AmdaZulo: A Superscalar LC-3b Processor

Steve Hanna, Tom Hughes, Mark Murphy

May 3, 2006

Abstract

The goal of AmdaZulo was to build the fastest LC3-b implementation in existence by implementing a superscalar design supporting out-of-order execution, register renaming, out-of-order retirement, speculation, and four-wide execution.

Contents

Chapter 1

Introduction

The LC-3b is a 16-bit byte-addressable ISA that was developed by Professor Sanjay Patel at the University of Illinois at Urbana-Champaign and Professor Yale Patt at the University of Texas-Austin for instructional purposes. Although the LC-3b is a rather simple ISA, it has enough functionality to support moderately complex programs written in LC-3b assembly. Furthermore, fully functional C programs can now also be compiled for the LC-3b with the aid of Mark Murphy's ucc compiler. The LC-3b supports 16 opcodes, which include ADD, AND, BR (branch), JMP (jump), JSR (jump subroutine), JSRR (jump subroutine register), LDB (load byte), LDI (load indirect), LDR (load register), STB (store byte), STI (store indirect), STR (store register), LEA (load effective address), NOT, RET (return), RTI (return interrupt), SHF (shift), and TRAP.

1.1 Motivation

The implementation we developed is entitled "AmdaZulo". The motivation behind the name is a combination of the names Amdahl and Tomasulo. Amdahl is a reference to the well-known Amdahl's Law, which states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used (Hennessey). In essence, Amdahl's Law emphasizes that consideration must be given to how much the new enhancements will really speed up the overall execution time. The "Zulo" part of the name refers to Tomasulo's algorithm, which is an algorithm to exploit the existence of multiple execution units. Specifically, it resolves WAR (write after read) and WAW (write after write) hazards by renaming values, while preserving RAW (read after write) dependencies.

Chapter 2

Project

The goal of AmdaZulo was to develop a working implementation of a superscalar LC-3b processor in structural VHDL. The tools used were *FPGA Advantage for HDL Design* by Mentor Graphics for the CAD design in conjunction with ModelSim for simulation. The development machines were Sun machines with AMD Opterons running RedHat Linux. Although *FPGA Advantage* supports standard VHDL, the project constraints required it to be done structurally with logic gates and appropriate delays.

2.1 Outline and Objectives

The main objective during the design of AmdaZulo was pure speed and a bit of bravado. Given the limited time for the development of the project, our group focused on implementing a working design rather than worrying about cost of the components. Practically, the design would be difficult, if not impossible to build due to its enormous size; however, size was not one of our objectives.

During the beginning of our design phase, we realized that our design was ambitious for the time we had to implement it, so we focused on implementing ideas that would have the most impact on performance, *i.e.*, we attempted to follow Amdahl's Law as much as possible. The main points that we thought would best improve performance included:

1. A larger than architectural register file to support register renaming. The LC-3b has 8 registers, R0 through R7, but we chose to build our processor with 32 physical registers that renamed with a Register Alias Table (RAT). Our implementation is inspired by the MIPS R10000 and NetBurst designs.

2. Use of Tomasulo's algorithm to support out of order execution and multiple issue.

3. Implementing four pipelines, two of which perform ALU operations and two which perform loads and stores.

4. Fetching four instructions at a time to support four pipelines.

5. Large, fast, fully associative caches, including an L1 instruction cache, an L1 data cache, and an L2 data cache.

6. Speculative execution across control instructions (*i.e.*, branches and indirect jumps) with efficient recovery on misprediction.

7. Excellent branch prediction in order to make speculative execution worthwhile.

2.2 Design Trade-offs and Resolution

A superscalar design is significantly more complicated than a five stage pipeline, so we were immediately faced with design decisions due to the limited amount of time and resources we had available to implement our design. As a result, our group read many papers on computer architecture including *An Efficient Algorithm for Exploiting Multiple Arithmetic Units* (Tomasulo 1967), *HPS, A New Microarchitecture* (Patt, Hwu, Shebanow 1985), *The Microarchitecture of Superscalar Processors* (Smith 1995), *Tuning the Pentium Pro Microarchitecture* (Papworth 1996), *The MIPS R10000 Superscalar Microprocessor* (Yeager 1996), *The Microarchitecture of the Pentium 4 Processor* (Hinton et al. 2001), in order to get an idea of the strategies others had used in the past. We also referred to *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson quite a bit due to the excellent data provided that compares various processor improvements such as branch prediction, cache associativity, *etc.*

From the beginning of our design, we realized that a lot of the components in our design really depended on each other in order to get full potential out of a superscalar design. For example, without good branch prediction, our ability to speculate across branches would be practically useless because nothing useful would actually be done during that time. Likewise, without fast caches, it would be difficult to be able to continually fetch four instructions at a time and keep all of our pipelines busy. Again, Amdahl's Law came into play, and we tried to add improvements that would do the most to speed up our processor.

2.2.1 Pipelines and Reservation Stations

During our initial design phase, we wanted to have four pipelines, each of which could perform any operation, meaning ALU or memory operations. The idea was to have a set of global reservation stations that would feed each of the four pipelines. However, after drawing out designs and thinking about complexity, we realized that the amount of hardware necessary to support that would be overwhelming. In exchange, we decided to dedicate two pipelines to ALU operations and the other two pipelines to memory operations. After the DECODE stage, we send ALU operations to the ALU reservation stations, which supply two of the pipelines and memory operations to the memory reservation stations (memory queue). Once in the ALU or memory reservation stations, the instructions can go down either of the two pipelines. Although there is a potential performance gain to be had if we had a global set of reservation stations that could send any instruction down any pipeline, we decided the additional complexity was not worth the performance gain.

