> ECE 411 MP3 Report

> fmax_420 - Spring 2017

Tadas Aleksonis

Tanishq Dubey

Rohan Mathur

Introduction

In this project, we aimed to convert a classic multi-cycle processor into a processor focused on performance, adding modern features such as pipelining, split L1 and a unified L2 caches, and many more features. In deciding these options, we took into consideration the impact the advanced feature would have on our performance, along with the complexity of the feature.

Our five-stage pipelined processor ended up supporting the entire LC3b ISA, supporting basic instructions such as ADD, AND, NOT, to memory instructions such as LDR, LDB, STR, STB, and more complicated instructions such as BR, JMP, TRAP, LDI, STI, and more. Aside from the basic instructions, it features split L1 caches (one instruction cache, and one data cache) with 128-bit line lengths and 2 set associativity, along with a cache arbiter and a unified L2 cache, with similar specs as our L1. It supports full data hazard detection/forwarding, which ensures the fastest execution time possible with minimal stalling. We added performance counters to track aspects such as cache hits/misses, stalls, and branch mispredicts (when our static branch prediction predicted wrong), which are accessible via memory-mapped I/O. Finally, for our required features, we added an eviction write buffer, which attempts to optimize writebacks to memory from our cache.

For our advanced features, we chose to implement two commonly seen features; memory stage leapfrogging, and basic hardware prefetching. We chose memory stage leapfrogging as one advanced feature to implement, as throughout our development process, we noticed that the main source of our stalls were memory accesses. If an instruction does not depend on the memory stage or any of the operands, which was common, it should be allowed to continue execution, which was our goal. To further limit the effect that physical memory has on the runtime of our processor, we decided to implement basic hardware prefetching. This fetches the next cache line whenever one cache line is accessed, which is faster for operations such as sequential accesses. With these two features, we aimed to address one of the biggest slowdowns that we saw with our processor throughout the development process.

Project Overview

The final project had a high complexity, which stemmed from the fact that multiple different features had to work in harmony with one another. In order to achieve this fine-tuning of the processor, it was imperative that we had some rigid system in how we approached the project as both a team and as developers.

At the beginning of each checkpoint, each teammate would opt-in to doing a certain task. This libertarian approach made the administrative overhead lower, and gave us more time in debugging. However, the drawback to the single-developer approach was often the merging of the different tasks. While each different module worked on their own, the merged logic would often have blowback. More time would then have to be spent debugging.

Management in the project was straightforward. We were all friends, with similar schedules, extracurricular obligations, and goals. Synchronization and having the entire team on the same page was a non-issue. Aside from the minor blips in scheduling or finishing a task (such as completing forwarding on time), there were no complications that arose in the team managing itself.

Given the opportunity to redo the project, our management would have improved had we started simultaneously started working on all checkpoints, in order to make the addition of features such as forwarding and leapfrogging a smoother process. Most time spent on the processor was patching fixes because of ad-hoc implementations that later caused issues in our logic.

Our work split was as followed:

Checkpoint 1: IF/MEM stages - Rohan EX stage - Tadas ID/WB stage - Tanishq Checkpoint 2: Full LC-3b ISA - Tanishq and Tadas L1/cache arbiter/physical memory - Rohan Checkpoint 3: Data forwarding - Tanishq and Tadas L2 cache - Rohan	Checkpoint 4: Performance counters - Rohan EWB - Rohan Branch prediction/Flushing - Tadas Debugging - Tanishq and Tadas Checkpoint 5: Instruction Cache Prefetching - Rohan and Tadas Memory Stage Leapfrogging - Tanishq

Design description

Overview

Over the next few sections, we discuss our processor milestones, along with their designs and testing strategies. Our advanced design features are then detailed after our milestones.

Milestones

Checkpoint 1

We believe we completed all of checkpoint 1 functionality, implementing a basic pipelined processor, supporting the instructions ADD, AND, NOT, LDR, STR, and BR. We did not handle any control hazards or data hazards in this checkpoint, as specified in the checkpoint documentation.

We implemented this pipeline hierarchy by writing separate modules for each of the pipeline stages, and also writing a single “buffer” module to use between each of the stages. This buffer has a unified input/output scheme to make wiring up easier for us. Input/outputs are simply filled into the next buffer as they are generated. Our write-back stage is really just a few wires going back to the ID stage.

A paper design of our pipelined processor can be viewed in Appendix A.

Testing

We wrote test programs with several NOPs in between to individually test the instructions implemented this checkpoint. We kept building on this one program by adding new instructions as we tested them, so that we could not only verify existing functionality at every stage of the development process, but also check functionality of new instructions as we were adding/checking support of them.

Checkpoint 2

By this checkpoint, we had implemented full support for all of the LC-3b ISA instructions, including the more challenging instructions such as STI/LDI, and more. We accomplished this by adding mem stage stalling individually, which allows instructions like STI and LDI to complete correctly, doing two memory accesses per instruction. We also had to add support for instructions such as TRAP, JSR/JSRR, and others, which were pretty similar to BR/other instructions that we had implemented previously.

We also added both of our L1 caches to our processor (icache and dcache), along with a basic cache arbiter that allowed both caches to have unified access to physical memory, instead of dual port magic memory like in Checkpoint 1.

A basic paper design for our L1/cache arbiter/L2 layout can be seen in Appendix B.

Testing

To test our ISA, we continuously wrote more and more test code that utilized each of the instructions one by one. Since the only provided test code was a large test, comprising of several interleaved operations that were both old and new, it made testing with it very confusing. Because of this, we opted to write our own test code testing each individual instruction independently, aside from some boilerplate code, to enable easier debugging, and verify that our processor was working as it should.

In addition to writing our own code to test each individual new instruction, we also simulated modules independently to verify they were outputting the correct data (such as simulating just the extend_trapvector8 module to make sure it was properly zero extending and left shifting once, as the TRAP instruction specifies). Being able to unit test these smaller modules before combining them all to create a bigger picture was essential in being successful with as many instructions as it is.

Test code that we wrote to test our TRAP instruction is in Appendix C.

To test our L1/cache arbiter, we ran the MP2 final test code, which tests aspects of caching such as writebacks, evictions, and more, as we found the test code to be pretty comprehensive. We also made sure the cache functioned correctly on our MP2 design before adding it to our MP3.

Checkpoint 3

We believe that we have completed all of checkpoint 3 functionality, with full data forwarding implemented and tested, along with a unified L2 cache in our cache hierarchy. We have written additional test code to check each of our forwarding paths, and we believe that our pipelined processor is correctly operating in each of the cases and can handle all needs of this. Our multi-cycle L2 cache was made multicycle as our arbiter is clocked with registers and a state machine, allowing our critical path to not extend all the way to L2 data lookups.

A basic paper design for our L1/cache arbiter/L2 layout can be seen in Appendix B.

Testing

Testing our L2 was straightforward, we used a variant of the MP2 final test code. We modified this test code to evict from L2 as well, to test that not only do L1 eviction/write backs function correctly, but L2 does as well. We ensured that our data lines in our L2 were getting filled with data, and that successive writebacks from L2/loads would load in the correctly written back value.

To test our data forwarding, we wrote our own test code, as we noticed that the provided test code did not test a lot of the forwarding cases we implemented. We tested each forwarding path independently of all the data dependencies we could think of, between different types of instructions (types meaning which ones use SR1 vs SR2 vs BaseR, etc).

Test code that we used to test our data forwarding and hazard detection can be found in Appendix D.

Checkpoint 4

The functionality for checkpoint 4 is mostly completed. Our performance counters work to count our stalls, branches, cache hits/misses, and other aspects of our system. Additionally, these counters can be reset by writing to the specific memory-mapped I/O address corresponding to each counter.

Our static branch prediction is static not-taken, which works in tandem with branch flushing to successfully execute all branch instructions.This feature is completed as well.

Our eviction write buffer, however, does not seem to be fully working yet. We seem to have an issue with our read hits into the EWB, when the EWB contains more up to date data than physical memory (since our EWB was placed between L2 and physical memory). Aside from this, our EWB functions as intended (it loads in our written back data, and eventually writes it to pmem, blocking other memory operations when it is writing, etc).

Our designs for our performance counters and our EWB can be found in Appendix E.

Testing

We wrote test code to test all aspects of our processor, as for this checkpoint, test code is not provided. For checking the performance counters, we first ran a lengthy program, and then loaded R1 and R2 continuously with the various counters, and then zeroed them out, and then fetched the counter values again. This allowed us to check functionality of both the counters and the reset signals for the memory mapped I/O.

We wrote test code to check our branch flushing and static branch prediction as well, by putting instructions that should not be allowed to the MEM stage right after taken branches, to see if those instructions were executed. They were not, and were instead flushed as expected.

We finally tested our eviction write buffer by running our modified MP2 code that tested our L2. This effectively checked to see if our write backs were eventually occurring, and if our reads from eviction write buffer were functioning correctly (through this code, we found our that it was not functioning correctly).

Test code to test our performance counters and our branch flushing can be found in Appendix F.

Checkpoint 5

We completed our advanced design features as detailed below.

Advanced Design Options

Memory Stage Leapfrogging

Memory Stage Leapfrogging (MSL) is designed to take advantage of the fact that some instructions do not use the memory stage in any way. This means that they can completely bypass the stage without any real effect on the functionality of the processor or the code it is running. However, to make this leapfrogging efficient, it is best to only leapfrog if there is something in the memory stage that would be causing the instructions in previous pipeline stages to be halted, specifically, when there is a stall in the memory stage. During this time, operations in the EX stage that do not depend on the operation in the MEM stage are “leapfrogged” directly to the writeback stage, This means that non-dependent operations are occurring while the memory stall happens causing an increase in instructions per second.

It should be noted that all instructions are forwarded except for those that depend on the memory stage registers, require the memory stage for memory access, or do a jump/PC change, as this is resolved in the mem stage. In addition it should be noted that the CC register is set by leapfrogged instructions and not the instruction that was stalled in the MEM stage, as the leapfrogged instructions are further ahead in program execution and have precedence in setting the CC.

Testing of memory stage leapfrogging was done in two stages. The first stage was to write simple test code that explicitly tests leapfrogging by doing a memory operation that is guaranteed to stall (initial miss) and then doing EX operations that do not depend on memory. Dependent operations are then tested along with a jump. Finally, branching and CC setting is testing by explicitly setting the CC register with a simple operations and attempting a branch which should not pass. This test code can be found in appendix G, Finally the main code was run to ensure backwards functionality.

Performance significantly increased with the addition of leapfrogging as many operations do not depend on the loading of data and can operate independently. In fact, running the final test code without leapfrogging took 2001342ns while with leapfrogging a time of 1857746ns achieved, leading to a 7.17% decrease in runtime.

Hardware Prefetching (basic)

Hardware prefetching allows for the cache to grab more than just the requested line of data in preparation for sequential accesses in memory. This, in theory, would allow for fewer cache misses as usually data is accessed in a sequential manner, especially within loops and similar constructs. To implement this, modifications were made directly to the cache control logic and not the hardware, as no extended memory or data lines are required. Changes made to the control logic are as follows. The main difference is that instead of simply fetching one line of data, two continuous lines of data are fetched in order. This means that for every fetch, a writeback is analyzed for each line, the writeback is conducted, and then the two lines of data are fetched. Noting that this could be done completely with control logic was essential to the design of the feature as it means that this design could be extended to fetch *n* lines if needed, however, though testing, discussed below, it was decided that 2 lines was enough to improve performance an ample amount.

It should be noted that in the end, we implemented cache prefetching only in the I-Cache and not the D-Cache, and for good reason. It was observed during testing that hardware prefetching in the D-Cache was, for lack of a better term, volatile in the sense that there was not a good method for predicting if prefetched data was useful or not. This was the exact opposite case in the I-Cache as instructions are for the most part parsed sequentially so prefetching meant that fewer cache misses were performed.

To test the hardware prefetching, specific test code was not written as the basic functionality of the cache was not modified. Instead, attempts to maintain high backwards compatibility were followed as correctness is more important than speed. Because of this MP2 final test code was used to ensure functionality and MP3 final test code was used to check for performance.

As previously stated, implementing prefetching in both I and D caches led to no significant performance increases, if any, indeed, in some cases, the runtime was slower! However, once we implemented a I-Cache only prefetching model performance was increased significantly, especially with the combination with MSL, we had a final runtime of 1212165ns, compared to 2001342ns, a 39.4% increase in performance! It should definitely be noted again that this performance increase comes from removing prefetching in the D-Cache AND leaving prefetching in the I-Cache. In fact, tests were done to have neither cache prefetch and results similar to MSL were achieved, and with double prefetching, an average of 1600000ns was achieved for the runtime, but again, that was not our fastest time.

Conclusion

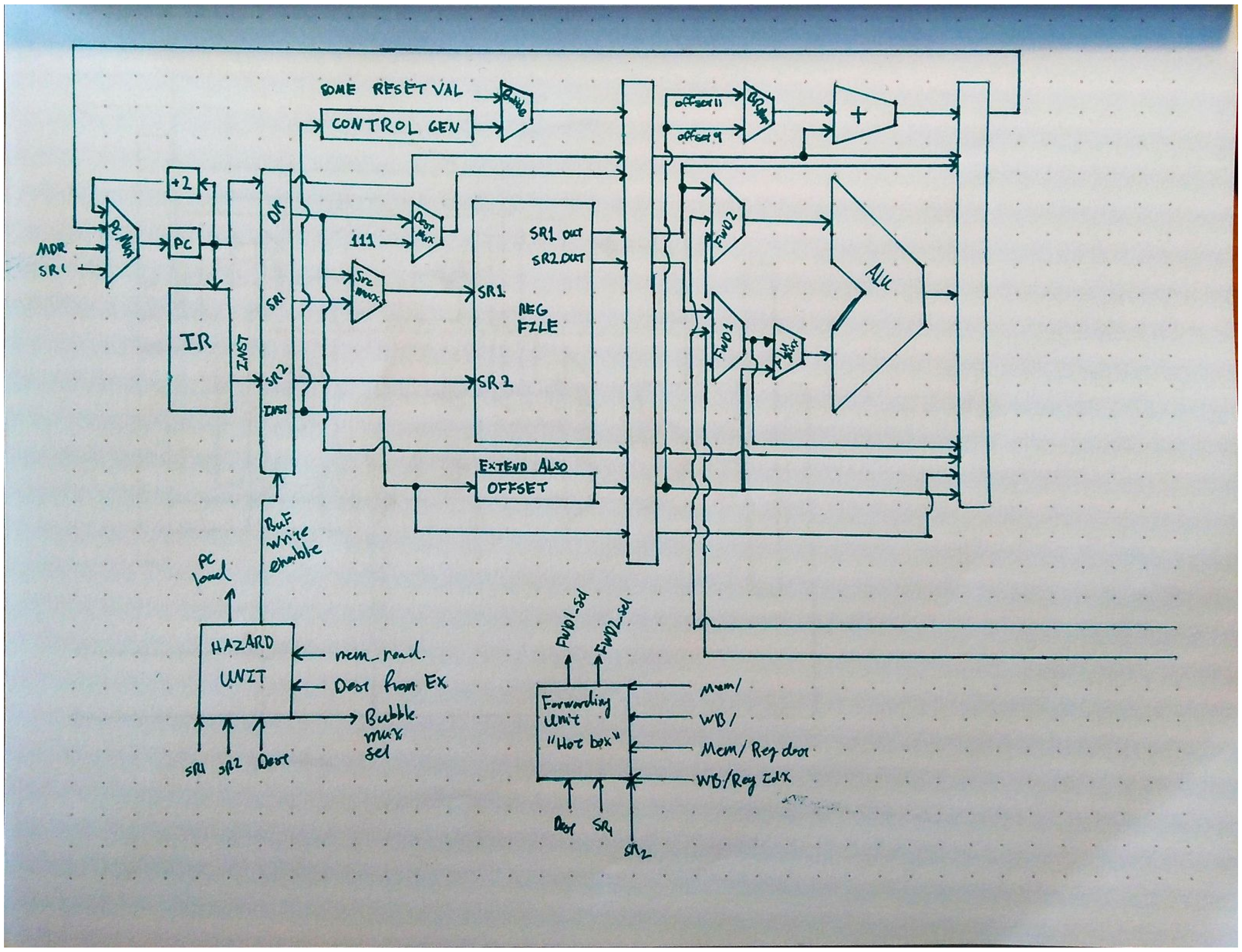
In conclusion, we sought to design a processor that would meet the requirements of the MP and meet our own standards. The processor successfully implemented pipelining, forwarding, leapfrogging, basic prefetching caching, among other things. The project as a whole was different from any other sort of project any of us have worked on; in comparison to ECE391 (the most comparable project-based class), this class had an entirely different paradigm in designing, implementing, and debugging.

However, given the limited time in the project, we were not able to implement more interesting ideas in addition to the basic processor (including the advanced designs). We speculated the benefits of replacing RTI with an instruction that would “flip” the ISA, where the same opcodes would be mapped to a set of new instructions; effectively, RTI could be used instead to sign extend the opcodes from 4 bits to 5 bits. Some of our own goals were successfully met. For instance, we succeeded in modularizing most functions, such as making each pipeline an instance of the same module.

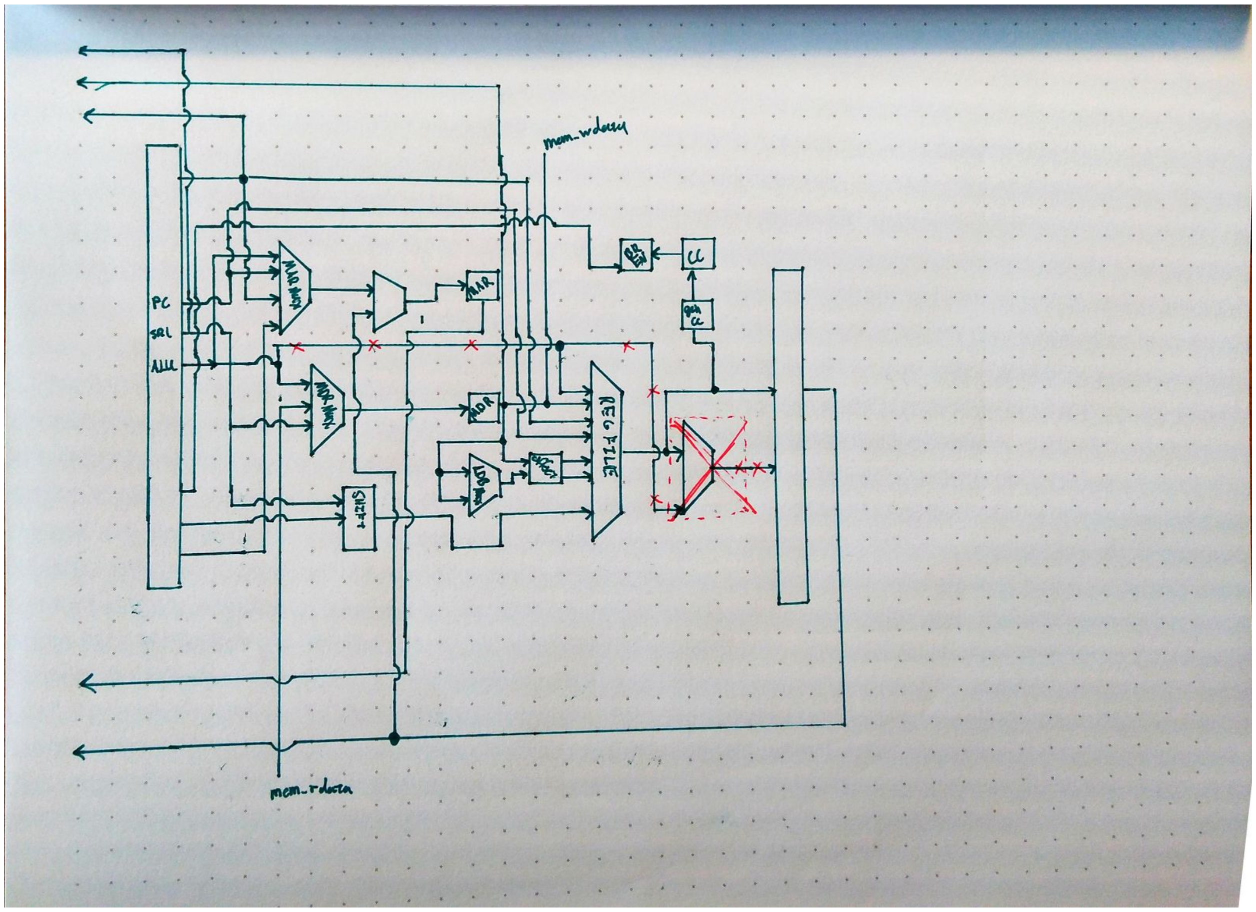
In the final analysis of our processor’s performance, some key numbers to note are the final running times of mp3-final.asm and the fmax. Unfortunately, we did not achieve an fmax of 420 MHz. Instead, it was an fmax of 123.73 MHz. Our final run time for mp3-final.asm was 1212165ns.

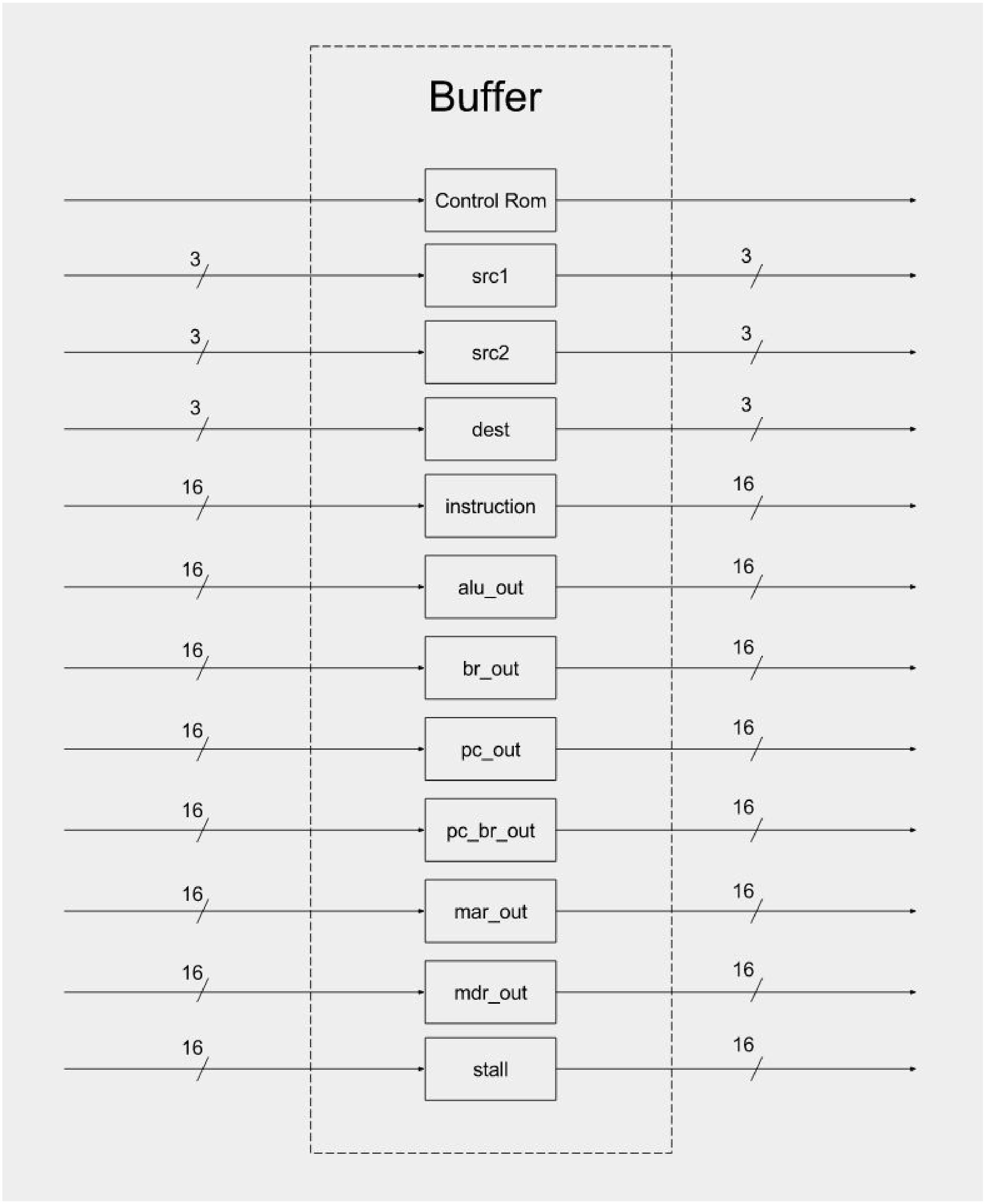
We did not expect the project to be as complex as it was, but in coming to a conclusion with the semester, there is a newfound respect for processor design and verification. We came into the class skilled at systems programming/debugging, but had to completely change our scope in order to implement this processor.

