Objective: The objective of this lab was to create a program that would handle and show the pipeline steps of a RISCV-5 CPU. This includes data hazards with forwarding by using the examples we learned in class and the equations given to us in the lab document. This lab was also to learn how to implement pipelines in C and then print out the process that they do. This was done over the course of 5 weeks by Jack Burgess, Ryan Clausius and Zach Millsap.

Overview: In general our program takes in some RISC-5 hex values that are then decoded and ran using the pipeline that we created in this lab. The pipeline is broken down into 5 different parts. These being the Fetch stage, Decode stage, Execute stage, Memory state, and then finally the write back. After this we also created a show pipeline function that would print whatever stage of the pipeline had just finished. This was done by telling the show pipeline function what part of the cycle we were on by a hardcoded int value. Then we would pass along a global instruction which would be printed out into the pipeline.

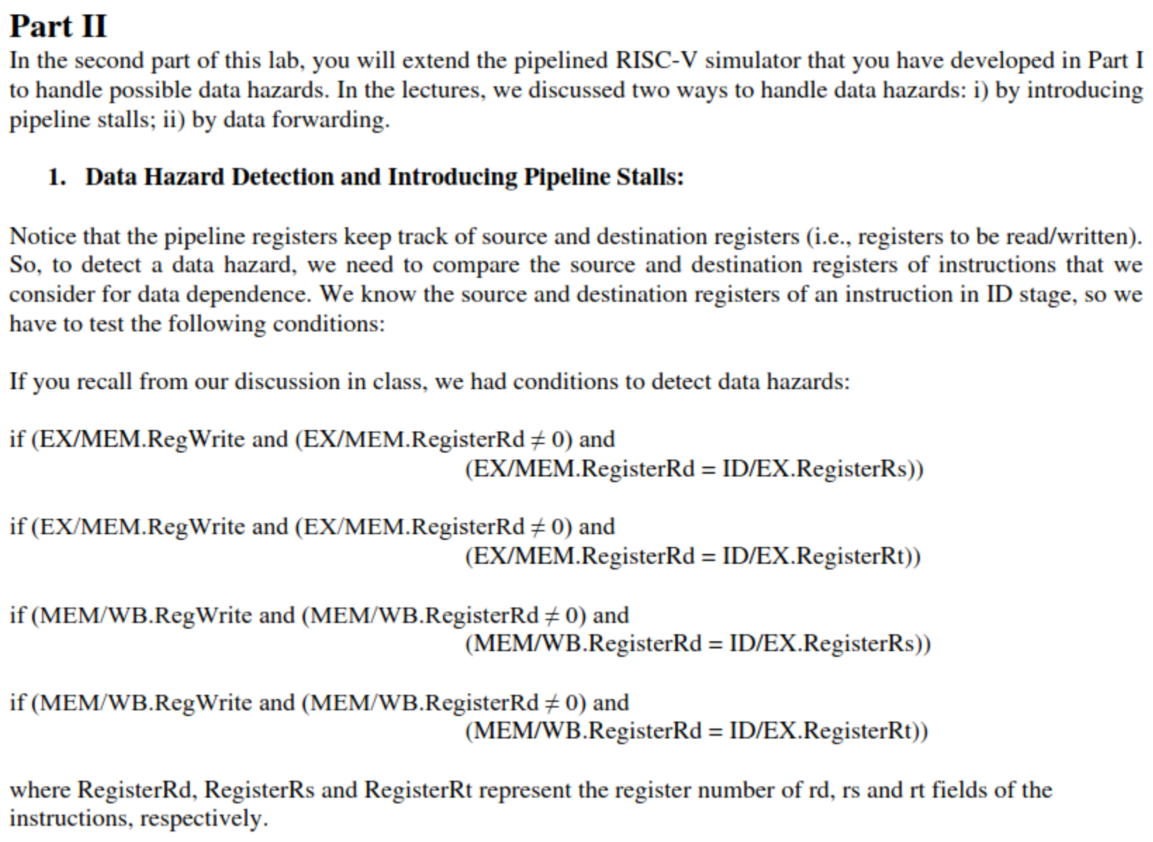
To understand the process we must first talk about how the pipeline will read the 5 stages. First it will do the Write back stage, then the mem, then EX, then Decode, then IF. This is because we must do the final parts of the pipeline first to free up those spots to move the early parts of the pipeline into those values. First was the fetch phase which was created by setting a global variable addr equal to the next PC state. This was then stored in the IF and ID register. This was then read by the mem read function that was given to us in the skeleton code. The Next state PC was then moved by 4 to account for the shift in instructions. This register is global and would be accessed by the DE function.

The Decode function is very similar to our lab 1 and 2 which was focused on decoding hex values to actual RISCV - 5 instructions. Using the code we created in lab 1 and 2 we reserve engineered a function that would read in the hex values and create a command. This could would be stored in the Decode register which would be the end of this pipeline stage.