2.2.2 Out-of-order Memory Instructions

Another design consideration was how to handle memory operations out of order. Although it is relatively straightforward to perform ALU operations out of order using Tomasulo's algorithm, memory operations are a bit harder. For example, consider the instructions:

```
A: STR R5, R0, 0
B: LDR R6, R0, 0
```

Instruction A is a store instruction that will write the data in R5 to the address stored in register R0. Instruction B is a load instruction that will load the value at the address stored in register R0 into register R6. After both instructions are executed, the final result should cause the value in R5 moved into R6. However, notice what would happen if instruction B executed before instruction A. R6 would get loaded with a value and then R5 would get stored to memory; this is definitely not correct.

Our initial solution to the problem of memory operations was to simply do them in order. We built a FIFO queue of reservation stations and attempted to execute two of the memory operations at the head of the queue in each cycle. However, after building the queue, we decided that we could squeeze more performance out of the memory operations by executing load instructions out of order as long as they were serialized with respect to stores. For example, consider the following instructions:

```
A: STR R5, R0, 0
B: LDR R6, R0, 0
C: LDR R3, R1, 1
D: LDR R4, R2, 3
E: STR R6, R5, 0
```

Note that as long as the first instruction in the sequence to execute is instruction A and the last instruction in the sequence to execute is instruction E, instructions B through D can execute in any order assuming that all their operands are ready (*i.e.*, assuming R0, R1, and R2 are ready). The result is that if a store instruction is at the head of the queue (*i.e.*, it is the next instruction in program order), it will block all memory operations until it executes. However, if we have a series of load instructions at the head of the queue, they may execute in any order, assuming their operands are ready. Although this is not complete out of order execution of memory operations, we decided that it would be enough of a performance boost. We wanted to make sure that the memory operations were not the limiting factor in the performance of our processor.

2.2.3 Reference Counting and Register Renaming

The LC-3b architecture specifies only eight registers. We decided to dynamically rename these and map them onto a 32-entry physical register file. The benefit of this approach is two-fold. The register renaming avoids WAW and WAR data hazards. Consider the following:

```
A: LDR R0, R6, 0
B: ADD R3, R0, 1
C: ADD R1, R3, R1
D: LDR R0, R2, 3
E: ADD R3, R0, 0
F: ADD R1, R3, R1
```

If instruction A results in a cache miss, but instruction D is a cache hit, then instruction E should be able to execute before instruction B. By renaming R3, the WAW hazard between B and E is resolved, since the two instructions will write to different physical registers. The WAR hazard between instructions E and C is resolved for exactly the same reason.

In the DECODE stage, a Register Alias Table (RAT) manages this mapping. It has one entry for each of the eight architectural registers consisting of a 5-bit number and a valid bit; the 5-bit number specifies the physical register file entry which holds the necessary data, and the valid bit indicates whether the instruction which will produce that data has committed.

When using register renaming, one must know when a physical register is not being used at any point in the pipelines, so that we can reassign it for use. Typically, there is a relatively simple mechanism to ensure that a register is not reused before it is free; this mechanism requires in-order commit, which we did not implement. Solving this turned out to be a complex problem; our solution demonstrates our systems programming background. For each of the 32 physical registers, we keep a reference count. The reference count is incremented each time an instruction writes or reads that physical register and is decremented when an instruction commits. If the reference count is zero, we know that the register is no longer in use at any point in our processor, so we can assign it again when issuing instructions.

This mechanism for register re-use provides an interesting problem when combined with speculative execution. Specifically, when a predicted control instruction is issued, we need to at the very least back up the RAT mapping; this represents the current architectural state of the machine. We also decided to back up the reference counts. In the case of a misprediction, this allows us to flush all speculative instructions and immediately reuse their registers. Thus there is only a single-cycle penalty on a misprediction.

2.2.4 Out-of-order Retirement

Real processors require that precise machine state be kept so that interrupts and exceptions can be handled correctly. Consider the following instructions:

```
A: AND R6, R6, 0
B: LDR R5, R1, R2
C: ADD R6, R3, R6
```

Assume that all three of these instructions are fetched and issued simultaneously, and instructions A and C execute, but B must wait in the memory queue for several cycles. In a real architecture, memory accesses can cause access violations, page faults, or various other exceptional events. These events typically require that the program be re-started from the exceptional instruction. However, re-executing the program from instruction B would cause incorrect results,

since instruction C has already executed. To resolve these kinds of problems, most out-of-order processors include a reorder buffer (ROB) which maintains precise architectural state at each point in time. The LC-3b does not require that interrupts or exceptions be implemented, and so a ROB is technically not necessary. As a result, checkpointing on speculation suffices to ensure correct program execution. Rather than building a reorder buffer (ROB) to ensure that instructions are committed in program order, we simply let them commit out-of-order. The reasoning behind this was two-fold: it can potentially be simpler to implement and there are potential performance benefits. This aspect of our design is probably the most unique, since we could find very few papers discussing out-of-order commits and no real implementations.

Chapter 3

Design

3.1 Overview

AmdaZulo consists of four pipelines, two of which are dedicated to ALU operations and two of which are dedicated to memory operations. All instructions go through the FETCH and DECODE pipeline stages before being sent to the corresponding reservation station. The additional pipeline stages for ALU operations include ALU_RES_STN (ALU reservation station), REG (register file read), ALU (ALU execution), and WB (writeback) for a total of 6 stages. The additional pipeline stages for memory operations include MEM_Q (memory queue), REG (register file read), CALC_ADDR (calculate address), MEM (access memory), and WB (writeback) for a total of 7 stages.

