Introduction

Throughout this semester we built a 5 stage pipelined LC3 processor with various advanced features including branch prediction and prefetching. Learning about the concepts involved and the intricacies and complexities of building such a processor is an important part of a higher education in ECE, as many of these topics form the basis of today's complex microprocessors. In this report we will introduce each of the milestones of our project, as well as the details of each of the advanced design features our processor implements beyond the basic 5 stage pipeline.

Project Overview

Our goal for this project was to create a reliable and competitive pipeline. We viewed the primary bottlenecks for a 5-Stage pipeline as branch mispredictions and memory operations. To increase the performance of our processor we targeted implemented sophisticated branch prediction and prefetching instructions.

The core features of this project are:

1. A 5-stage, pipelined datapath that implemented the LC3b ISA.
2. Split L1 Instruction and Data Cache.
3. A Unified L2 Cache.
4. A 4-way Branch Target Buffer with LRU replacement strategy.
5. A Global 2-Level branch history table.
6. Early resolution of unconditional branches.
7. Hardware prefetching.

To begin, we worked as a group designing the pipeline which could handle all (except RTI) LC3b Instructions. After designing and implementing a basic pipeline that could handle instructions without hazards, we modified our datapath to include hazard detection and forwarding. Once we finalized the design, we divided up the work such that:

- Nathan worked on the caches and hardware prefetching.
- Colin worked on hazard detection and branch prediction.
- Sheng worked on debugging and the Branch Target buffer.

Due to our planning and hard work, we successfully designed a working processor with CPI of ~1.1 on the mp3-final test code.

Design Description

a) Overview - Our project is divided into 3 major checkpoints, with the final stage encompassing all the advanced design options. Each of these checkpoints is described in more detail below.

b) Milestones

   i) Checkpoint 1 - by checkpoint 1 we had implemented a basic 5 stage pipeline supporting all LC3 instructions without any sort of hazard detection. This was tested with code containing NOPs to prevent hazards from resulting in incorrect execution. Most of the bugs at this point were simply due to incorrect decoding or from passing signals through the pipeline incorrectly, and were solved fairly easily. The pipeline is fairly standard at this point, and the full diagram is shown at the end of this document.

   ii) Checkpoint 2 - By checkpoint 2 we had our i cache and d cache both hooked up to physical memory through the arbiter. The arbiter was clocked at this point, although later on we turned it into a fully combinational circuit. We also had forwarding and hazard detection implemented. We added forwarding muxes from the mem and wb stages to the ex stage, and implemented stalling for when there was a dependency on a memory instruction. Branches were essentially predicted always not taken by default, so flushing was performed when a taken branch was detected. At this point we noticed a good number of bugs in testing due to incomplete memory operations causing the caches to end up in incorrect states, corrupting future memory accesses. This was solved by stalling the flush signal until the memory operations were completed, although we later changed the memory operations to correctly cancel on a flush so that significantly fewer cycles were wasted. This checkpoint was able to be tested with all our previous code samples since hazards were detected.

   iii) Checkpoint 3 - As we had gotten ahead in our checkpoint 1 and 2 designs, the only feature left for checkpoint 3 was integrating the L2 cache. This was a fairly simple procedure as it should have been a drop-in replacement for memory. Unfortunately due to the design of the caches, we had a few minor bugs due to the cache raising the response signal even when memory operations had not been requested. However, once those problems were solved the processor appeared to work on all the code samples we had accumulated from previous checkpoints. The only things remaining were our advanced design options.

c) Advanced Design Options

   i) Baseline machine: for performance analysis, we will consider the baseline machine to be our checkpoint 3 machine with a large L2 cache. With an L2 cache size of 16 lines of 256 bits each (so 2KB total size) and no advanced features, the processor runs mp3-final in 0.995ms, giving a CPI of 2.7.

   ii) BTB

A) Design - the branch target buffer is 32 lines 4 way associativity. We use the least significant bit for offset, pc_if [5:1] bits for index and remaining bits for tag. The branch target buffer predicts the result at IF stage. In IF stage, we use the pc_if[5:1] to index into buffer and then compare the tag from pc with tag stored in the buffer. If the tag is the same which means hit in BTB, give the value stored in the buffer to predict address. If not hit, give the pc_if value to predict address. The BTB will be updated in WB stage, after we get exact PC value for both conditional and unconditional branch instructions. A 4 way pseudo LRU buffer is used for branch target buffer. It has two advantage. FIrst, it provided a large buffer which can store as many instructions as we want. The more value stored in the branch target buffer, the more precise prediction will make by BTB. Second, the pseudo LRU uses least bit to index a reasonable place for new value to store. It could save much space and time to find the least recent use pc in branch target buffer. Similarity, the if the pc value get hit in branch target buffer, the new pc value will replace the old one then update the LRU. If not hit in branch target buffer, the new pc value as well as the tag and LRU will find the least recent used one to be replaced.

