Checkpoint 1

1. How many stages is the datapath you've drawn? (i.e. How many cycles does it take to execute 1 instruction?)

A: We have three different stages in our datapath. Essentially, we treat BIOS, IMEM, DMEM and UART modules as registers in our pipeline because they synchronously read and synchronously write. Thus, we place pipeline registers in parallel with RAM modules which means that we eliminate one of the stages, namely the MEM stage.

# of stages = (# of positions of pipeline regs + 1) - # of eliminated stages
# of stages = (3 + 1) - 1 = 3 stages

Our 3-stage pipeline is WF/D/X

Why is WF together? Remember that RISCV151 implementation has a RegFile that is asynchronous read & synchronous write. This means that we can write to RegFile (WB) while fetching instructions (IF) simultaneously, thus WF!

If it is confusing, just think that RegFile acts like a register when we are writing data to it (synchronous write) and it acts like a combinational logic when reading data from it (asynchronous read). Furthermore, we can treat RegFile similar to IMEM where we divide the RegFile into two parts for clarity (they are still an instance of the same module).

WB + IF	D
RegFile (WB)	-
PC Reg (IF)	PC Reg (D)
BIOS or IMEM (IF)	RegFile (D)

It takes 3 cycles to execute 1 instruction

2. How do you handle ALU → ALU hazards?

addi x1, x2, 100
addi x2, x1, 100
A: We use Forwarding. Since we also want to handle branch data hazards, we are feeding in the value from the ALU in the previous cycle to the inputs of the branch comparator (in case the branch instruction has a dependency with a previous instruction). In this case, for example, the forwarded data will then move into the rs1 register (x1) for the second addi instruction and thus we will successfully handle the data hazard.

3. How do you handle ALU → MEM hazards?
addi x1, x2, 100
sw x1, 0(x3)
A: We use Forwarding here. In this example, we will take the result of the ALU from the previous cycle and then feed this into the rs2 register (where x1 in the sw instruction is) into our BIOS and DMEM block

4. How do you handle MEM → ALU hazards?
lw x1, 0(x3)
addi x1, x1, 100
A: We use forwarding. We get the result from the MEM stage and feed it into the rs1 and rs2 MUX's right before the branch comparators. This helps deal with any data dependencies related to branching. In the case that we do not need the result for branching, the data from MEM will continue to go through our datapath and in this case, the result from x1 will be passed into rs1 so we can successfully execute the addi instruction.

5. How do you handle MEM → MEM hazards?
lw x1, 0(x2)
sw x1, 4(x2)
also consider:
lw x1, 0(x2)
sw x3, 0(x1)
A: We use forwarding here taking the result of the first MEM instruction (from lw) and forwarding that result into either rs2 (so we can store the new value at an address) or putting the result into the ALU so we can calculate the new address we want to write a value to.

6. Do you need special handling for 2 cycle apart hazards?

addi x1, x2, 100
nop
addi x1, x1, 100
A: We want to avoid stalling to ensure we can keep our CPI low. Since this hazard is 2 cycles apart and the first addi instruction has not written back the result of x1 into the register file by the time the second addi instruction is in the decode stage, we forward the data. We forward that we are going to write in the writeback stage (going into the RegFile) and feed that into a mux prior to the pipelined registers at the end of the decode stage.

7. How do you handle branch control hazards? (What is the mispredict latency, what prediction scheme are you using, are you just injecting NOPs until the gbranch is resolved, what about data hazards in the branch?)
A: For branch control hazards, we will determine the result of the branch comparison in the execute stage. In the case that the branch is not taken, we want our program to execute as normal.

If the branch is taken, we want to stall 1 cycle/flush the instruction in the decode stage, and change our PC counter to where the branch's label indicates. So our prediction scheme is that we are predicting the branch is never taken. The mispredict latency is 1 extra cycle.

In terms of data hazard for the branch, suppose the branch instruction relies on a register value that has not been written back to the RegFile yet, we are using data forwarding from ALU and Mem to handle these cases by feeding in these forwarded values into the branch comparator to ensure we are avoiding data hazards.

8. How do you handle jump control hazards? Consider jal and jalr separately. What optimizations can be made to special-case handle jal?
A: Because we know the jal and jalr instructions will jump to another place in the program that is not PC + 4, we want to optimize this process rather than waiting for the entire instruction to execute to know where we want to jump to.

