5-Stage Pipelined RISC-V Processor

Team ECEasy – Eric Chen, Gerardo Porras, Bryan Thai

Introduction

In this project, we created a pipelined implementation of the RV32I processor. In the previous assignments, we worked on making a regular RV32I processor. In a regular RV32I processor and the LC3 processor, we had to wait for an instruction to finish before fetching the next instruction to execute. This implementation of the processor is undesirable because of its slow nature. A pipelined implementation of the processor means that instead of waiting for the current instruction to finish before fetching the next instruction; we will improve the performance of the processor by overlapping the execution of instructions. The reason why we would want to overlap the execution of instructions is because a pipelined implementation has a significant speedup compared to the regular processor due to the increased throughput.

Project overview

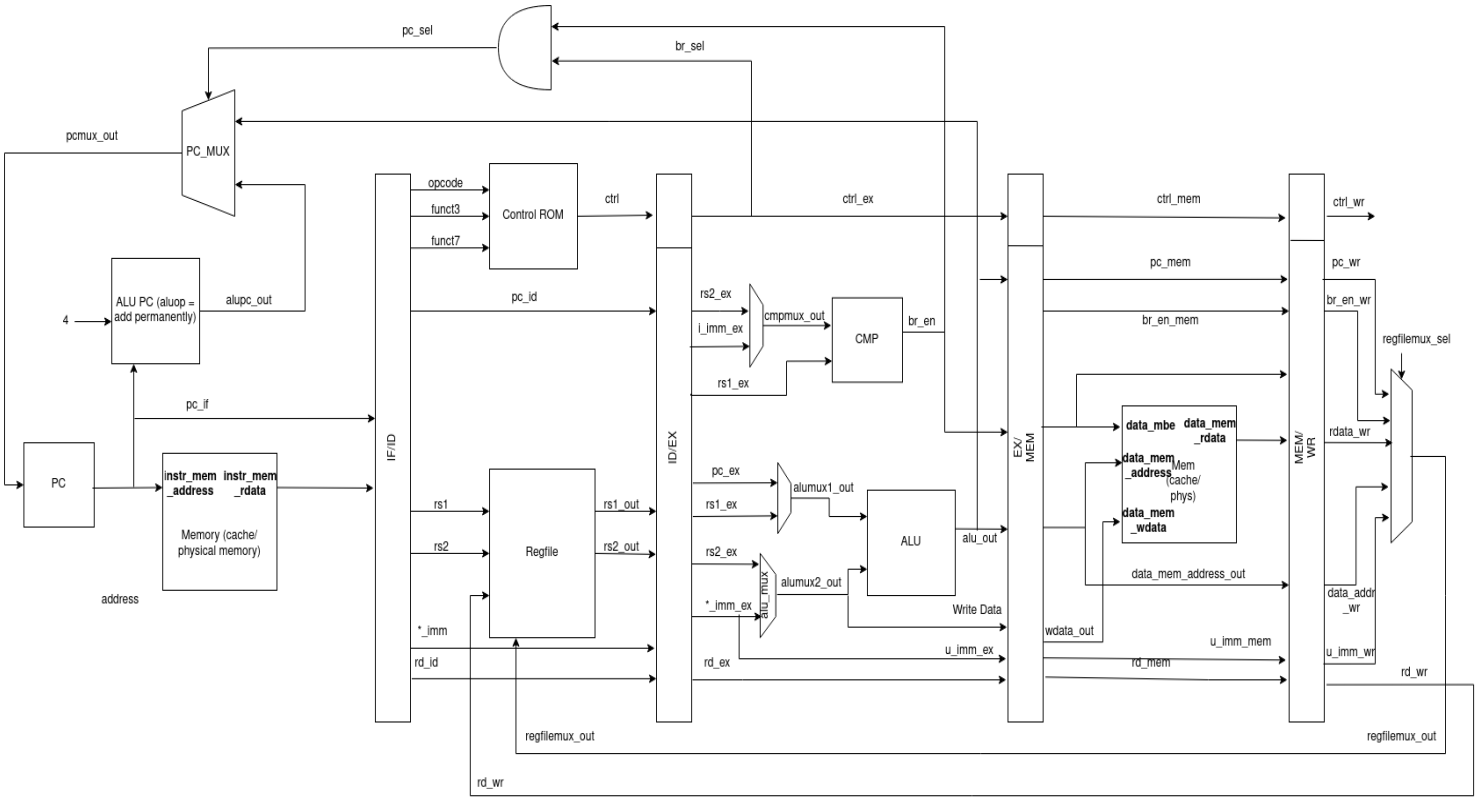
This project was a group effort to finish. For every checkpoint the team would meet up to discuss the design of the pipelined datapath. The team would then discuss multiple free days to work as a group. While beginning to program the pipelined datapath, we had a dedicated person update the datapath design whenever it was discussed that we were either missing paths or components in the design. Work did not get split up onto individuals until starting to work on advanced features for the processor. For the advanced features each individual took an advanced feature to work on on their own and later reconvened to check on the progress and help each other in the process.

