This code is a modified simulator to account for pipelining multiple processes at a time. Instead of one process running at a time, we modified it to allow for multiple processes to be running on different stages. By having multiple process stages running during a single clock cycle, we significantly cut down on the total time a program runs.

It first reads in machine code line by line and places instructions into registers and memory. It initializes all stages to noop's and 0's in the buffer values before filling them with passed in instructions. Then it fetches program counter 0 (pc-0) to start the pipelining. There are five stages in pipelining, instruction fetch stage, instruction decode stage, execute stage, memory stage, and writeback stage. Pc-0 would begin at the instruction fetch stage during the first cycle. During the second cycle, pc-0 would move to the instruction decode stage while pc-1 would begin in the instruction fetch stage. After each cycle, each respective program counter is moved on to the next stage until the fifth stage, write back, where by this point each respective type of instruction would have completed its operation. In this example, at cycle five, pc-0 is in the writeback stage and will be completed when cycle six begins and pc-1 moves into the writeback stage.

Data hazards occur when an operation is completed but it didn't do what it was supposed to because it had the wrong information. When it comes to hazards with branching, we predict the path not taken, meaning we just use the next pc assuming the branch was incorrect, therefore there is no reason to change the pc. We continue on with the next 3 pcs until the branch reaches the writeback stage. If we predicted correctly the pipeline continues on since all of the next operations are the correct operations to be completed. If we predict wrong, the previous three program counters are flushed, meaning we turn them into noops because they are the incorrect operations to run. The correct pc is now right before the start of the pipeline since the noop's removed the incorrect pc's and it will be fetched in the next cycle.

Data forwarding is important because there are five different stages running simultaneously. If one registers information is changed before the writeback stage, and another process tries to use that register, it will have outdated information and the output will be incorrect. If one of your registers has been updated in the past three cycles it must be forwarded in order for newer instructions to have the most up to date information. The way forwarding works is only add nand and load word changes the contents of a register. If you are an operation that uses a register, it must check the previous three cycles completed to see if there was an add, nand, or load word operation that had a destination register that the current operation is going to use as input. To implement this idea, we look back at the past three instructions to see if there was an add, nand, or load word. If there was, then we check if the destination register that operation is writing to is one of the current registers we are trying to use. Then we just take the most up to date value from that destination register and use it instead of our current register value to compute the current instructions operation.

Load word stalls are the last thing we had to implement. Similarly to forwarding, if our current operation is a load word, we check in the execution stage if the next instruction uses the register we are loading. In this case the previous instruction is forced to remain at its current stage and in front of it a noop is inserted to give the load word time to complete the writeback stage. Since it has been given a one stage gap, the load words destination register will be forwarded to the previous instruction giving it the most up to date information for the register it is using.

Some things we struggled with were implementing the forwarding because at first we didn't know how to translate the idea of forwarding into code. Another thing we struggled with was knowing which value to pass to the next buffer because each operation has its own order of values in the machine code. The last thing we struggled with was the fetches and retired. For some reason we struggled to get the test cases to match our output when it came to the fetch and retire tests.