The Execute stage would take in the op-code array created by the decode function and then command the ALU output to change based on what the instruction was. In our code you can see that if it is an ADD type then the function will change the output and then store it into the new register. If not then this will cause an IMM to form. Which is also accounted for by adding the IMM to the execute stage.

In Mem we would once again check the opcode and then use the mem write function that was given to use in the skeleton file. This would save the ALUoutput into the MEM_WB register. Then in the Write Back stage. We load this into the MEM_WB register to be used in our show pipeline code. It does this by reading the mem read function with whatever was in the EX_MEM register. Our show pipeline is created by saving a string of each instruction from each stage and then printing it out whenever that step of the pipeline is completed. This will cause the pipeline to get printed from Write back first then IF last.

Part 2 was very challenging as this code was somewhat hard to understand and implement since the skeleton file was not created by our team. There are 4 things we must check for whenever we are doing a hazard detection. You must check to see. This is given by this screen shoot.



Part II

In the second part of this lab, you will extend the pipelined RISC-V simulator that you have developed in Part I to handle possible data hazards. In the lectures, we discussed two ways to handle data hazards: i) by introducing pipeline stalls; ii) by data forwarding.

1. **Data Hazard Detection and Introducing Pipeline Stalls:**

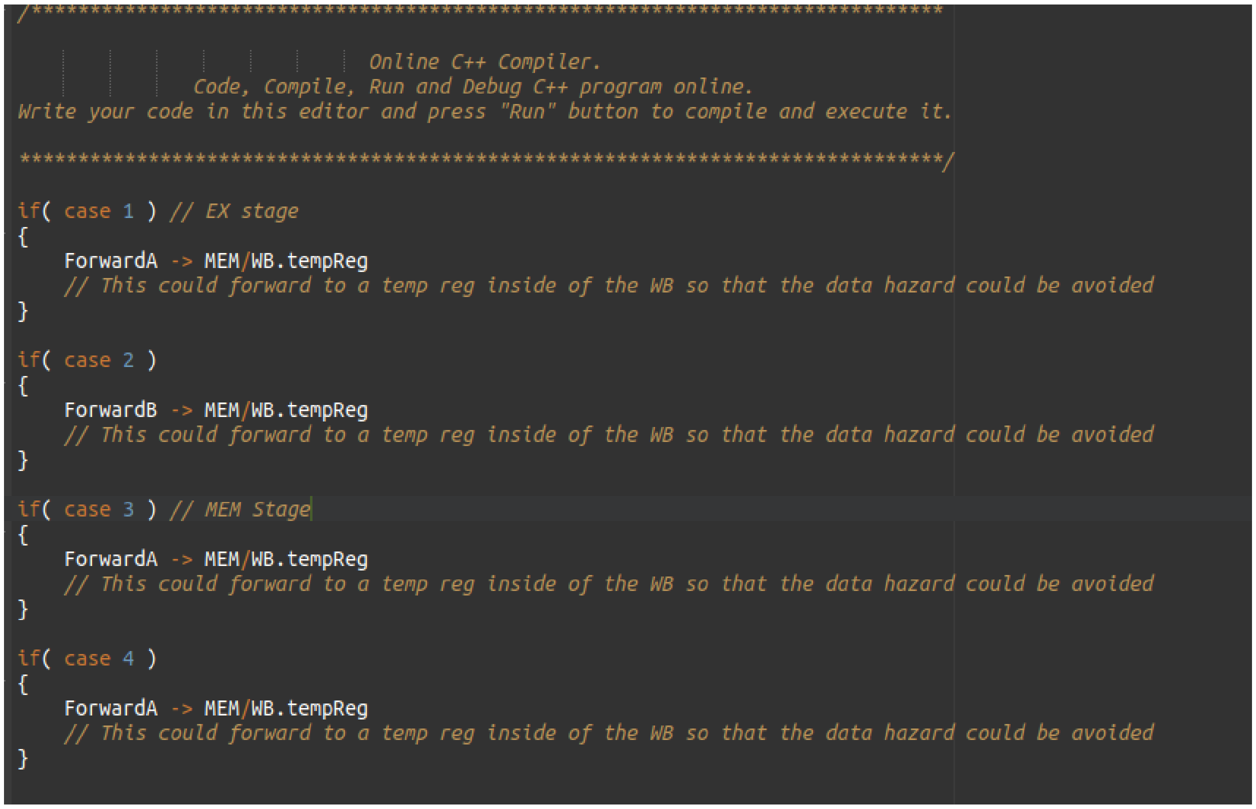
Notice that the pipeline registers keep track of source and destination registers (i.e., registers to be read/written). So, to detect a data hazard, we need to compare the source and destination registers of instructions that we consider for data dependence. We know the source and destination registers of an instruction in ID stage, so we have to test the following conditions:

If you recall from our discussion in class, we had conditions to detect data hazards:

```
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and
                                        (EX/MEM.RegisterRd = ID/EX.RegisterRs))

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and
                                        (EX/MEM.RegisterRd = ID/EX.RegisterRt))

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and
                                        (MEM/WB.RegisterRd = ID/EX.RegisterRs))

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and
                                        (MEM/WB.RegisterRd = ID/EX.RegisterRt))
```

where RegisterRd, RegisterRs and RegisterRt represent the register number of rd, rs and rt fields of the instructions, respectively.

