RTL through OS

Reid Long
Electrical and Computer Engineering
Carnegie Mellon University
Pittsbugh, PA
relong@andrew.cmu.edu
reidlong@icloud.com

Teguh Hofstee
Electrical and Computer Engineering
Carnegie Mellon University
Pittsbugh, PA
thofstee@andrew.cmu.edu

***Abstract*—Over the last several decades computers have grown increasingly complex. Conventional wisdom indicates that it is impossible to fully understand all of the moving parts in a modern processor; we challenged the status quo in a quest to see how close we could get to understanding a complete computer. Starting at the abstraction of register transfer logic, we developed a single core RISC-V processor with a custom implementation of the RV32IAS specification then wrote microkernels to run on the processor we implemented. Key features included a memory system with a unified Instruction and Data cache, an eight stage pipeline, an interrupt controller, timer interrupts, PS/2 keyboard (delivered as interrupts to the kernel), a color VGA display, and a suite of performance counters. Furthermore, we wrote a collection of benchmarks to evaluate the correctness of the system and microkernels to demonstrate the processors capabilities.**

***Index Terms*—RTL, RISC-V, Operating System, Cache, Processor, Architecture**

I. Introduction

"What I cannot create, I do not understand" - Richard Feynman[1]

Richard Feynman's quote summarizes our approach to our Electrical and Computer Engineering Design Experience capstone. We strove to demonstrate mastery of the summation of our experiences at Carnegie Mellon University by building a quintessentially Electrical and Computer Engineering product, a computer[2]. Reinventing a modern multi-core, out-of-order, superscalar processor in a semester would have been impossible, so we were forced to select what was most relevant to our undergraduate experience and fine tune a project that was novel, challenging, and exciting. We developed a single-core RISC-V processor which would run on a Zedboard's Zynq-7020 fabric (more commonly known as a soft core).

II. Design Requirements

A full specification of the processor can be found on the team website [2], so we will only include the high level requirements here.

- R1: The processor shall implement a custom subset of the RV32IAS instruction set architecture.
- R2: The processor shall be consistent with the RISC-V User Level Instruction Set Architecture v2.2 [3]
- R3: The processor shall be consistent with the RISC-V Priviledge Architecture v1.10 [4]
- R4: The processor should execute microkernels in real time[3]

From this specification, we derived two key metrics to monitor throughout the development process: correctness and performance (measured in runtime).

We measured correctness based on how many binaries we were able to execute successfully. These binaries included constrained random assembly programs, hand crafted edge case tests, and microkernels to demonstrate capabilities during presentations.

Since we had millions of dynamic and static instructions in our tests, it would have been infeasible to manually verify the correct output. Instead, we developed an architectural simulator for the specific RV32IAS architecture we planned on implementing. We used this to generate golden copies of the architectural state including register dumps, video memory dumps, and a checksum of main memory. In addition, we included SystemVerilog concurrent assertions throughout the microarchitecture to verify that our assumptions were satisfied during execution. When all three models of the architecture were combined, we were able to efficiently identify ambiguities in the implementation. When an ambiguity was detected, we would manually inspect to determine which of our three components (architectural simulation, microarchitecture, concurrent assertions) was inconsistent.

The secondary metric was performance. Based on our analysis of a microarchitecture similar to what we planned on building[4], we set optimistic performance goals of 0.7 instructions per cycle (IPC) running at a clock rate of 100Mhz with forward branch prediction accuracy of 85% and backward branch accuracy of 99.9%. These estimates were based off of a microarchitecture that modeled zero cache misses (magic single cycle memory) and had fewer features; however, it was the closest model to the microarchitecture we were planning on building. After calibrating our expectations, we evaluated what performance goals we would need to achieve for the microkernels to be able to execute in real time.

[1] American Theoretical Physicist, Nobel Prize in Physics 1965

[2] As an added benefit to our project we have been able to increase our value in the event of nuclear holocaust or zombie apocalypse [1]

[3] Real time is defined by the response time a human can perceive. This is a meta metric of cache performance, clock frequency, and instructions per cycle

[4] This was based off of the microarchitecture from 18-447: Introduction to Computer Architecture

In order to boot the largest potential binary within within 30 seconds, we estimated that a clock frequency of 33Mhz would be sufficient, but decided to set the goal at 100Mhz to include a margin of error in the design (100Mhz clock frequency would provide an estimated 10 second boot time and is the native frequency of the Zedboard's Zynq-7020 fabric).