Appendix A

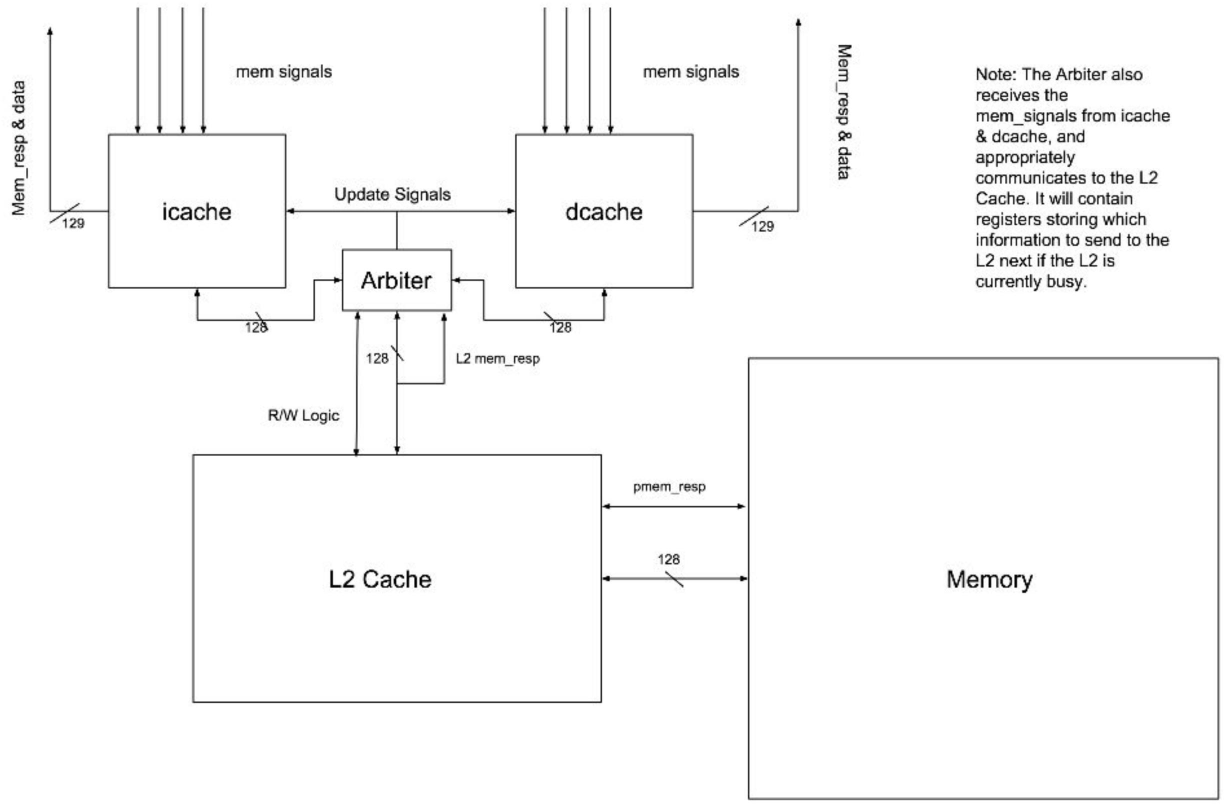


SOME RESET VAL
CONTROL GEN
Bubble
+2
PC Mux
PC
MDR
SR1
IR
INST
DR
SR1
SR2
INST
111
Dest Mux
Sr2 mux
SR1 OUT
SR2 OUT
SR1
REG FILE
SR2
EXTEND ALSO
OFFSET
offset11
offset9
+
FWD2
FWD1
ALU
Buf write enable
PC load
HAZARD UNIT
mem_read
Dest from EX
Bubble mux sel
SR1
SR2
Dest
FWD1_sel
FWD2_sel
Forwarding Unit "Hot box"
Mem/
WB/
Mem/Reg dest
WB/Reg idx
Dest
SR1
SR2


mem_wdata
PC
SR1
ALU
MAR Mux
MAR
MDR Mux
MDR
LDB Mux
SHIFT
REG FILE
BR EN
CC
Gen CC
mem_rdata


Buffer
Control Rom
src1
src2
dest
instruction
alu_out
br_out
pc_out
pc_br_out
mar_out
mdr_out
stall
3
3
3
3
3
3
16
16
16
16
16
16
16
16
16
16
16
16
16
16

Appendix B



Note: The Arbiter also receives the mem_signals from icache & dcache, and appropriately communicates to the L2 Cache. It will contain registers storing which information to send to the L2 next if the L2 is currently busy.

Appendix C

TRAP Test Code:

```
ORIGIN 4x0000
SEGMENT  CodeSegment:
    ADD R1, R0, R0      ; clear all register values
    ADD R2, R0, R0
    ADD R3, R0, R0
    ADD R4, R0, R0
    ADD R5, R0, R0
    ADD R6, R0, R0
    ADD R7, R0, R0
    TRAP SUBROUTINE
    ADD R1, R1, 5
SUBROUTINE:
    DATA2 SUBROUTINE2
SUBROUTINE2:
    LDR R5, R0, 5
    RET
FINISH: BRnzp FINISH
```

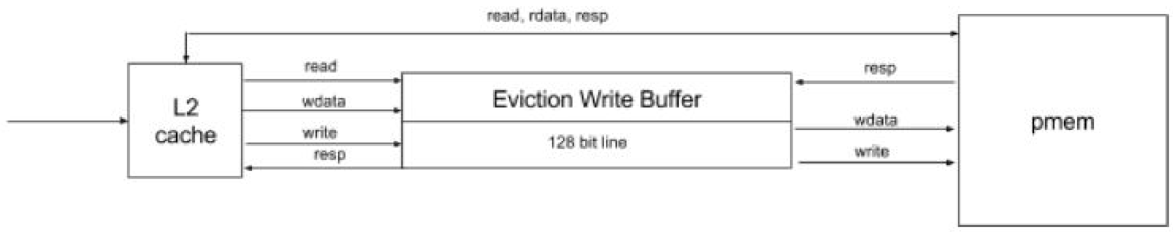
Appendix D

Data forwarding test code:

```
ORIGIN 0
SEGMENT  CodeSegment:
       ADD R3, R3, 5
       ADD R2, R3, -2
       AND R1, R2, R3
       ADD R6, R1, 9
       NOP
       AND R3, R3, 0
       NOP
       ADD R2, R3, 1
    LEA R0, DATA
    LDR R2, R0, LVAL1
    ADD R2, R2, 4
       BRnzp HALT
HALT:
       NOP
       NOP
       NOP
       NOP
       NOP
       BRnzp HALT
       NOP
       NOP
       NOP
       NOP
       NOP
SEGMENT
DATA:
LVAL1:  DATA2 4x0020
LVAL2:  DATA2 4x00D5
GOOD:   DATA2 4x600D
```

Appendix E

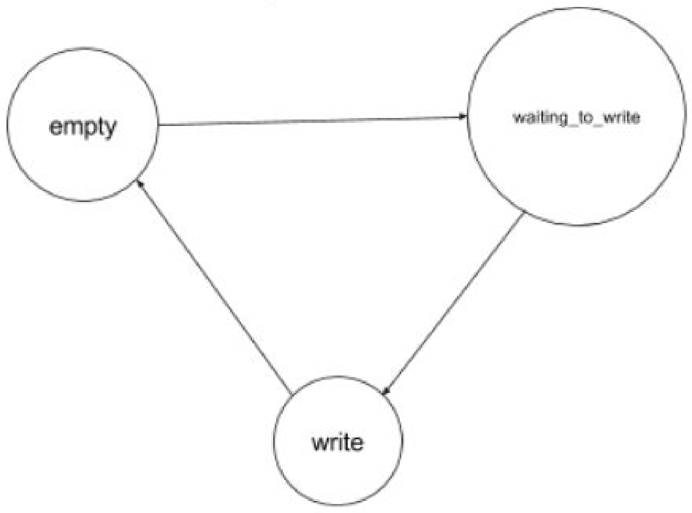
Eviction Write Buffer Design



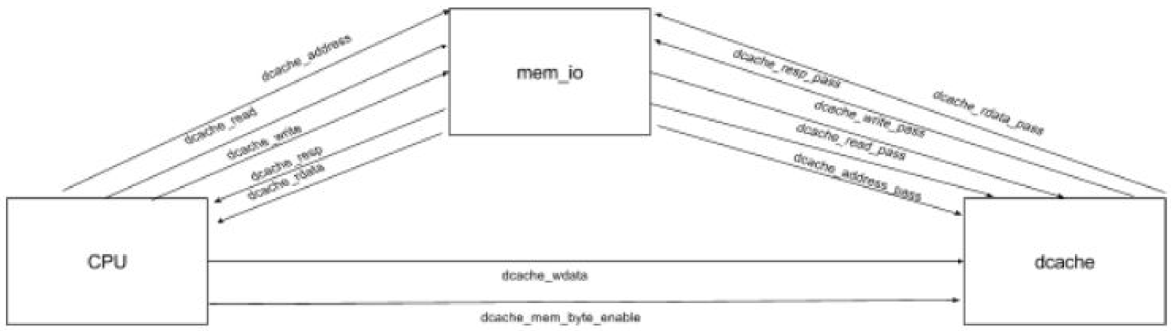
Our eviction write buffer takes care of all the writes from L2 as long as the line is free in the buffer. At a later time (as long as read/write are not high), it sends a write signal from itself to pmem.