3.2 Fetch

The FETCH stage is the initial stage in the operation of AmdaZulo and bears the responsibility for fetching four instructions at a time as well as the branch prediction mechanisms. The fetch stage communicates with the instruction cache (I-cache) in order to fetch four instructions at a time. A line in the I-cache is 256 bits, which corresponds to 16 LC-3b instructions. Four instructions are fetched in a single cycle and up to four are then sent to the DECODE stage. If there are any control instructions in the group of four instructions that are fetched, the FETCH stage will only issue up until the first control instruction. Furthermore, the next group of four instructions will not be retrieved from the I-cache until the previous four instructions have been sent on to the DECODE stage. The main reason for doing it this way was to prevent the logic from being too incredibly complex. For example, consider the following instructions being fetched in a cycle:

```
A: ADD R1, R2, R3
B: ADD R3, R4, R5
C: BRnz LOOP
D: ADD R4, R4, 1
```

During the first cycle, Instructions A through C would be dispatched to the DECODE stage and during the next cycle, Instruction D would be sent down

the pipeline to the DECODE stage, assuming that we predict the branch not-taken.

The Program Counter (PC) register resides in the FETCH stage, and typically contains a four-instruction aligned address. However, control instructions can set the PC to non four-instruction aligned addresses, so we have additional hardware to send down the correct PC+2 with the group of four instructions, which is then used by the DECODE stage.

3.2.1 Implementing TRAP

Typical control instructions such as branches and jumps can be predicted relatively accurately. However, memory-indirect control instructions such as TRAP allow for much more variability. Also, a mis-speculation across a TRAP would take a very long time to resolve. Thus when we encounter a TRAP instruction, we halt FETCH until the memory access has completed.

3.2.2 Branch-Target Buffer

In order to have a high-bandwidth instruction stream and keep our processor busy, we created a Branch-Target Buffer (BTB) that resides in the FETCH stage. The Branch-Target Buffer is essentially a cache indexed by the PC that has the branch target as its data. When a branch is resolved, its target is updated in the BTB. In this way, the processor is able to know whether one of the instructions being fetched is a branch and is able to predict its target.

Our implementation of the BTB only stores PC-relative control instructions (*i.e.*, JSR and BR) since most likely an indirect jump such as JSRR or JMP will not have the same target each time it is executed. In order to maximize the effectiveness of our BTB, we implemented it as a 128-way (fully associative) cache. As a result it stores a total of 256 bytes of data. As with all of our caches, it implements true LRU replacement, so it should be able to store the targets of 128 branches before replacing any.

3.2.3 Tournament Predictor

Working in conjunction with the BTB is the Tournament Predictor (TP), which allows for predictions of whether a branch instruction is taken or not. After reading about various types of branch prediction schemes, we determined that the Tournament Predictor had the best performance. As specified in the objectives section, good branch prediction is necessary in order to make speculation worthwhile and ensure fast total execution time.

A tournament predictor combines two branch prediction algorithms. This scheme attempts to choose the best performing branch prediction algorithm for a given address.

The first algorithm we use is *gshare*, which keeps a register of the last p predictions. The p for our predictor is seven bits. Whenever a branch is resolved, the result of the branch is shifted into the aforementioned register. In addition, a two-bit saturating counter is updated according to whether the branch was predicted correctly or incorrectly.

When we attempt to resolve a branch, we take the exclusive or of the branch's address and the shift register to index into a table of saturating two-bit counters.

If the result is either two or three we predict the branch taken, otherwise we predict untaken.

The other algorithm we chose was a bit simpler. This method uses the low seven bits of the PC to index into a table of two-bit saturating counters. The branching thresholds for counter values remained the same between the two algorithms. When the branch is resolved, the appropriate counter is updated accordingly.

We choose between the two predictors by using a table of two bit saturating counters that is indexed by the branch's address. When a branch is resolved correctly using a specific algorithm, the table of counters is updated. Using this scheme we can initialize the *gshare* predictor without incurring the initial large misprediction rate. In addition, the table based predictor works particularly well when the number of times a branch is seen in a program's lifetime is relatively small.

3.3 Decode

The DECODE stage of the pipeline is responsible for maintaining the Register Alias Table (RAT), the reference counts, and decoding instructions from their 16-bit format to a set of 32 control bits usable by the stages following DECODE. This is the most complex stage of the processor and, except for the caches, the most expensive, since it must not only maintain the active copies of the RAT and the reference counts, but also the checkpoints which have been made due to speculation.

3.3.1 Register Alias Table (RAT)

The Register Alias Table (RAT) maintains the mapping between the architectural registers (R0 through R7) and the physical registers (P0 through P31). Through register renaming, the RAT allows AmdaZulo to avoid WAW (write after write) and WAR (write after read) hazards.

3.3.2 Reference Counts

As registers are renamed in the DECODE stage, a method must also exist to determine which physical registers are available for use. The method we devised was to use reference counts on each of the physical registers. Whenever a physical register is used as a destination or source operand, the reference count for that physical register is incremented. Later, when the instruction enters the WB stage, the corresponding reference count is decremented. Additional complexity is added to keeping track of reference counts due to the implementation of speculation. When the processor begins to execute speculative instructions, the active reference count is moved into a backup. A total of two backups exists, which means that we can speculate across at most two control instructions (see "Control Unit" section).