For jal, we will add an additional adder in the decode stage once we recognize that our instruction is a jal instruction. Since our new PC value is PC + Offset, we will add these two values from our instruction and existing PC value and feed that back into our PCSel mux so we do not have to have any stalls in the program and our program can begin executing the instruction at PC + Offset at the next cycle.

For jalr, we cannot use this additional hardware in the decode stage since jalr requires us to decode the rs1 value from the RegFile. Thus, we will execute the jalr stage in the execute stage by calculating rs1 + immediate and then feeding this result back to our PCSel mux. This will require one stall cycle because we do not want to execute anything in the program until we can jump to our desired location.

In both cases, since we are making optimizations, one thing that we have to do is ensure we are writing PC + 4 back to the rd register. To ensure this, we will have the jal and jalr instructions continue to go through the datapath for the purpose of writing PC + 4 back into rd.

9. What is the most likely critical path in your design? I think the CP is longer than just one phase. I will look into this now.
A: The most likely critical path in the design is during the execute phase. This is because this phase contains 2 MUXs, the branch comparator and immediate generator, another MUX fed into the input of the ALU and finally the ALU itself.

10. Where do the UART modules, instruction, and cycle counters go? How are you going to drive **uart_tx_data_in_valid** and **uart_rx_data_out_ready** (give logic expressions)?
A: The UART modules, instruction, and cycle counters go in the "Mem"/pipelined register phase (specific to our design). The I/O memory map is used for the UART as well as the instruction and cycle counters which will be used to determine CPI.

Based on the spec, we are not responsible for setting uart_tx_data_in_valid and uart_rx_data_out_ready as the software is, but we do know whether we are writing or reading from the UART based on the address that is passed in and whether it is read or write access. If it is a read access we will be dealing with the load instructions including lw, lb, lh, lhu, and lbu and for write access the store instructions including sw, sh, and sb. Additionally, for the load instructions we assume that the software has uart_rx_data_out_ready set to 1 to know we can read new data from the UART and for store instructions the software sets uart_tx_data_in_valid to 1 so we send data to the UART.

11. What is the role of the CSR register? Where does it go? (check if addr 0x51E is in the instruction)
A: We know that the CSR register is independent of RegFile and Memory modules and thus we can place them either on the Execute (X) or Write Back + Instruction Fetch (WF) stage. The role of the register is to indicate the status of the simulation including if it's running or if it has ended. If we get a 1 in the CSR register it will indicate that the program was successfully run and if we get a value greater than 1 then it will indicate we have failed a test.

As a result, the CSR register must be anywhere on the data path as long as each instruction has access to it so the test is aware of the state of the program. In this case, we are putting it in parallel with when our memory reads and writes occur.

12. When do we read from BIOS for instructions? When do we read from IMem for instructions? How do we switch from BIOS address space to IMem address space? In which case can we write to IMem, and why do we need to write to IMem? How do we know if a memory instruction is intended for DMem or any IO device?
A: We read from the BIOS for instructions when the memory address from [31:28] is 4'b0100 and the address type is PC or Data. We read from the IMEM for instructions when the address at [31:28] is 4'b0001 with address type of PC and read-only access. We switch from the address spaces based on the PC[30] value which is fed into the Mux between the 2 memory blocks. We switch from the BIOS address space to IMem space by differentiating the memory addresses.

We write to IMEM when our [31:28] address is 4'b001x and our address type is Data. We write instructions to the IMEM when we are loading in our program. These instructions are coming from the UART and we are writing those instructions into the IMEM. Once PC[30] is set to 1 then we know we begin reading from the IMEM instead of the BIOS for instructions.

Additionally, when PC[30] == 1'b1 then we write to the instruction memory. Similarly, we know a memory instruction is intended for DMEM or an I/O device based on the [31:28] address.

Miscellaneous

Todo list:

1. Questions for check off
    a. Walk through the 2-cycle hazards
    b. Mem-to-ALU and ALU-to-ALU should be forwarded prior to the branch comparator but not Mem-to-Mem or ALU-to-Mem
    c. Ensuring successful counting of instructions and cycles
2.

Objectives of Project: