the data for both if they're cache hits. While the cache signals a miss, the hazard unit stalls all stages until the data is ready from data memory and written to cache.

Writeback

Again, this is one of the simpler stages that only required us to duplicate code for the single-issue design to produce the dual-issue design. However, we did need to add logic that correctly chooses between addresses that depend on the instruction. For example, if one instruction is a branch, we need to use the branch target address of this instruction if it's a branch taken and not PC+8. We implemented this logic by defining two wires, PC1 and PC2. We used the predict taken, branch taken, and jump flags from the fetch and decode stage to determine the selection of the program counter if the processor was a single-issue design. Then, we use a multiplexer to determine the program counter by choosing between PC1 and PC2. This choice is made depending on which instruction has a misprediction or jump instruction. If the first instruction meets any of these requirements, PC1 is chosen. Otherwise, PC2 is chosen. This works as intended because besides those three cases, the program counter is either PC+8 in the decode stage, PC+8 in the fetch stage, or the predicted program counter from the branch predictor. These signals are all the same for two instructions in the same cycle, so it doesn't matter which PC is chosen if no instruction-specific address is used.

Testing

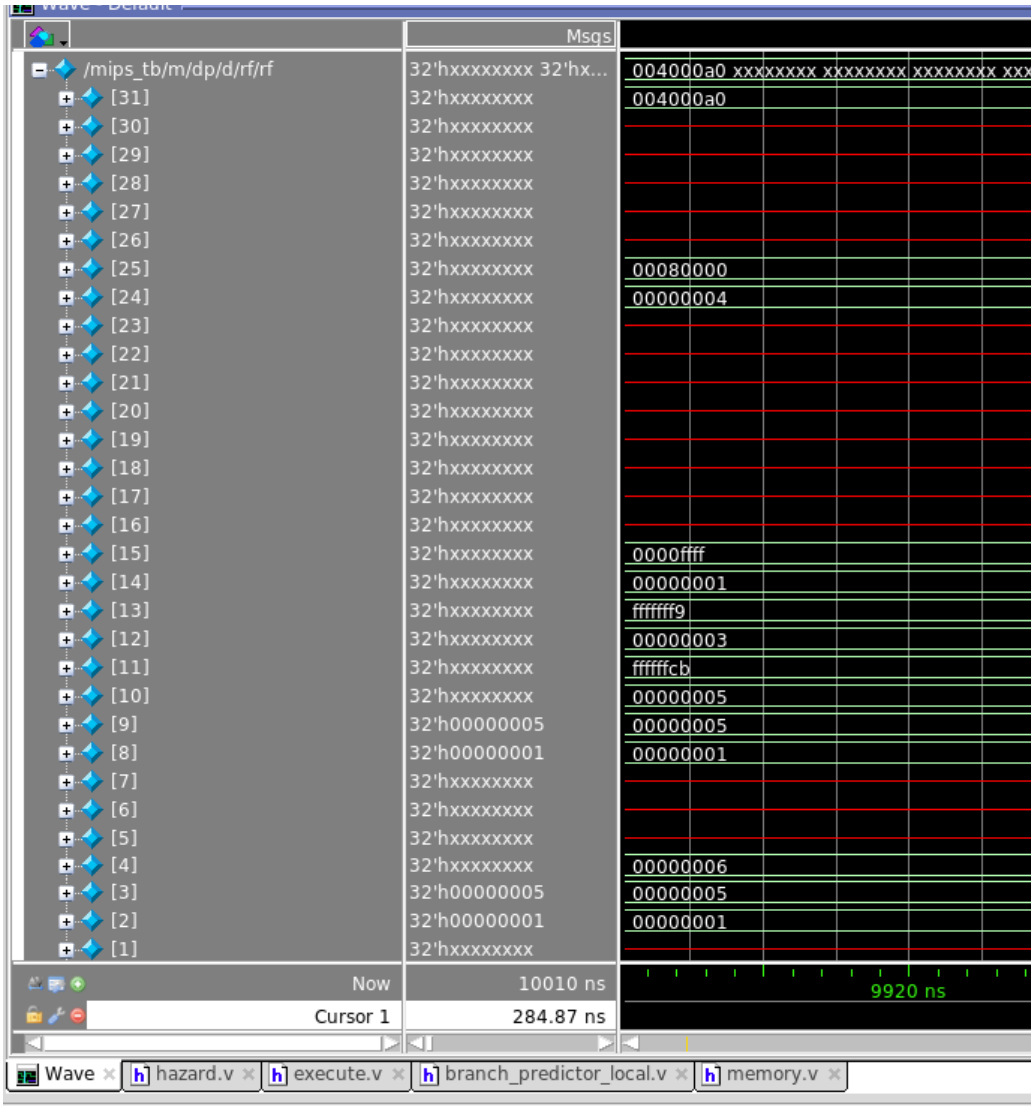
Stress Testing

The first program we run is the same as the program we ran in the first project that stress-tested our pipeline (minus the multiplication). This program will stress the processor because there are almost no independent instructions, many hazards, and multi-cycle instructions like loads and branches. Though we are not leveraging the strengths of the dual-issue processor with this kind of program, we can test how robust it is since it tests almost every instruction and hazard case we need to account for.