When speculating, the active reference counts are incremented for the speculative instructions that are issued. However, if non-speculative instructions commit, they decrement the corresponding reference count backup, which is necessary so that speculative instructions are kept separate from non-speculative

instructions in the case of a misprediction.

3.3.3 Control Unit

As the name implies, the control unit handles control instructions. Although it sits outside of the DECODE stage on our design, it functions in parallel with the DECODE stage. Its main function is to resolve control instructions and back up control state across control instructions for speculation. In order to implement the back up, we have two control "reservations stations" that contain the backup of all the control state information, such as the NZP bits. The control unit also reports back to the FETCH stage when a misprediction occurs or a branch is resolved. In either case, the correct target address of BR and JSR instructions is returned to be stored in the BTB. The Tournament Predictor in the FETCH stage is also updated with whether the control instruction was predicted correctly or not.

3.4 Reservation Stations

AmdaZulo uses two sets of "reservation stations": one group of 16 for ALU operations and another group of 16 organized in a memory queue for the memory instructions. (Note: the "reservation stations" used by AmdaZulo are not exactly the same as those described in Tomasulo's algorithm, but perform a similar function.) The reservation station component that forms the basis for the ALU reservation stations and memory queue is essentially the same.

The reservation station's purpose is to buffer an instruction until its operands are ready. For each of the valid operands that an instruction needs, the reservation station has a tag stored. The tag is a five bit value that corresponds to the physical register that it is waiting for to be ready. If the data corresponding to that register exists, the DECODE stage sets the valid bit on that operand's tag. Otherwise, the reservation station listens for the tag it needs on the four Common Data Buses, which come from the WB stage. Once an instruction has all its operands, it raises the ready bit to signal that it can be executed.

3.4.1 ALU Reservation Stations

AmdaZulo contains 16 ALU reservation stations along with placement and selection logic that make up a single stage of the ALU pipeline. The placement logic chooses an empty reservation station for the instruction coming from the DECODE stage. The selection logic chooses up to two instructions that have their ready bits set and sends one down each ALU pipeline. If fewer instructions are ready to execute, NOPs are sent down instead.

3.4.2 Memory Queue

As described in the "Out-of-Order Memory Instructions" section, any form of store instruction serializes memory instructions, but load instructions that fall in between store instructions can be executed out-of-order. As instructions come down the pipeline from the DECODE stage, the placement logic inside the memory queue places the instruction in program order into a queue of 16

reservation stations. As instructions are added to the queue, the "tail" pointer gets incremented to point to the next free reservation station in the queue. There also exists a "head" pointer that points to the first unexecuted instruction in the queue.

The job of the selection logic is to select up to two instructions to send down each memory pipeline. If there is a store at the head of the queue and it is not ready, then the NOPs are sent down in order to keep store instructions in order. However, if the head points to a series of load instructions that are ready, they can be executed in any order. After each clock cycle the head and tail are both incremented by zero, one, or two, depending on how many instructions were placed and executed that cycle.

Speculation within the Memory Queue

Adding an extra layer of complexity to the memory queue is the implementation of speculation. In order to execute memory instructions speculatively, we created a "speculative cache" (see "Speculative Cache" section). The idea is that speculative store instructions write to the small speculative cache in order to prevent the data cache from being overwritten in the case of a misprediction. Speculative load instructions read from both the speculative cache and the data cache at the same time. If there is a hit in the speculative cache, that value is used, otherwise the data from the data cache is used. In the case of a misprediction, the speculative cache is invalidated.

In order to support this speculative cache, we decided to send down speculative stores down the pipeline twice: the first time when it is speculative and the second time after it has resolved and is no longer speculative. The idea is that the first time, the data only gets stored into the speculative cache and when we resolve the store as valid, it should go into the data cache. Support for this operation adds a bit of complexity to the memory queue selection logic because a speculative store will remain at the head of the queue after it is sent down the first time, so that it can be sent down again when it is resolved. However, after the speculative store goes down once, the instructions in the queue following the store can now proceed. As a result, if a speculative store is followed by a lot of speculative loads, it is possible to have many empty spots in the queue between the head and the tail. Since the head only increments by at most two each time, there could potentially be a few cycles of "catch up" after the resolution of a speculative store.

3.5 Register File Read Stage

The register file read stage exists in both the ALU and memory pipelines. It consists simply of reading from the 32-register physical register file. We made our register file have four ports, so that we could be reading from each of the four pipelines at once. We were required to make the register read delay 28 ns, so we turned the read into a pipeline stage.

3.6 ALU Stage

The ALU stage is in the two ALU pipelines and consists of performing the appropriate ALU operation such as ADD, AND, NOT, SHF, or LEA.

3.6.1 Carry Lookahead Adder

Rather than using the given adder with a 32 ns delay, we built our own carry lookahead adder that only has a delay of 9 ns.

3.6.2 Barrel Shifter

Rather than using the provided shifter, we built a barrel shifter in order to speed up shift operations.

3.7 Writeback Stage

The writeback stage is where instructions broadcast their tags and data on the Common Data Buses (CDB). The data results are also written to the register file during the first half of the cycle in order to keep it up to date (reading from the register file is done during the second half of the clock cycle). ALU pipeline 0 is connected to CDB0, ALU pipeline 1 is connected to CDB1, memory pipeline 0 is connected to CDB2, and memory pipeline 1 is connected to CDB3. The reservation stations all listen to the CDBs for the tags on which they are waiting. Additionally, the RAT listens on the CDBs in order to determine when the values to which architectural registers map are ready.

3.8 Caches

All caches in the memory system are fully associative. The delays associated with the cache lines are only incurred on a write or when a read causes a set-switch; full associativity eliminates the second cause and makes reading from caches extremely fast. The amount of hardware required to build a 128-way associative 4-kilobyte cache is enormous. However, it is also very repetitive. We discovered that by structuring the components into a binary tree, we essentially only needed to do an amount of work which grows logarithmically (rather than linearly) with the amount of hardware built. Essentially, the top level of the Instruction cache and L2 data cache datapaths consists of 2 64-way fully associative caches; each of which consists of 2 32-way fully associative caches, *etc.* The ability to make a single-cycle read 4-kilobyte instruction cache eliminated the need for a unified L2 cache. As a result, the I-cache and the L2 D-cache both interact directly with a DRAM arbiter, which in turn interacts directly with DRAM. The total amount of cache memory in AmdaZulo is 8.5 kilobytes.

3.8.1 L1 Instruction Cache

The I-cache is a single ported 4-KB, fully associative cache with 256-bit cache lines. Exact LRU is used as the replacement strategy among the 128 cache lines. Access time for a cache line is 13 ns.

3.8.2 L1 Data Cache

The L1 Data Cache (D-cache) is a dual-ported cache to support reads and writes from both memory pipelines in a single cycle. It has a total size of 512 bytes, a line size of 256 bits (32 bytes or 16 instructions), and is fully-associative with 16 ways. The line-replacement scheme is LRU, but the LRU can only be updated by a single access in a given cycle. Thus some accesses do not affect the LRU calculation, but this approximation has worked well in tests. The delay for a read-hit is 10 ns, and the delay for a write-hit is approximately 15 ns longer.

3.8.3 L2 Data Cache

The L2 D-cache is 4-KB, single ported, and fully associative, with the same read time as our L1 Instruction cache (13 ns). Writes require 60 ns longer, due to the cache's size.

3.8.4 128-way Exact LRU

The 128-way exact LRU used in both the Instruction and L2-Data caches was built hierarchically; its top level diagram consists of two 64-way LRU's, each of which is a 32-way LRU, *etc.* It is organized as a 128-entry list; when an access occurs, the each entry in the list up until the accessed entry shifts backward, and the accessed entry is moved to the front. The amount of hardware required for this is enormous, since each of the 128 entries must know whether any of the previous entries was the matched entry. Essentially, each entry needs an enormous OR gate to detect the result of a comparison of the MRU entry with each previous entry.

3.8.5 Speculative Cache

The speculative cache is a small 32-byte cache that can be invalidated on a misprediction, and stores the results of speculatively executed stores. It is dual-ported and consists of a single cache line. A speculative store which misses in the speculation cache stalls the memory pipeline.

3.9 DRAM

Our DRAM has 256-bit bandwidth and a total size of 64 KB.

Chapter 4

Performance

Performance statistics were gathered by running test code that was provided in the course (finalcode.asm) as well as test code that the group wrote ourselves (madd.asm). One of the group members, Mark Murphy, wrote an LC-3b C compiler, with which we were able to generate LC-3b test code more easily than writing it by hand.

Results from simulations of the test code were parsed using Perl scripts. After simulating a piece of test code to completion, we saved relevant signals with the "List" tool in ModelSim, and then parsed the results with Perl. The graphs were generated using a combination of Perl scripts and gnuplot.

The two test programs we ran were madd.asm and finalcode.asm. Madd.asm is a program that adds a lot of vectors in a random order and as a side-effect it touches a lot of cache lines. The finalcode.asm program is the code that we were given to demo our processor.

4.1 Performance Evaluation

Figure 4.1 shows the number of instructions we commit in a single cycle while running finalcode.asm. During about 37% of the cycles we do not commit any instructions, which is mostly due to cache misses. About 38% of the time we commit a single instruction, 19% two instructions and 3% three instructions per cycle. Although it is not evident from the graph, there are a few cycles where four instructions are committed in a cycle. Even though it is possible that five instructions could commit in a cycle (*e.g.*, two ALU instructions, two memory instructions, and a control instruction), it never happened in this test code.

Figure 4.2 shows the same type of statistics as Figure 4.1, except that the code being executed is the madd.asm program. Overall, this program does not commit quite as many instructions per cycle as the finalcode.asm program, which is most likely due to the large number of cache accesses performed in madd.asm.

Figure 4.3 is a bar graph of our branch prediction accuracy. It should be noted that these predictions relate only to the branch instruction and not indirect branches. On average we obtained 90% accuracy using our tournament predictor. We considered implementing the "single wire" predictor, but we felt that it was in our favor to consider the branches taken. By taking the branches,

we add the entry to the BTB and also fetch the cache line associated with the branch's target; both of these are acceptable behavior due to the likelihood that the target of the branch will be executed in the future. Additional performance would come from code optimized for our architecture. The hand written loops found in the two programs tested do not fully utilize our branch prediction algorithm.

Figure 4.4 is a bar graph of the overall control instruction prediction. Madd.asm performed significantly better in predicting control instructions than the finalcode.asm. At the time of running this code target prediction with indirect control instructions was not implemented. This effect can be clearly seen when observing the performance of the finalcode program. The finalcode.asm program is comprised mainly of subroutine calls as well as return from subroutine calls; while the madd program is comprised mainly of branches. What the graph shows is that 45% of the time, while executing finalcode.asm, we are doing useless work. We are certain that a basic implementation of indirect target prediction will increase the amount of correct work done per cycle by many magnitudes.

