Ian Daniher and Jimmy Wu
CompArch Lab3
12/11/12

Introduction

Lab three was centered around the expedient design and analysis of a basic CPU. Framed as a last minute proof of concept for an investor demo, we needed to be able to make an argument for both the function and efficacy of a CPU core that would be used to control a Turboencabulator, executing, per control loop iteration, an update of an eight tap FIR, six trigonometric calculations of sine and cosine[1], and the square root of a number. Upon setting out on this task, we prioritized analysis of the required tasks and the creation of a modular, single cycle CPU with a high percentage of test coverage.

Deliverables and Test Results

Planning and analysis of the problem

Our timing analysis occupied a nontrivial amount of our total work time, but yielded extremely useful information and influenced our inclusion of an ALU capable of performing single-cycle multiplication and division. The full discussion can be found at https://github.com/itdaniher/ca-Lab3/blob/master/README.mkd, but we estimated the calcuation of an FIR to require 24-48 operations depending upon the location of the constants and data, the execution of CORDIC to require approximately 148 operations per input, for at total of 888 operations per six vanes, and the execution of the newton-raphson square root estimation algorithm to require between 24-60 operations, depending upon the exact input. This sums to a worst case per-tick operation count of approximately one thousand operations.

With the operations required by our target user fully analyzed, we elected to build a single cycle, MIPS-based CPU with an ALU that placed high emphasis on function count over optimization. Many of the algorithms required necessitated expedient multiplication and division. We started our work on this project by identifying the key components of our soon-to-be CPU, divided the tasks appropriately, and built an ALU, a decoder, and an instruction memory / fetch element. Each module executed on the falling edge of a clock, and contained an independent test / design-verification module. The creation of these modules was fairly straight forward, but the decision of synchronous vs. asynchronous execution took several revisions until we were content with the reliable function of each part of our CPU. After these three modules were completed, we set out to write a macromodule that contained the remaining aspects necessary for the function of our CPU, namely registers, RAM, a program counter, and a mechanism for the repeated execution of operations.

---

[1] misunderstanding of lab request - approved by Eric VanWyk

ALU

Our ALU can be found at https://github.com/itdaniher/ca-Lab3/blob/master/alu/alu.v#L1, also included for your convenience in this archive. As mentioned earlier, we whole-heartedly embraced design decisions optimizing for functionality over efficiency, and optimized for compatibility with the MIPS instruction set over refinement. Our final ALU included support for all boolean logic operations, as well as more complex arithmetic, including multiply, divide, and support operations. Given sufficient time, it would be worthwhile to attempt to combine operations such as addition and subtraction, and analyze the possibilities for structural overlap between multiplication and division. Additionally, our ALU does not differentiate between signed and unsigned numbers, a functional difference which could be easily remedied by the addition of extra carry / catch logic. Our ALU is tested-working.

Instruction Fetcher

Our instruction fetcher, located at https://github.com/itdaniher/ca-Lab3/blob/master/fetch/fetcher.v#L1, and also included in this archive, implements a very simple abstraction on top of Verilog's "$readmemh" functionality. For actual synthesis, some similar abstraction built on non-volatile memory would be needed, but this served to verify the functionality of all parts of our CPU. For the isolated testing of this module, a very simple python script, located at https://github.com/itdaniher/ca-Lab3/blob/master/fetch/progmemGen.py, was written, to populate a file "progmem.hex," with the first 65,536 integers as hexidecimal numbers. This provides a one-to-one mapping between input address and output value, allowing for quick visual design verification. Our fetcher is tested-working.

Instruction decoder

Our instruction decoder, located at https://github.com/itdaniher/ca-Lab3/blob/master/decode/decode.v, and also included in this archive, implements simple bitslicing logic to divide a thirty two bit instruction into smaller words conveying meaning semantically bound to the instruction type, information only contained in our higher-level cpu module. That said, this decoder was constructed based upon information located at http://en.wikipedia.org/wiki/MIPS_architecture#MIPS_assembly_language and corroborated by the MARS emulator. Our test method was to feed in thirty two bit instructions and observe that they were split into smaller bit length words in accordance to expected functionality.

Post-mortem reflections

Our goal was to have a fully functional assembly library which would allow the customer to perform the actual square root, CORDIC, and FIR operations on our implemented CPU.

Our plan was as follows: for each of the operations, do a timing analysis by enumerating the instruction breakdown. Figure out exactly which instructions are needed. Then, make a high-level design of the CPU with these timings in mind. Next, transfer the high-level diagrams into a Verilog API. Finally, implement the Verilog API.

We got as far as implementing the Verilog API. The timing analysis simply took too long. We started out downloading C algorithms for each of the operations, compiling the C code, and using a MIPS decompiler to translate the machine code directly into assembly. This would allow us to run through the assembly to obtain counts of which instruction were being used and how many times per cycle. However, we noticed that this was taking too long, so we scrapped our work. Instead, we researched the algorithms for calculating the required operations and determined theoretical counts and timings for each instruction.

Honestly, I doubt that we could have tackled this differently. The nature of this project is that the implementation (writing code) is easy, while the understanding of how to design the top-down structure of how the parts fit together is hard. We were very analytical in our approach and this allowed us to gain an understanding of what we were trying to optimize and how we can make design choices that fit the optimization. The alternative would have been to just jump in and get an un-optimized CPU working, but I suspect this would have taken even more time and led to an increased amount of frustration.

Perhaps if we were working at a real hardware company we would have just cranked that CPU out and been done with it, but the purpose of this assignment was to help us learn and solidify our understanding of how CPU design works. I personally believe I would have gotten much less out of this assignment if we had gone the other way.

As mentioned earlier, we observed numerous ways that our design could be enriched to better meet our customers needs. To provide a recap of perhaps the three most relevant, see the following brief discussion.

- ALU Efficiency
- Synthesizability Optimizations of RAM and NVM
- Assembly library
- Methods to load and store data from the outside