Project Extra Credit

1. Branch Decisions in the Decode Stage

To implement this in our design, we moved all the jump and branch logic in the decode stage from the execute stage. That way, we didn't have to account for any control hazards, and we could resolve branches immediately without sending in NOPs to stall for any values. By moving the logic into the decode stage, we automatically gain -1 CP for a total of 2 CPI for all branches and jumps.

Furthermore, the pros and cons of this design change are determined by the area restraints surrounding the synthesis of the ISA. By moving the branching and jump logic to the decode stage, we clutter an already-busy stage of the pipeline. We must add a forwarding unit for RAW hazards, branch conditionals, an adder, and multiple muxes. However, this also frees space in the execute stage and allows it to only perform ALU operations. Overall, there's an increased chance of missing timing in the decode stage, due to the overhead by adding in additional logic.

Below is an example of a test that benefits from branch resolutions in the decode stage:

```
// Simple test showcasing the benefits of making branch decisions in decode
lbi r1, 0
lbi r2, 5
beqz r2, 4              // Should not branch
addi r1, r1, 5          // Branch is in decode here
addi r1, r1, 5          // Branch would resolve here if resolutions in execute
add r2, r1, r1
bnez r1, 10             // Should branch to the halt instruction
addi r1, r1, 5
addi r1, r1, 5
addi r1, r1, 5
addi r1, r1, 5
addi r1, r1, 5
halt
```

Our CPI from this was:

```
SUCCESS CPI:2.5 CYCLES:15 ICOUNT:6 IHITRATE: 0 DHITRATE: 0
```

2. Additional Forwarding Paths

Due to us resolving jumps and branches in the decode stage; to prevent stalling on RAW hazards, we needed to add additional forwarding paths. Specifically, we added forwarding paths from the execute stage, from the output of the ALU, and the input of instruction_ext_8 for LBI instruction (we finish writing the value of SLBI in the decode stage), and in the memory stage we forwarded the ALU output wire and the memory out wire from there as well, depending on the instruction in the memory stage, and finally we forwarded the register write data from the write back stage which is the same for any instruction. Additionally, in each stage, we forwarded pc_plus_two in case the RAW hazards encountered were from R7, which JR and JALR write to with pc_plus_two.

To highlight this forwarding in a more obvious way, here is another test we ran:

```
lbi r1, 1
lbi r2, 2
lbi r3, 3
lbi r4, 4
lbi r5, 5
beqz r1, 5      // Does not branch
beqz r2, 5      // Does not branch
beqz r3, 5      // Does not branch
beqz r4, 5      // Does not branch
beqz r5, 5      // Does not branch
lbi r6, 0
beqz r6, 2      // Does branch
halt
add r1, r2, r3
halt
```

Our CPI from this was:

```
SUCCESS CPI:2.6 CYCLES:21 ICOUNT:8 IHITRATE: 0 DHITRATE: 0
```

It's obvious here that if we had to stall for any of the register data for any of the stall hazards, we would have a CPI > 5. I believe that the overhead, once again, is fairly concentrated in the

decode stage because that's where the branch processing happens. In the other stages, it's only wires that come out of the latches and are connected to the forwarding unit, which forwards values to the branch condition unit in decode.

Explanation of Verilog Files in Directory for Extra Credit

Main directory:

- Verification (Contains all our self-made tests)
- Verification/extra_credit_tests/
    - Contains the two tests I mentioned earlier
- Verilog (Contains all our Verilog for the ISA)
    - Harrison_data_forwarding.v (The data forwarder in use for all other instructions (ignore forward_controller.v)
    - Branch_forwarding (Detects data hazards for the branch and jump instructions only and forwards data to branchCnd.v)
    - brandCnd.v (Contains the conditional logic for branches and jumps
    - proc.v instantiates harrison_data_forwarding and branch_forwarding in the top level of the latches, and the decode.v file instantiates branchCnd.v for the decode stage.