III. Architecture

The entire system is running on a ZedBoard interfacing with DRAM, PS/2, and VGA. The processor implements the RV32IAS specification which is the architectural contract with the software microkernels. Figure 1 depicts the contracts between key components. The ZedBoard has 512MB of DRAM available for use, but we artificially limit the amount of DRAM available to our RISC-V processor. We reserve 128MB of DRAM for the Arm core. This allows us to develop debugging probes using the Arm core without interfering with the memory available to the RISC-V processor.

There are two significant changes from the original design report [9]. The first is that we reevaluated scope of the project to not include a virtual memory system (and thus not include a multi-tasking kernel). This change was decided shortly after the design proposal was submitted when we realized that the proposed schedule underestimated the complexity of the getting I/O working on the system. In particular, implementing a working DRAM interface was surprisingly challenging and severely delayed our proposed schedule.

The second change is that we switched from booting off of an SD card to booting off of the hard Arm core on the Zedboard. This simplified our boot process and also enabled us to use the Arm core as a debug probe while running the synthesized design.

A more minor change from our original proposal is that we did not synthesize integer multiplication/division/remainder (the "M" extension). After implementing and validating these instructions in simulation, we attempted to synthesize them, but the design was unable to meet timing. We decided that software emulation would be sufficient and prioritized other features.

Figure 2 is a high level model of the interfaces within the processor. The "System Description" section includes additional details about the microarchitecture of the processor including references to datapath design.

IV. Design Trade Studies

Table I is a summary of the entire regression suite we used for validating correctness. We report the overall values across the entire benchmark suite and the best value achieved for any specific benchmark. To provide additional perspective into the performance metrics, we have also included plots of individual benchmark results in Figure 3, Figure 4, Figure 5, and Figure 6.

All of our benchmarks were executed with a 50Mhz clock frequency. We were able to synthesize the core at 98MHz; however, due to limitations with Vivado cross domain clocking we were restricted to running at power of 2 multiples of the

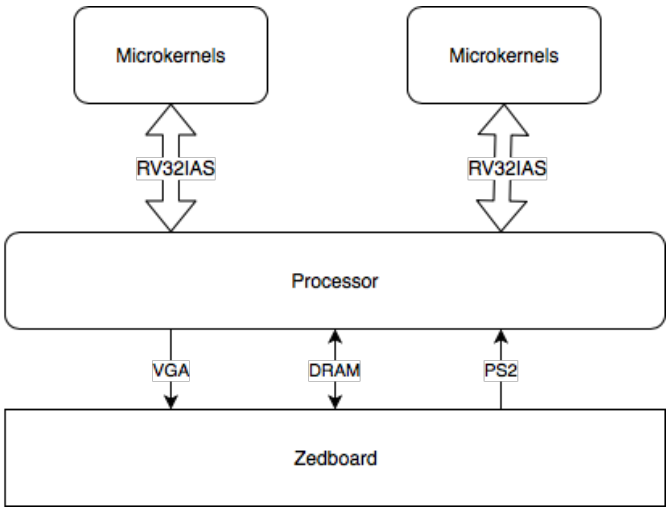

Fig. 1. High level functional block diagram with interfaces

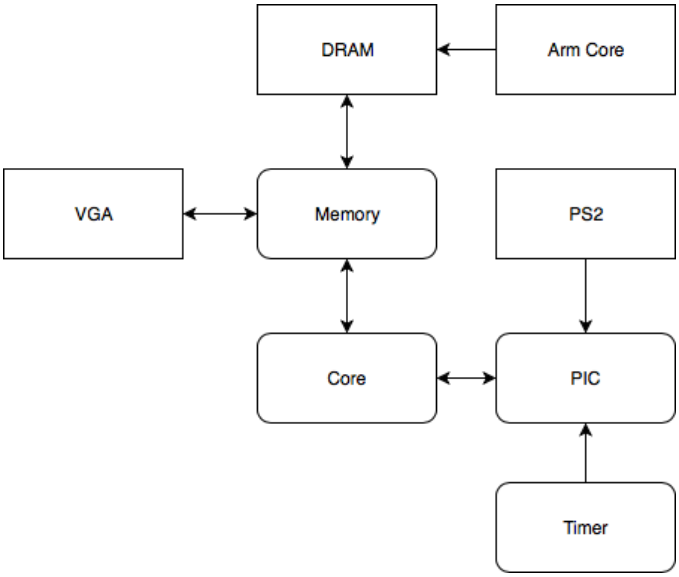

Fig. 2. High level processor microarchitecture block diagram

TABLE I
Overall Benchmark Measurements

Metric	Overall	Max
IPC	0.47	0.99
Cycles	22.2M	10.1M
Instructions Retired	10.5M	5.0M
Branch Predictor Accuracy	99.4%	99.999%
Forward Branch Accuracy	84.2%	100%
Backward Branch Accuracy	99.8%	99.98%
Forward Jump Accuracy	58.2%	99.3%
Backward Jump Accuracy	99.7%	99.999%
Instruction Hit	98.7%	99.9999%
Data Hit	99.2%	99.9%
Binary Size	203.8MB	9.5MB