Figure 4.5 shows the structural hazards due to a full physical register file (reg), full ALU and memory reservation stations (res and mem), and full control backups (ctrl). Madd.asm has a large number of register hazards, which is mainly due to its random writing of registers; the same problem does not appear in finalcode.asm. Neither the ALU or memory reservation stations are a structural hazard, which means that the 16 we built were enough. The most important piece of information that can be extracted from this figure is the large percentage of control hazards due to the fact that we can only speculate across at most two control instructions. It is obvious that it would be better to speculate across more branches, but in order to make this improvement worthwhile, we need to increase the accuracy of our target prediction; otherwise we simply execute incorrect instructions, which is useless.

Figure 4.6 is the cycles per instruction analysis. The average CPI of our processor is approximately 1.2. This number indicates two things. First of all, our processor would benefit significantly from a compiler that optimized for our architecture, which would allow us to approach the theoretical limit of 0.2. In addition, a significant decrease in CPI would come from implementing indirect target prediction. Our current result is still a significant increase over the standard five stage pipeline, and with slight modification our performance will increase significantly.

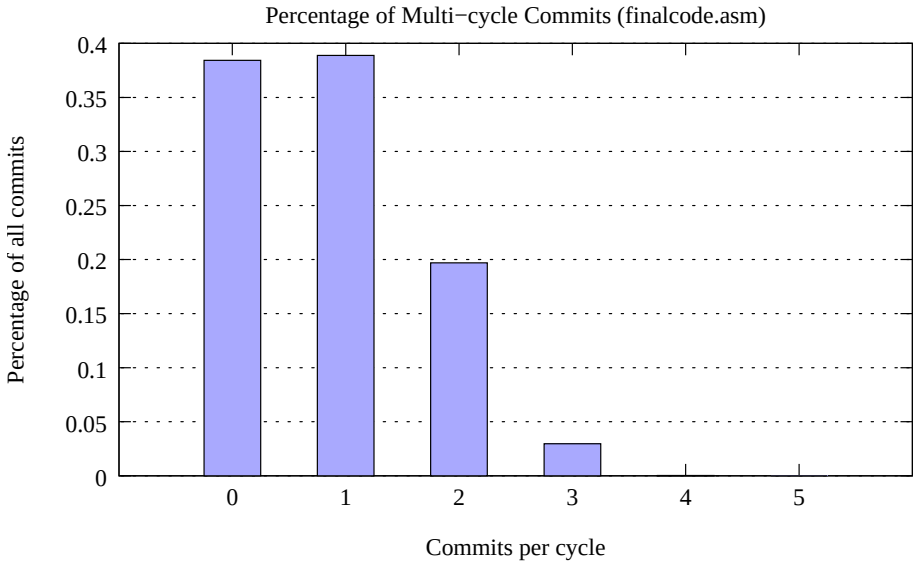

Figure 4.1: Per-cycle commits when running finalcode.asm

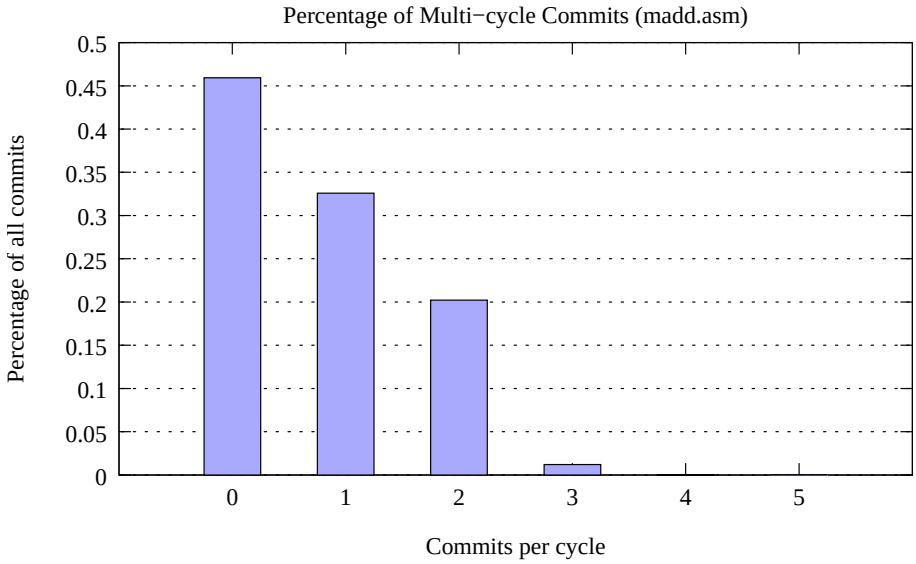

Figure 4.2: Per-cycle commits when running madd.asm

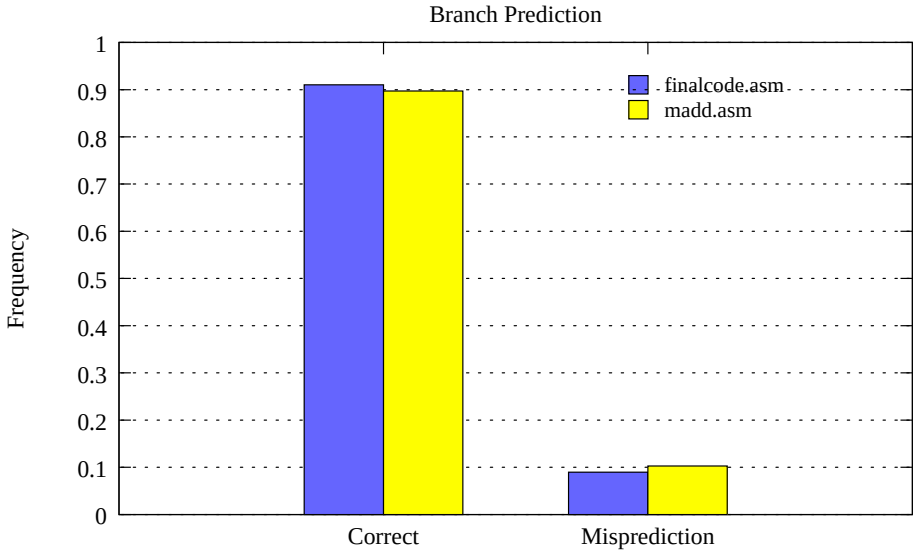

Figure 4.3: Branch Prediction Frequency

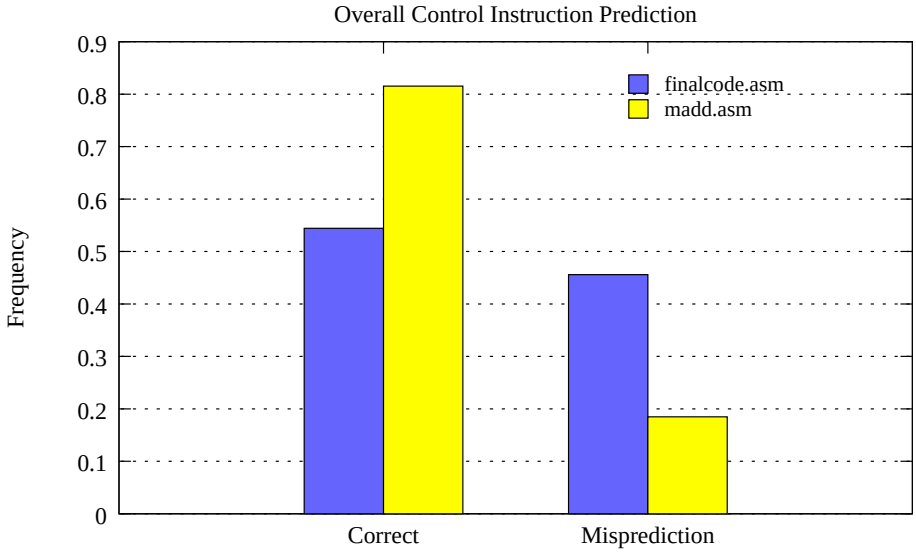

Figure 4.4: Overall Control Instruction Prediction Frequency

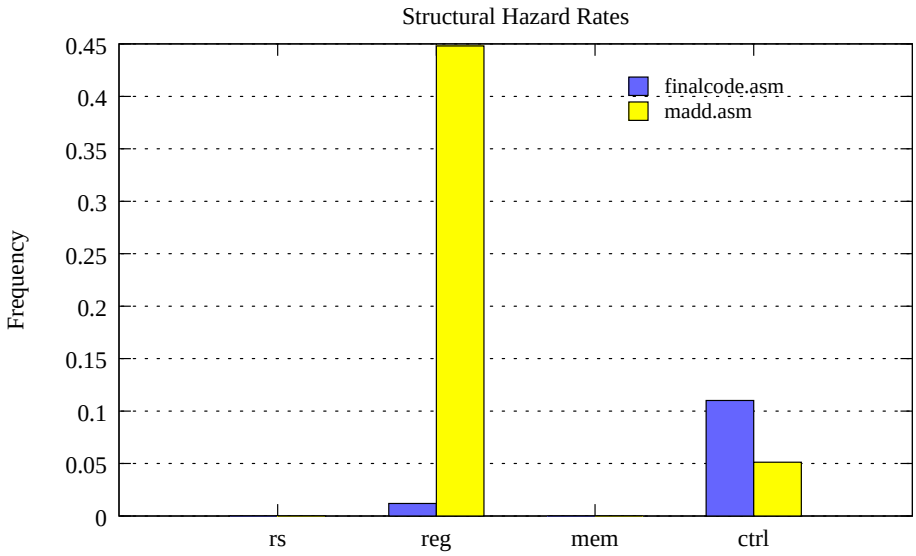

Figure 4.5: Structural Hazards

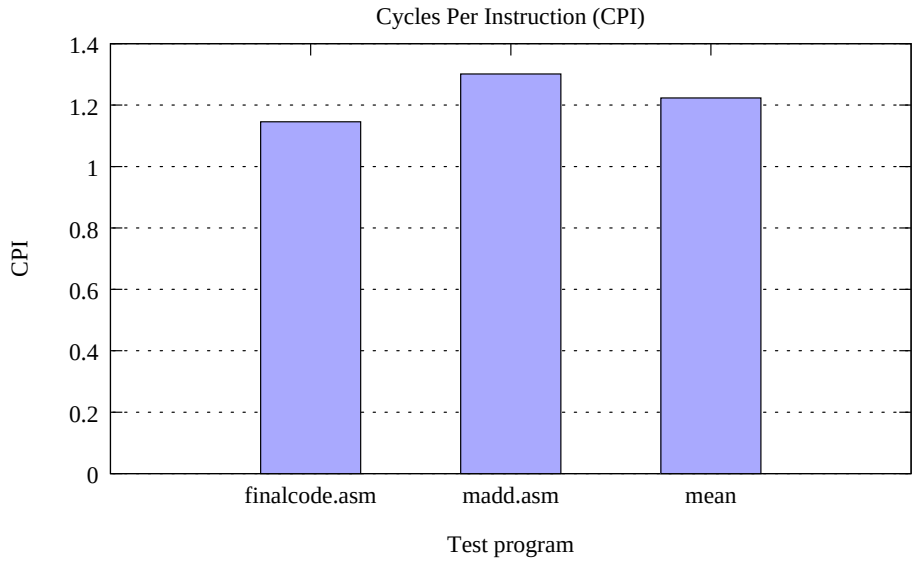

Figure 4.6: Cycles Per Instruction (CPI)