The one weird case will be when we are waiting to write, but then another L2 write comes in, which can't occur as there is only one entry in the eviction write buffer. In this case, we will prioritize the write to pmem from the eviction write buffer, and then come back to the empty state where we will fill the buffer with the new line L2 wants to write.

It will have a state diagram such as below:



Performance Counters Design:



MEM_IO is going to essentially pass through the read/write/resp/rdata/address signals to dcache all the time combinationally, unless the address is found to be one of the memory mapped I/O addresses (near 0xFFFF)

If one of these addresses is read/written to, then mem_io will not pass through the read/write signals. On a read, it will set rdata to the counter value assigned to that specific address, and then send a resp back. On a write, it will set a counter reset wire high, which resets the counter in the specific stage it comes from.

Branch instruction/Branch taken counters are going to be in the MEM Stage, so that both counters can be kept in sync of each other.

IF Stalling counter will be placed into IR in the IF stage of the pipeline, where the stall originates.

MEM Stalling counter will be placed into the MEM stage of the pipeline, where that stalling originates.

Cache hits/misses will be placed into each of the cache's (icache, dcache, L2 cache) control logic.

All of these counters will also have a reset signals originating in mem_io and going to the other parts of the hierarchy. When the counters detect this signal is high, they will reset their value to 0.

Appendix F

Test code written to test basic branch flushes:

```
ORIGIN 4x0000

ADD R1, R1, 3
BRp TAKEN
NOT R6, R6
NOT R7, R7
NOT R5, R5
NOT R4, R4
NOT R3, R3
NOT R2, R2
TAKEN:
ADD R4, R4, 1
ADD R3, R3, 1
BRnzp TAKEN
NOT R1, R1
NOT R2, R2

SEGMENT CodeSegment:

ONE:    DATA2 4x0001
TWO:    DATA2 4x0002
NEGTWO: DATA2 4xFFFE
TEMP1:  DATA2 4x0001
GOOD:   DATA2 4x600D
BADD:   DATA2 4xBADD
```

Test code written to test performance counters:

```
SEGMENT  CodeSegment:

   LDR  R1, R0, NEGTWO  ; R1 <= -2
   LDR  R2, R0, TWO     ; R2 <= 2
   LDR  R4, R0, ONE     ; R4 <= 1
   NOP
   NOP
   NOP
   NOP
   NOP
   NOP
   NOP
   BRnzp LOOP
   NOP
   NOP
   NOP
   NOP
   NOP
   NOP
   NOP


ONE:    DATA2 4x0001
TWO:    DATA2 4x0002
NEGTWO: DATA2 4xFFFE
TEMP1:  DATA2 4x0001
GOOD:   DATA2 4x600D
BADD:   DATA2 4xBADD
BR_COUNT:               DATA2 4xFFFE
BR_MISPREDICT_COUNT:    DATA2 4xFFFF

LOOP:
   ADD R3, R1, R2       ; R3 <= R1 + R2
   AND R5, R1, R4       ; R5 <= R1 AND R4
   NOT R6, R1           ; R6 <= NOT R1
   NOP
   NOP
   NOP
   NOP
   NOP
   NOP
   NOP
   STR R6, R0, TEMP1    ; M[TEMP1] <= R6
   LDR R7, R0, TEMP1    ; R7 <= M[TEMP1]
   ADD R1, R1, R4       ; R1 <= R1+1
   NOP
   NOP
   NOP
   NOP
```

```
    NOP
    NOP
    NOP
    BRp DONE
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    BRnzp LOOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP

HALT:
    LDR  R1, R0, BADD
    BRnzp HALT
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP

DONE:
    LDR  R1, R0, GOOD

    LDR R1, R0, BR_COUNT
    LDR R2, R0, BR_MISPREDICT_COUNT

    LDR R1, R1, 0
    LDR R2, R2, 0

    BRnzp DONE
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
```

Appendix G

```
ORIGIN 0
SEGMENT CodeSegment1:
START:
    LEA R0, DataSegment
    LDR R5, R0, FOED
    ADD R1, R1, 1
    ADD R2, R2, 2
    ADD R3, R3, 3
    NOP
    NOP
    NOP
    ADD R6, R6, 6
    LSHF R6,R6,2
    ADD R2, R2, 2
    ADD R3, R3, 3
    NOP
    NOP
    NOP
    ADD R1, R1, 1
    ADD R6, R6, 6
    ADD R5, R5, 1
    LDB R4, R0, ZERO
    ADD R5, R1, 4
    BRz HALT
    LSHF R1, R1, 1
    NOP
    NOP
    NOP
HALT:
        BRnzp HALT

SEGMENT DataSegment:

ZERO:   DATA2 4x0000
FOED:   DATA2 4xF0ED
BOMB:   DATA2 4xB006
```