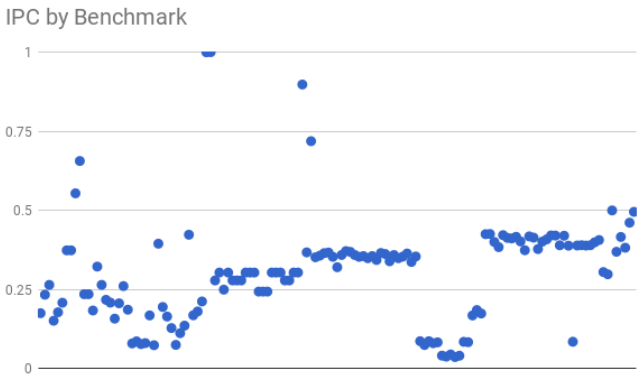

Fig. 3. Instructions per Cycle by benchmark

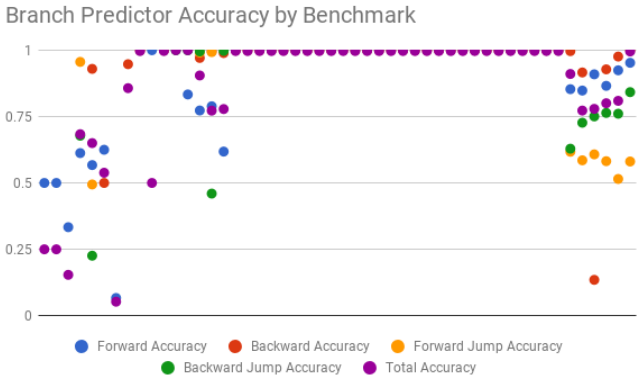

Fig. 4. Branch Predictor accuracy by type and by benchmark

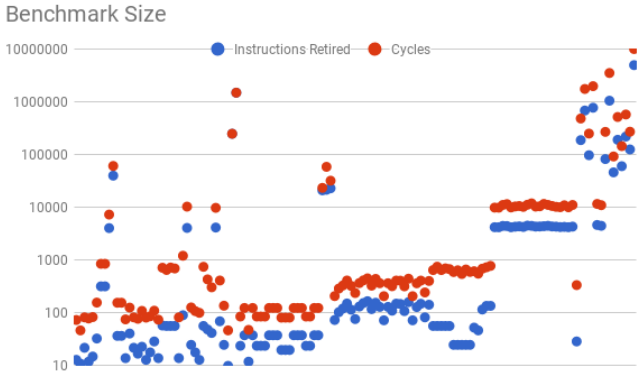

Fig. 5. Benchmark size in terms of cycles and instructions retired by benchmark

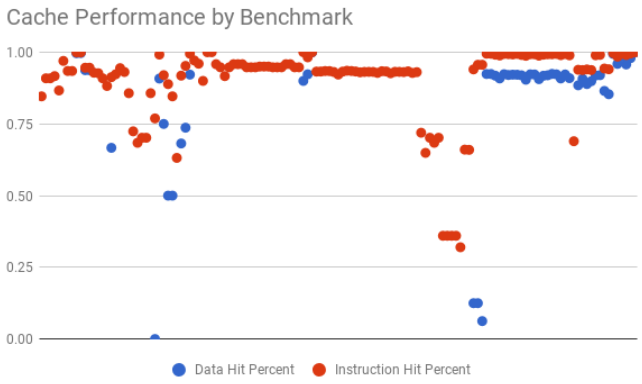

Fig. 6. Cache performance by benchmark

VGA interface clock (25Mhz). We do not believe this is a fundamental limitation of our design or the Vivado toolchain, but since we were achieving sufficient real time performance at 50Mhz, we decided to focus our efforts on other aspects of the design.

The IPC we saw in most of our benchmarks (See Figure 3) was lower than our initial estimates predicted (predicted 0.7, actual overall was 0.47). When collecting real-time performance counters on our demo binaries we saw IPC values closer to 0.6 instructions per cycle which is more consistent with our expectations.