These are the equations that were used to create the different if else statements for our lab 2. We had issues finishing this however as we could not complete the task of using the pipelines to forward to the next. This would make our code Seg fault. As such we did not have this in our file submission. Instead I have provided some Psuedo code of how it would have worked. In our code itself we have the logic state and when the date hazard checks would be triggered.



```
/******************************************************************************

                              Online C++ Compiler.
               Code, Compile, Run and Debug C++ program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

if( case 1 ) // EX stage
{
    ForwardA -> MEM/WB.tempReg
    // This could forward to a temp reg inside of the WB so that the data hazard could be avoided
}

if( case 2 )
{
    ForwardB -> MEM/WB.tempReg
    // This could forward to a temp reg inside of the WB so that the data hazard could be avoided
}

if( case 3 ) // MEM Stage
{
    ForwardA -> MEM/WB.tempReg
    // This could forward to a temp reg inside of the WB so that the data hazard could be avoided
}

if( case 4 )
{
    ForwardA -> MEM/WB.tempReg
    // This could forward to a temp reg inside of the WB so that the data hazard could be avoided
}
```

Our thinking for the data hazards is simple. If one of the cases is hit then we need to forward the current register to whenever the stall is happening so we can quickly resolve it. This was talked about both in class and in the lab manual. We also created a global int called EnableForwarding that would bet set to 0 or 1 (bool). This was used in the EX stage. However since it was not used we set it to 0. However in our handle pipeline function we checked to see if the enable would work. While we were not able to fully implement the data hazards in our code we believe that we had created the create logic to check and see if there was a pipeline error. To fix this in the future we could finish the Psuedo code given above and use the pipeline to better solve this issue.

Struggles and Challenges: This lab was the hardest lab that any of my group members had encountered but to this point. This was challenging as we had to use the knowledge from our older two labs and use a new skeleton code. The hardest part was fully understanding the skeleton code base given to use and using it to implement the pipeline in full. This was solved by using the global pipeline registers and saving each value to its assigned spot. However some of our code from lab 2 did not perform its purpose which is shown when printing the pipeline.

I would say that the hardest part was getting the correct syntax and function flow to accurately know what was going on in our pipeline. It was very hard to detect errors in our program as most of the errors were seg faults or were caused by earlier parts of the pipeline that another group member was working on. To solve these issues we had to spend time on each stage as a group and talk through how we would be implementing each stage so that which member was working on the next stage could accurately complete it.

Milestones and Implementation: Here are some key moments and milestones that we thought were worth sharing. These are in order in which they happened.

- Group was created and the original code was reduced to a compilable form
- Lab 2 was referred to as we needed some functions for the DE stage
- The skeleton code was reviewed and the new variables were learned
- The RISCV-5 sim was learned
- Some global variables were created and the show pipeline and handle pipeline were created
- Zach completed a file reading function

- Jack created an input file by transforming the Instructions to Hex
- Ryan & Jack Completed the Decode and EX function
- Jack Completed the IF function
- Zach completed the MEM and WB function
- Ryan completed the handle pipeline and made the 5 different functions work using global registers
- Some helper functions like Hex to Binary were completed by the group (DECODE)
- Part 2 was started
- The part 2 code was scrapped as it could not handle the instructions and would seg fault
- The lab report was created
- Part 2 was cleaned but and some pseudo code was created to show our thinking on how to handle the data errors
- The lab was turned into a zip file
- This file was checked and submitted

I found that we were able to complete all of part 1 of the lab with little issue. Some of the decoding from lab 2 would not always work and this shows in our final submission. However we would accurately call and use each part of the pipeline and how it using the show pipeline function.

However part 2 was very difficult. Like I mentioned earlier as we were able to detect errors any further progress would cause our code to seg fault. This was frustrating as part 1 was working and we could show the pipeline. We are disappointed that we could not fully complete the second half of this lab however we believe that our thinking is accurate and the code and documentation can show that we were on the right track. With more time we believe that this could have been completed.

Work Done by Group:

Zach : Created the File Read and the main function. He also created the memory and the EX function. This was done by using the register and calling the Offset created by the PC counter and the IMM. Zach Also helped with the input file.

Jack: Created the Input file by translating instructions. Completed the IF function which would grab the instruction. Helped complete the Decode file and WB file with Ryan. Created the lab report and the show pipeline function. With the pseudo code and the explanation. Did the show pipeline function with Ryan

Ryan : Created the decode and WB file. He also did most of the EX function. Ryan also created the data hazard detection in part 2. Which was the only part of part 2 that was done successful. He created all of the global variables and used the skeleton code to get all of the functions to speak with each other. He also did the show pipeline function with Jack