   In addition, the branch target buffer also support for other branch instructions such as JSR, TRAP and JSSR. It's simple to add them in WB stage, just judging the opcode. For TRAP instructions, there is a slight difference. The memory value in WB stage instead of ALU value in WB stage will be loaded. The BTB will not be updated if the opcode in WB stage is not branch instructions. The branch target buffer only give the predicted address when BTB hit. IF the BTB is not hit or the branch predictor predicts not taken, the pc value should be the original pc value.

B) Testing - We use both factorial code and final test code for BTB testing. It turns out that the BTB could update correctly in WB stage. Also, the BTB will give prediction in IF stage. We find that the branch target buffer combined with branch predictor improve the code execution time a lot. And in many situation, the BTB provides the correct prediction PC value for next stage, so it's no need for flushing all incorrect instructions which wastes much time.

C) Performance Analysis - we analyze the BTB and branch predictor together.

iii) Branch Predictor

A) Design - the predictor was a global 2 level branch history table. We defaulted our global register to be 4 bits wide and our table to contain 32 entries. The table was indexed into by the global history register xor'd with the top bits of PC[5:1]. The global history register shifts by 1 for any branch that gets executes, and the new bit is set to 0 or 1 depending on the direction the branch went. Care had to be taken to ensure that when entries in the history table were written to, that they were indexed into using the same global history value that they had been predicted with. This required sending the global history register value down the pipeline until the branches were resolved (in our processor, this occurs in the wb stage). The counters themselves are 2 bit saturating counters.

In addition, the predictor was split across the if and id stages. There were 2 main reasons for this. First, it allowed for a higher clock frequency. By indexing into the table in the if stage, it could be indexed into independently of the fetch itself, since the result was not needed until the next cycle. The other reason is that by completing the prediction in the id stage, the prediction could know for sure whether the instruction was a branch or any other kind of instruction that would modify the control flow. This allowed the predictor to always predict taken for instructions such as jsr, trap, etc, without the btb having to keep track of them. Although it predicted taken on some of these other instructions, care was taken to only update the history table for branch type instructions, since the other instructions should not be taken into account for the prediction mechanism itself.

One interesting small optimization we also did was add a valid_inst bit to each instruction in the pipeline. This allowed detecting whether an instruction opcode was all 0s (ie branch opcode) because of a flush or because it was actually a branch instruction. This prevents flushes from causing incorrect predictions due to the counters being incremented/decremented incorrectly.

B) Testing - the testing procedure used was fairly simple. The original code I had written for factorial at the beginning of the semester had a good number of branches in it, so I just ran the factorial code and manually verified the history table and flushing was occurring correctly. Once things seemed to be working, the mp3-final.asm code was run, and it sped up significantly, leading me to believe it was working. When we implemented performance counters for branch prediction rates, they indicated over 90% accuracy on the mp3-final.asm code.

C) Performance Analysis - once branch prediction and the btb are added (no prefetching), the mp3-final code runs in 0.639ms, an improvement of 1.6 times, and reduces CPI to 1.7.

iv) Prefetching Unit

A) Design - The prefetching buffer was placed between the L2 Cache and physical memory and acted similar to a read-only cache with 1-way and 8 lines. When the L2 Cache preformed reads to memory, the prefetch buffer would intercept these requests and check if the requested data was present in the buffer. The reason we placed the prefetch buffer in between the L2 Cache and physical memory was to ensure that the L2 cache would never evict data due to replacement from the prefetch.

We designed the buffer to prefetch the next sequential line after an I-Cache miss. This way, we keep the prefetch buffer small while reliably increasing the performance of the processor since the sequential line of what gets loaded into the I-cache is likely to be used.

B) Testing - the testing procedure was to verify that the instruction we want prefetched gets loaded into the prefetch buffer (which goes to the L2 and I Cache). After that we had to ensure that writes to physical memory would invalidate any matching address in the prefetch buffer. Since the reorder buffer acted very similar to a cache, it was fairly simple to implement and verify an error free design.

