Nasr Alae-eddine
119010531
CSC3050 Computer Architecture

Assignment 3 report

Development environment

- Pre-configured VM provided by the TAs

Supposedly : (Reference from the BB VM setup)

- Linux Kernel: 5.10.5 (zen)
- GCC Version: 10.2.0
- Clang Version: 11.0.0
- GNU Make: 4.3
- CMake Suite: 3.19.2
- Ninja: 1.10.2
- Icarus Verilog: 11.0

Overview and How to Run:

The project is written in numerous verilog files which is all accounted for by the *Makefiles* provided under the directories src/alu and src/cpu. While being inside any of these two typing "make test" in the terminal will run the respective test of the alu or cpu.
The Design of the CPU has been made to handle the *Hazards* and in practice was able to run all the test cases provided with the project requirements

Project Goal

In this project, we are required first to implement an important computation unit in CPU, the Arithmetic and Logic Unit (ALU). In verilog this module is supposed to be able to take in the raw full instruction, decode it to know which exact operation to perform and which registers to use. The output is the 32 bit result of the operation on the provided 2 inputs.
Then Based on the ALU as the main computation the objective is to implement a 5-stage pipelined CPU which can execute the MIPS instructions and keep track of the memory.

Design

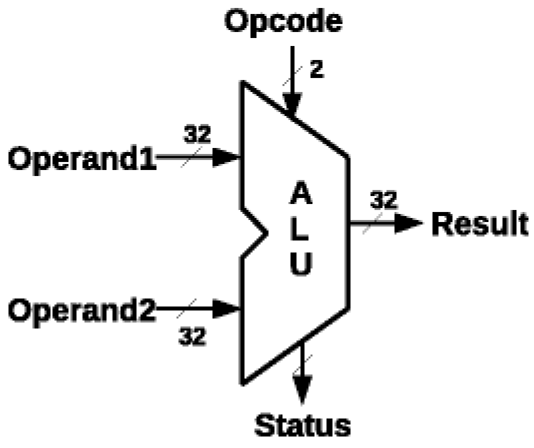
ALU :

The ALU module will take in two 32-bit binary numbers, A and B, and perform the following operations depending on the given 32-bit MIPS instruction code:
-add, addi, addu, addiu

- sub, subu
- and, andi, nor, or, ori, xor, xori
- beq, bne, slt, slti, sltiu, sltu
- lw, sw
- sll, sllv, srl, srlv, sra, srav

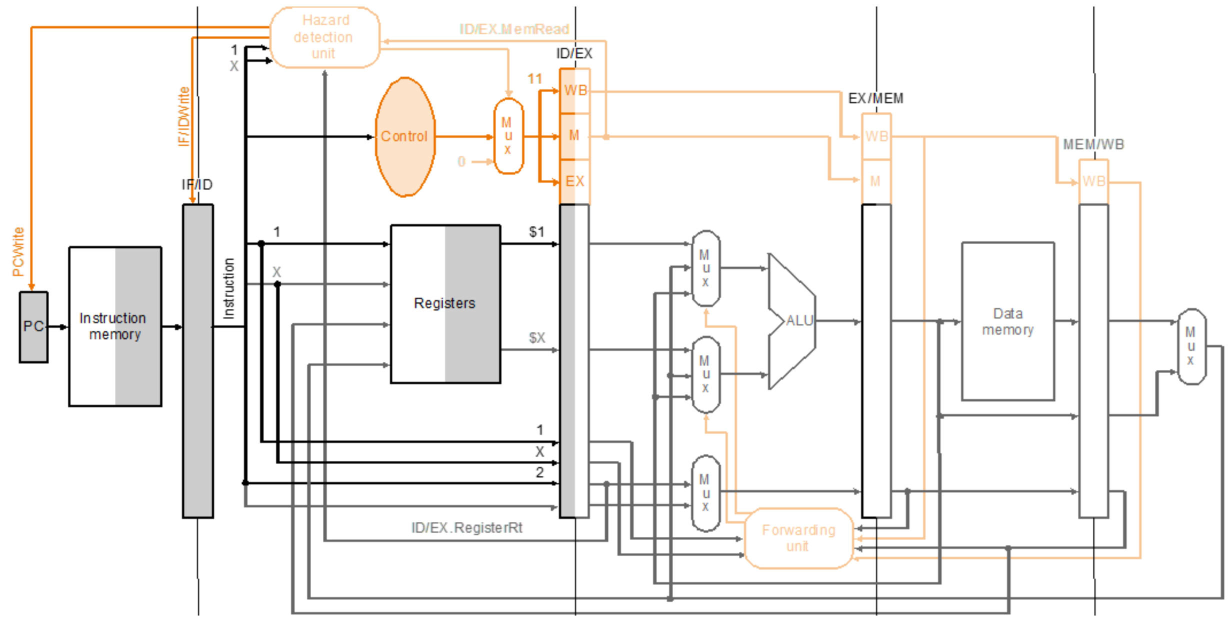
The output of the ALU module will be a 32-bit binary number that represents the result of the operation and a 3-bit flag specifying overflow, negative values and zero.