Design description

Overview

The project was to design a pipelined RV32I processor. The implementation of a pipelined RV32I processor comes with its difficulties and problems that need to be resolved. The pipelined RV32I processor faults are with the data and control hazards that come with. Data and control hazards ensure that the correctness of the processor to be questionable. Data hazards were resolved by implementing a forwarding unit and a transparent register file. Control hazards were resolved by implementing a hazard detection unit and a branch predictor.

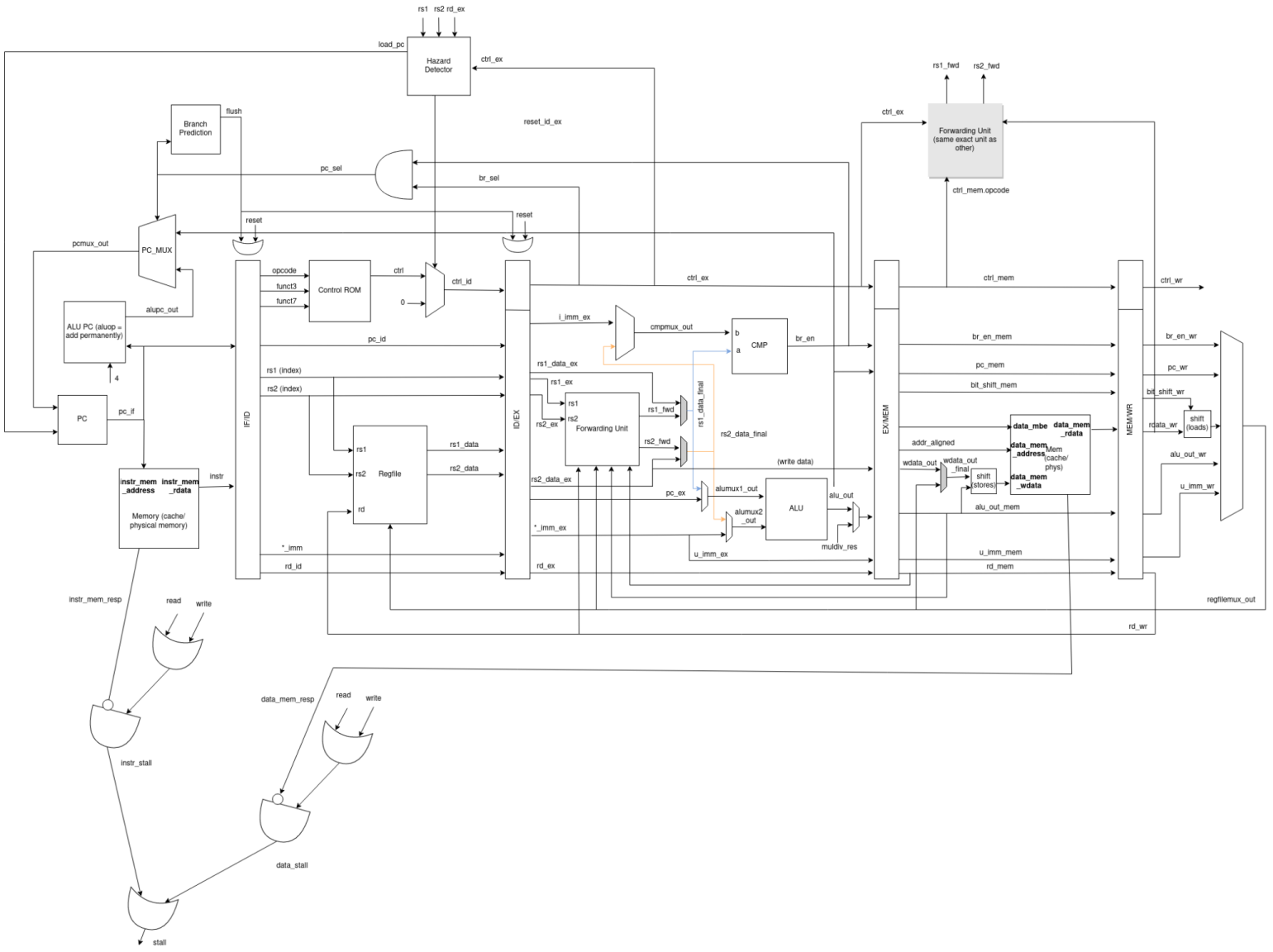
Milestones: Checkpoint 1



In Checkpoint 1, we set up a basic pipeline that can handle all of the RV32I instructions. This pipeline consists of the basic components that a processor would have like a PC register, Regfile, ALU, and etc. The main difference between the pipelined processor and the regular processor is this new component control ROM and the stage registers. In this pipelined processor the PC will constantly get a new value every clock

cycle meaning we will fetch a new instruction at every clock cycle. This new instruction will then be placed in the IF/ID stage register where it will be stored before a new instruction is fetched. While this instruction is stored at the IF/ID stage register, it will run through everything in the ID stage – most importantly the control ROM. The control ROM will take the opcode, funct3, and funct7 of the instruction to initialize a control word which will be a struct that contains the control signals for the corresponding instruction. The control word will go down the pipeline changing the control signals to its corresponding control signals, eventually leaving the pipeline once it reaches the write back stage.

Milestones: Checkpoint 2



In Checkpoint 2, we implemented solutions to the hazards and added L1 caches. The Forwarding unit was placed in the execute stage and changes the values for rs1 and rs2 whenever there is a conflict between the rs1, rs2 addresses and the rd address in either the mem or writeback stage then change the corresponding conflicted register value to the rd value. The hazard detection unit plays two roles, one role is to take care of read after load conditions and stall pipeline if both caches are to be serviced. In the case of the read after load condition, the hazard detection unit will check if the rd address in the execute stage is the same as either the rs1 address or the rs2 address; if this is the case and the current opcode at the execute stage is load then stall the IF/ID stage register and the PC register. This will allow the Regfile to be updated with the correct value and the next instruction will