Chapter 5

Cost

We calculated that the cost of the processor is approximately 450,000 units, by the cost metrics presented on the course web page. The majority of the cost is due to the cache hierarchy, which comes to about 400,000 units with the datapath requiring the remaining 50,000 units.

5.1 Cache Hierarchy

Fully associative caches require very large amounts of hardware to implement. Our implementation makes this manageable to build, however it is still unrealistically expensive to implement in silicon. In particular, there must be some way to multiplex between all of the data lines in the cache in order to select the correct data. Our implementation masks out all lines but the desired one based on the tag comparisons, and then ORs all of the resulting lines together. Essentially, it is a multiplexer with the select lines pre-decoded. This scheme requires a total of 65,280 logic gates; in the cost calculation scheme of this course, each gate costs 2 units. Since the cost of the data stores themselves are 3000 units, and there are also 128 11-bit tag-comparators required, the total cost of each of the the 4 KB fully associative caches used is approximately 142,520 units before the cost of the LRU logic is taken into account.

The LRU logic consists of 896 bits of state, 128 7-bit comparators, 128 2-input 7-bit multiplexers, and 127 127 input OR gates. Technically, 127 inputs are unnecessary for all but the last OR gate in the chain, but to allow for the same component to be used for each, 128 input gates (consisting of a tree of 2-input gates) were used. The total cost is 42,868.

The total cost of a 4 KB fully associative cache is at least 185,388 units. This is surely a lower bound, since more logic was necessary than was included in the computation. However, that logic was minuscule compared to the components which needed to be replicated 128 times.

AmdaZulo included 2 of those caches and a 512 Byte fully-associative L1 data cache, whose cost can similarly be calculated at approximately 38,722 units. So, the cost of the memory hierarchy alone is approximately 409,498 units.

5.2 Datapath

The cost of the Datapath is more difficult to compute. The main cost is centralized in the Decode stage and reservation stations, so we will focus there.

There are a total of 34 reservation stations, each holding 44 bits of state in registers; the cost is 254 units. Each reservation station has 8 5-bit comparators for tags on the CDBs, bringing the total cost of the reservation stations to 1410.

Our register file is 32-entries, and has 8 read ports and 4 write ports. The total cost is 4,160 units.

The cost of DECODE is split between the register re-use logic and the register alias table. In order to detect the registers used by instructions committing and instructions being issued, we have a block containing 768 5-bit comparators. The cost of this block is 26,112 units. To maintain the reference counts requires 192 bits of state per backup. Including the cost of the RAT which is at least 3000 units, decode costs at least 32,082 units.

In all, we estimate that the cost of the Datapath is less than 50,000 units.

Chapter 6

Additional Observations

6.1 Superscalar in a Semester

Given the complexity of a superscalar design, and in particular our design, one might wonder how it is possible to successfully implement it in a single semester with only three people. The answer is simply an *enormous* amount of time. Every minute outside of class was spent working on the design and building it. For the past 10 weeks, we spent roughly eight hours a day working on the project. We estimate the total combined time that we worked on the project was *at least* 1000 hours. Furthermore, we almost always worked as a group of three, which was especially important with a design of this magnitude. The design was masochistically hard, but in the end we managed to have a successfully working design of which we are all extremely proud.

6.2 Things we Learned

In building a system of this magnitude, the benefits and costs of each design decision become apparent in many unexpected places.

The ability to commit 5 instructions simultaneously was clearly overkill. The simulation data shows that cycles in which 4 instructions commit are extremely rare; 5 instructions were never committed in the same cycle. Perhaps with better optimized code, the parallelism of our machine could be more fully utilized; however, until ucc supports optimizations this will not be realized. Alternatively, a scheme of memory address disambiguation to allow independent stores to execute out of order would allow for more complete utilization of the parallelism of our memory pipelines. Such schemes are inherently complex and difficult to implement, but possibly justified.

A reorder buffer would simplify several aspects of our design. The first would be register re-use. While utilization of the registers would decrease slightly, the structural hazard posed by the reference-counting would be eliminated. The scheme of register re-use used in other register-renaming machines which relies upon in-order commit is simpler to implement, and does not limit the number of outstanding control instructions. Additionally, a ROB could eliminate the need for RAT backups as well, if a single non-speculative copy of the RAT were maintained by the ROB. However, these simplifications make single-cycle

misprediction recovery more difficult. The performance trade-off can only be determined by further experimentation, but the hardware cost clearly favors the ROB.

Chapter 7

Conclusion

A superscalar design is an ambitious task for three people to build in a single semester. Although our performance was not as fast as the theoretical limit, it is faster than a five stage pipeline design and *works* despite the enormous amount of complexity. We can significantly improve our CPI by performing indirect target prediction, which we plan to implement for the competition. We achieved all of the goals we set out to accomplish including out-of-order execution, register renaming, out-of-order retirement, speculation, and four-wide execution, and learned a significant amount in the entire process.