stress_test.s:

```
addi $v0,$0,1  #v0 = 1
ori $v1,$0,5   #v1 = 5
and $t0,$v0,$v1 #t0 = 1
or $t1,$v0,$v1 #t1 = 5
add $a0,$t1,$t0 #a0 = 6
sw $t1, 48($a0)
lw $t2, 48($a0) #t2 = 5
sub $t3, $t0, $t2 #t3 = -4
beq $t2,$t1,2 #branch
addi $t0,$0,0xAAAA  #Should not perform
addi $t0,$0,0xFFFF #Should not perform
addi $t4,$0,3  #t4 = 3
xor $t5,$t4,$t2 #t5 = 6
beq $t5,$t4,0 #Should not perform
```

```
xori $t5,$t2,1 #t5 = 4
bne $t5,$t4,0 #branch
addi $t0,$0,0xAAAA #Should not perform
slt $t6,$t5,$t4 #t6 = 0
slti $t6,$t5,8 #t6 = 1
lui $t7,1 #t7 = 10000 HEX
sub $t7,$t7,$t0 #t7 = FFFF HEX
andi $t8,$t7,4 #t8 = 4
lui $t9, 8 #t9 = 80000 HEX
sw $t1, 52($a0)
xnor $t5,$t4,$t2 #t5 = fffffff9
JAL 0x100025
```



The final state of the register file is shown above. After running the stress test and following the register values throughout the execution, we can safely conclude that our processor design is robust and it adequately handles every hazard it may face.

Dual-Issue Friendly Testing

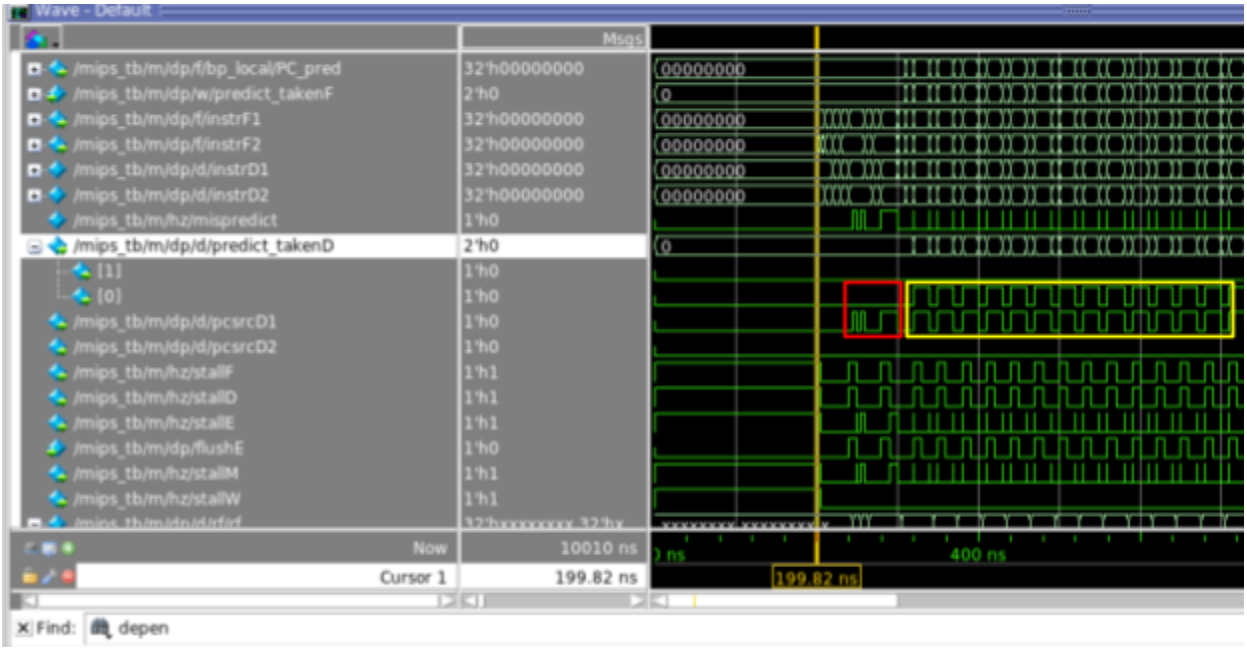
The second program we run on this processor is to see the dual-issue processor shine with the branch predictor. We run a simple for loop that increments two variables.