The design is relatively simple. Just check for all different cases first for R-type opcode and funct and then implement the I-type instructions and Jump output.
But I must say that the alu.v used in the pipelined cpu is different since it takes in a 3 bit ALU control signal rather than the whole instruction so it is simpler and the decoding is done elsewhere:



Pipelined CPU:

A 5-stage pipelined CPU is a processor architecture that breaks down the execution of instructions into five stages: instruction fetch, instruction decode, execute, memory access, and write back. Each stage in the pipeline operates on a different instruction simultaneously, increasing the processor's throughput and reducing the overall execution time. However, the pipelined CPU is also prone to hazards that can lead to incorrect program execution. To prevent these hazards, we implement hazard detection logic that stalls the pipeline when necessary. Even if the project is heavy on workload and hard to implement. The actual design of the cpu is clear and can be confined to dataflow charts such as the one below:



Inspired by this design my first idea was to first implement each module individually and then link them up (wires) to complete the project. Note : through debugging I may have swayed a little from the exact flow graph as some modules performed better when given certain inputs … etc but in general the same main idea is followed.

Modules

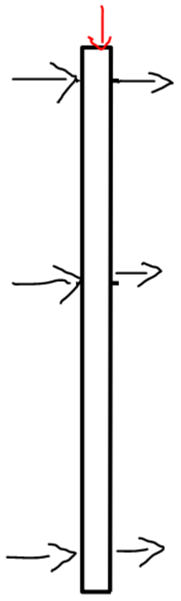
To implement a 5-stage pipelined CPU with hazard detection, we break down the problem into smaller problems and implement various modules for each stage in the pipeline. We start by implementing the instruction fetch stage, where we fetch instructions from memory and feed them to the instruction decode stage.

During this stage the InstructionRAM.v used is the one provided by the teachers so no need to elaborate further on that. My implementation was just the PCBuffer,the PCAdder and a Multiplexer to choose which PC to send back into the buffer to have a controlled continuous flow of the accurate Program Counter depending on the situation (normal + 4 or branch or jump).

In the instruction decode stage, we decode the instructions and generate control signals that are used to control the subsequent stages in the pipeline.

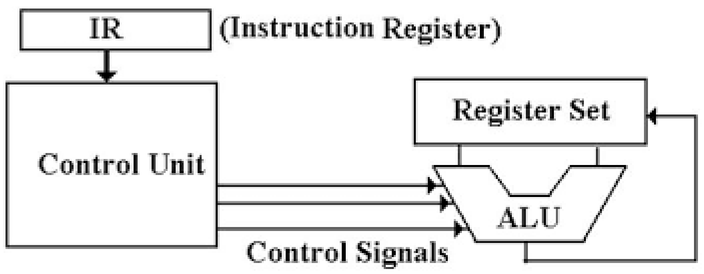
Between any of the two stages we use buffers that take in a bunch of inputs and output them with no change unless a stall or reset signal is sent. This makes sure the pipelined nature of the CPU is maintained.

IF_ID,ID_EX,EX_MEM and MEM_WB buffers:



The Control Unit takes in mainly the instruction 32-bit and also a few signals sent back from later stages and is the heart of the cpu in how it manipulates the signals changing the behavior of almost every other module depending on what instruction is being read. This Unit also calls onto the Hazard Units to deal with any danger that comes up.

The Regfile holds all the registers. It supports writing values into these registers with a 5 bit indexing. And it also can read values with 2 output wires out



Next, we implement the execute stage, where we perform arithmetic and logical operations on the data. This is taken care of by the ALU that was discussed previously.

The memory access stage follows, where we read or write data to/from memory. Here we use the MainMemory module provided by the teacher with no change.

Finally, in the write back stage, we have a multiplexer to choose a write back to send to the stages back in the register file.

Small modules have been skipped for the report such as Muxes and Comparator.

Hazards

To prevent hazards, we implement hazard detection logic that identifies when a hazard occurs and stalls the pipeline until the hazard is resolved. For attempting to use the same resource in two different ways at the same time we need to wait and that is done by stopping some buffers in the ID stage and waiting for updates from further stages.

We also implement a forwarding unit that allows the execute stage to forward its results to the subsequent stages in the pipeline to prevent data hazards.

Dealt with in a different way are Branch/jump hazards that occur when the CPU encounters a branch or jump instruction that changes the program counter (PC) and disrupts the normal execution flow of instructions in the pipeline. To deal with branch/jump hazards, we need to predict the outcome of the branch/jump instruction and take appropriate actions.

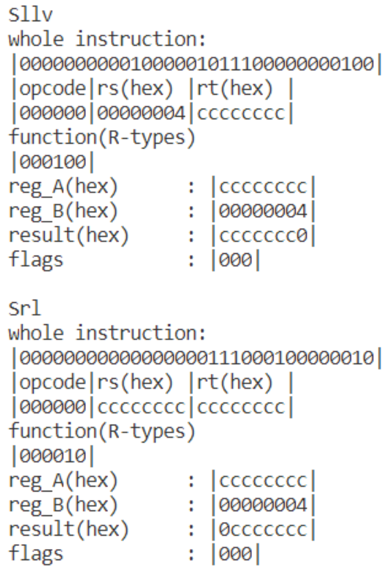
In the ID section I have talked about a multiplexer which chooses the accurate PC to forward, this part is crucial to make sure the hazard is dealt with.

NB: Since this strategy is implemented J and Branch instructions will end up stalling for a cycle

(no jump = retake normal process flow from IF)

Testing:

All the instructions for the ALU were tested under src/alu/test_alu.v with different values made by me as a personal test. The results were verified by just myself computing the result with a calculator and my organic brain. Here is a visual output for some instructions:



```
Sllv
whole instruction:
|00000000001000001011100000000100|
|opcode|rs(hex) |rt(hex) |
|000000|00000004|cccccccc|
function(R-types)
|000100|
reg_A(hex)      : |cccccccc|
reg_B(hex)      : |00000004|
result(hex)     : |ccccccc0|
flags           : |000|

Srl
whole instruction:
|00000000000000000111000100000010|
|opcode|rs(hex) |rt(hex) |
|000000|cccccccc|cccccccc|
function(R-types)
|000010|
reg_A(hex)      : |cccccccc|
reg_B(hex)      : |00000004|
result(hex)     : |0ccccccc|
flags           : |000|
```

The CPU was tested first by some easy personal mips tests I made. These were left in /test_case but the testing was more focused on the 8 cases provided by the teachers. Running make test with a test loaded in CPU_instructions.bin will output the Memory in data.bin