take the correct rs1 or rs2 value. If the both caches are waiting to be serviced then stall the entire pipeline and wait for the both caches to be serviced before resuming the pipeline. We also implemented a static branch predictor which in our case will always assume not taken meaning we will have the PC add 4 unless the branch is taken which will change the value of PC to the new branched instruction address and reset the IF/ID and ID/EX stage register which would be the predicted instructions. We also changed the Regfile to be a transparent Regfile; this means that the Regfile will check if the current rs1 or rs2 addresses are equal to the incoming rd address and change the values of rs1 or rs2 to the new incoming rd value if condition is met.

As previously mentioned, the L1 caches were a major part of this checkpoint as well. To reduce the number of memory access conflicts, our design featured two caches: one for instructions (useful in the fetch stage) and one for load/store instructions (useful in the mem stage). As with any memory hierarchy, the caches were placed between the CPU and physical memory, with the main goal of decreasing memory access times. The cache was a 2-way set associative cache, and featured a least-recently used replacement policy. Cache lines would only be written to physical memory upon eviction, and under the condition it had been modified since it entered the cache, as specified by the dirty bit. One challenge with using two separate caches on the same level is that we only had one bus to physical memory. This meant that only one cache could be serviced at a time, whether it was requesting data from physical memory or trying to write back a cache line. To resolve this, we introduced a new component: the arbiter. The arbiter is designed to choose which cache (instruction or data) to service in the case of a conflict. For our design, we prioritized the data cache over instruction cache as this would reduce the number of stages that would be stalled behind a memory miss. Based on which cache it chose to service, the arbiter would send appropriate signals to physical memory, via a cache line adapter, so that the data could be handled correctly.

Milestones: Checkpoint 3

In checkpoint 3, we attempted to implement various advanced design features, including branch prediction, support for M-extension instructions, as well as a cache prefetcher. While we were unable to fully implement these features, we were also made aware of issues concerning our forwarding unit. In the process of debugging this, we also decided to completely overhaul how we stored our CPU data, by creating a reg_word struct that would contain relevant information about register addresses and data.