dual_issue.c:

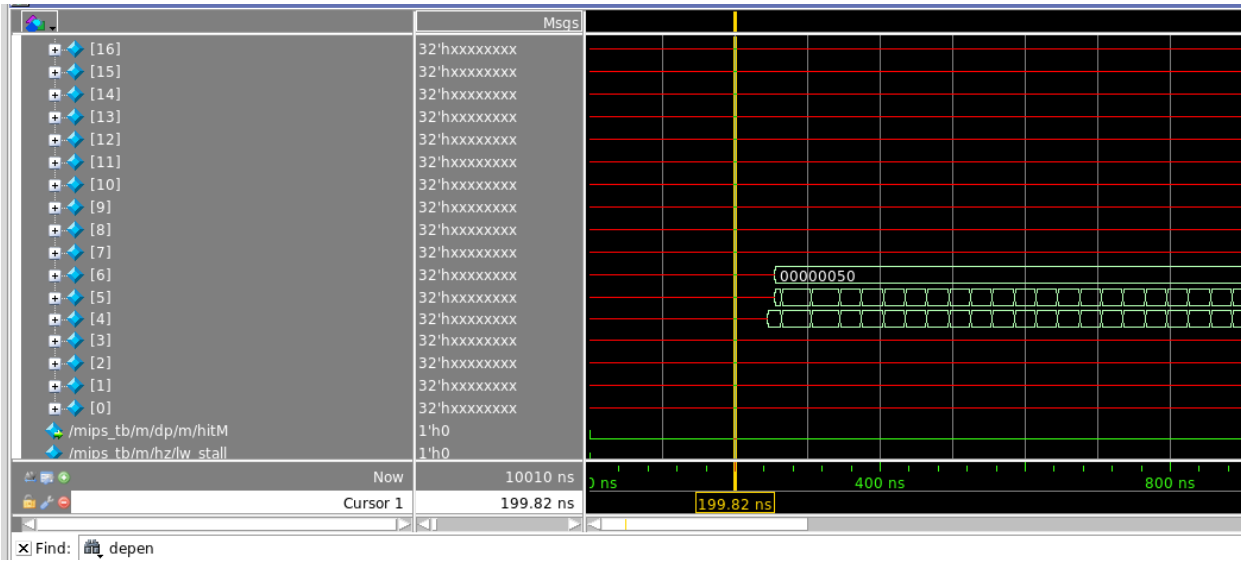
```
do {
      a0 += 1
      a1 += 2
} while (a0 != 0x50)
```

dual_issue.s:

```
addi $a0 $0   0x00
addi $a1 $0   0x00
addi $a2 $0   0x50
NOP # to optimize performance
addi $a0 $a0  0x01
addi $a1 $a1  0x02
bne  $a0 $a2 0xfffc
```



We can see the branch predictor learns to branch and the predict taken flag matches the branch taken flag after a few cycles.



Shown above are the registers from the register file. We can see that the dual-issue design is advantageous here because the two registers $a0 and $a1 are updated at the same time. There is one stall in every iteration because the branch checks against $a0 and there needs to be a bubble between the branch and $a0 for the branch to be able to evaluate in the decode stage.

Conclusion

Through designing this processor, we learned a lot about determining possible hazards. These were usually given to us by the textbook, or were relatively simple compared to those we faced in designing the dual-issue processor. Since we worked with three single-issue versions of this processor in the past, we picked up on patterns of behavior that the processor exhibited when it's not behaving as intended. This made it easier to debug and sped up development. Just like with previous designs, sketching schematics and deriving hazards before implementing saved us a lot of time in the debugging phase.

When we learned about and implemented the pipelined processor, it became clear how we can optimize programs to run better on different kinds of processors. For example, in the pipelined processor, it was best to avoid branching or multi-cycle operations. In this processor design, it became clear that we would see improvement in CPI by running highly parallelized programs. Namely because multiple independent instructions can be executed at the same time. For repetitive and highly parallelized programs, we would benefit greatly from a multi-issue processor that could process more than two instructions at a time. This project gave insight into how modern CPUs and GPUs leverage parallelism as well.