C) Performance Analysis - .With just the prefetch buffer the mp3-final code runs in 0.901ms which gives a CPI of 2.43.

v) Final performance analysis - until now, each of the performance analyses have only taken into account each individual feature. However, considering the final product in which all features are taken into account, the running time of mp3-final is 0.542ms, a speedup of 1.8 times. The CPI for this is 1.5. If we increase the L2 cache size, the running time can be pushed even lower down to ~0.4ms, which would be a speedup of ~2.5 times and gives a CPI of ~1.1.

Additional Observations

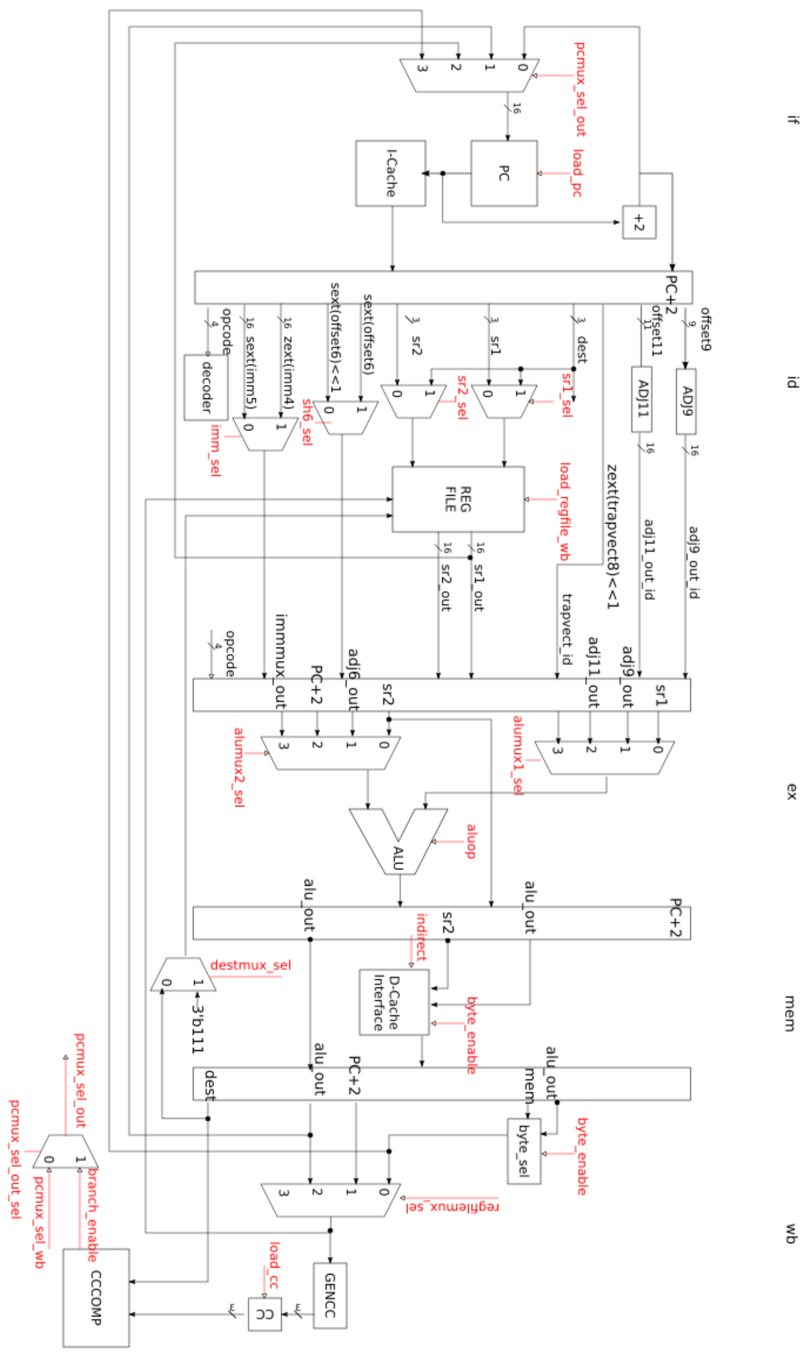
Some of our features could certainly be improved. For instance, with JSRR we could try and grab the values directly from the registers (we currently do this only for R7, since it is the most commonly used register for this instruction). It might also be possible to never branch to an incorrect address by calculating the offset early enough so that the btb would not even be necessary for branch instructions (at which point trap would probably be the most useful instruction for btb to pay attention to). For BTB, when we update the BTB, because we use the pseudo way, in some case, the LRU could provide two least used way, then two value will be replaced by new pc value. For prefetching, we could experiment with data prefetching in addition to instruction prefetching.