Advanced design options

2-state Bimodal Branch Predictor

The branch prediction module utilizes a parent module to interface between our datapath and each predictor algorithm. By doing this, we are able to adapt various different predictor models to our code without having to change much. It uses inputs from the instruction register to generate a predictor word that gets passed along the pipeline, with all the necessary prediction variables, namely the prediction, the target address to branch to, and the alternate address in case of misprediction. We also use a misprediction flag that gets set in the EX phase. Once this word enters the EX phase, the misprediction flag is set, accuracy counters are updated, and we either flush the pipeline, or continue on.

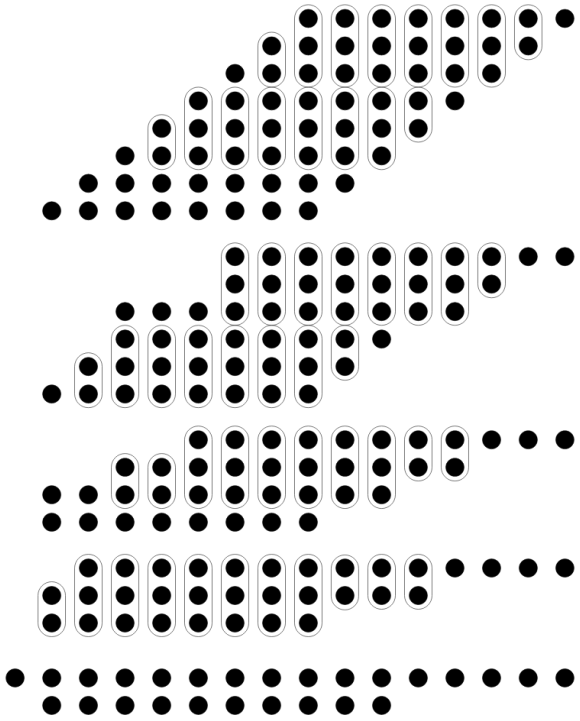
To test this, we simply ran the code and saw if it provided any performance benefits, while also keeping track of the accuracy counter. As we finished the implementation late, the accuracy is moderately low, hovering around 60% depending on the style of branches we encounter (loops, vs conditional jumps, etc). I believe this is currently a result of the gap between the generated prediction and the misprediction, as we don't change the state until we reach the EX stage and if we have consecutive branches such as in the CP2 code, it will mispredict twice before the states begin to change to accommodate mispredictions.

RV32 M-Extension

One advanced feature we were able to fully implement was the RISC-V M-Extension. This extension introduces support for multiply and divide instructions, which are missing from the baseline instruction set. In order to do so, we needed to implement a multiplier and divider in hardware.

For the multiplier, we originally considered using a basic shift-add multiplier, in which we repeatedly add shifted versions of the multiplicand to itself. However, after a little bit of research we decided that a more advanced multiplier would be reasonable to implement and provide more speedup benefits. One that piqued our interest was the Wallace Tree multiplier. This multiplier executes in three stages. In the first stage, we take partial products. For us, since we had 32 bit operands, we would have $32^2 = 1024$ total partial products. In a graphical representation, these are arranged in a parallelogram to represent the different weights (1s place, 2s place, etc.). In the second stage, groups of 3 rows are added together and compressed into groups of 2 rows. This

process is repeated until 2 rows remain in the end. In the final stage, these 2 rows are added, giving us the final sum.



For our divider, we decided to implement a basic long division algorithm. This performs by subtracting until we get at or below 0. Our accumulator keeps track of how many times we subtract, and represents our quotient. To test both the multiplier and divider, we had to enable the risc32im architecture, and could then write test code using multiply and divide instructions, ensuring all 64 bits of multiply instructions (2 registers) and the quotient and remainder for divide instructions were correct.

Challenges included implementing the multiplier and divider into our existing design. Namely, that meant being able to account for data hazards effectively. To solve this, we needed additional MUXes that would determine whether our arithmetic output was created by the ALU or multiplier/divider, and send this data forward. Another challenge was having to account for signedness. This fix was not too bad either, as you simply take the magnitude of both numbers, and tack on the sign at the end depending on how many negatives you began with.

Conclusion

This project proved to be somewhat challenging at times, but definitely taught our group a lot on how to focus and work effectively as a team. Beyond learning more about creating a pipelined processor and how to handle issues that arise with it (such as hazards), this project served as a chance to improve our SystemVerilog, debugging, and general organizational skills. In the end, we were happy to implement a working pipelined processor that correctly executes all instructions and addresses any hazards. Unfortunately, we did come up a little bit short in terms of our advanced features, but at the very least were able to think in depth about them and what benefits they would provide to our processor.