Conclusion

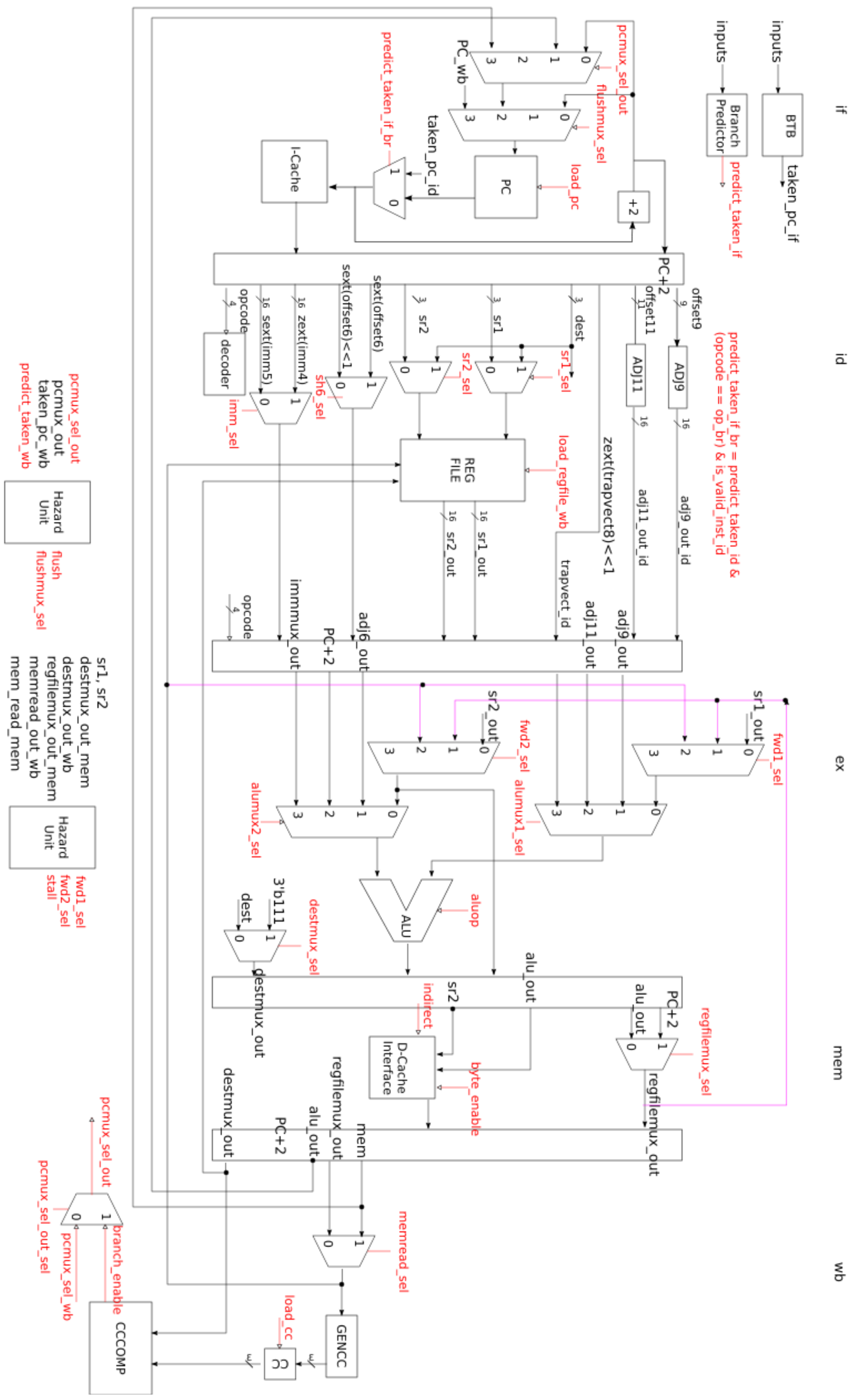
In conclusion, this project helped us understand and appreciate the complexity involved in processor design. We were able to complete our 5-Stage pipeline, several advanced features and achieve a notable CPI.

Pipeline Diagrams

Here are a couple pipeline diagrams of interest



Basic 5 stage pipeline



The final diagram including hazard detection, flushing, and branch prediction