One factor that contributes to low instruction per cycle metrics is that our processor has a direct-mapped cache. Initially, we expected to implement a set associative cache; however, we noticed surprisingly high hit rates for most of our benchmarks and demo kernels. The biggest benefit we saw when adding the cache was its addition — the size of the cache wasn't nearly as important as just having one. This allowed us to prioritize other aspects of the design instead of increasing the cache size; however, some benchmarks exhibited pathological behavior which would drive down instruction per cycle metrics (see Figure 6).

V. System Description

The top level interface for our processor is the "Chip" interface. It acts as a wrapper around the memory subsystem, the core, the DRAM controller, the VGA controller, the keyboard controller, and the timer controller. Figure 7 is a depiction of the datapath.

The processor uses three different clock domains. The external VGA interface is running at 25Mhz. Our VGA controller operates four times as fast as the VGA interface, so video memory is clocked at 100Mhz. The main clock in the system which drives the core is 50Mhz.

A. Memory Subsystem

Within the top level "Chip" interface we have the memory subsystem shown in Figure 8. The primary responsibility of the top level memory subsystem is to route requests to the appropriate controller (either video memory or main memory).

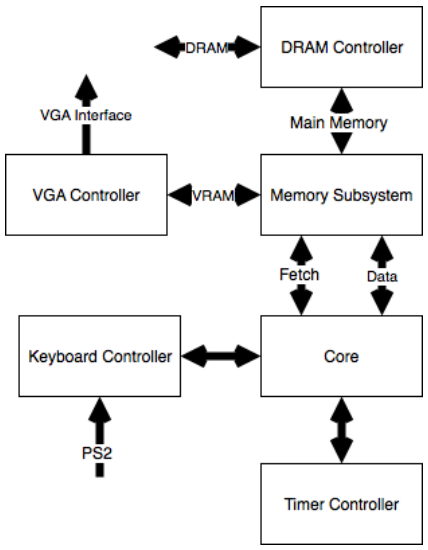

Fig. 7. Top level "Chip" interface for the processor

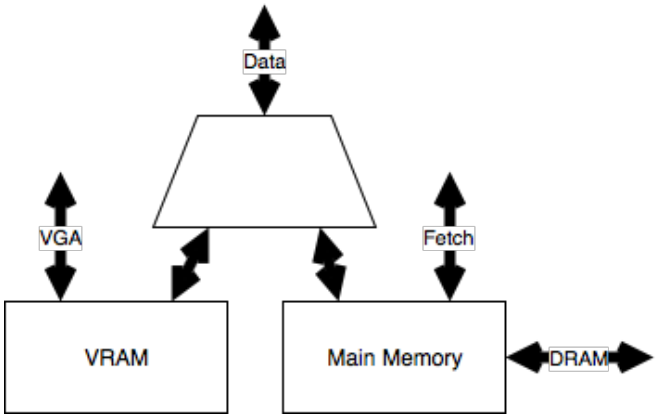

Fig. 8. Top level memory controller

Both video memory and the cache are dual ported, so we do not allow the VGA interface to access main memory and we forbid instruction fetch from video memory. Standard load/store instructions can operate on both video memory and main memory.

Our memory controller only supports a single pending operation on each of the three ports (VGA, Data, Fetch). Initially, we didn't consider this to be a limitation; however, when we began to synthesize the entire design we realized that the critical path in the design was routing the signals from the cache to the rest of the microarchitectural pipeline. After optimizing the microarchitectural pipeline, we reduced the critical path down to the clock-to-out propagation delay within the memory controller which was longer than the 10ns period of a 100Mhz clock. As mentioned in the section on Design Trade Studies, the Vivado toolchain forced us to use only power of two clock multiples of our slowest clock which meant that our clock frequency choices were 100MHz or 50MHz. Luckily, our specification was able to tolerate such a large drop in clock frequency, so we were able to execute at 50MHz. Given additional time, we would have liked to re-write the memory subsystem to supported multi-issue with internal pipelining.

The "Main Memory" module is the interface to our caching infrastructure. A stylized data path is shown in Figure 9. The Translation (TLB) block in the datapath is acting as a pass through component currently. We would like to extend it's capabilities to enable virtual memory translation in the future.

In the event of a cache miss, both the fetch port and the data port of the memory module release control of their cache port to the DRAM control FSM. While not strictly necessary for correctness, this simplifies the design while providing minimal overhead since the core's pipeline will stall on the incomplete memory operation anyway.

The cache is direct mapped with a cache line size of 64 bytes and a total cache size of 4KB. The cache has a single cycle access on a cache hit.

B. Core

Figure 10 depicts the microarchitectural pipeline of the core. Originally we predicted that we would need to split instruction fetch into two stages to meet timing (and thus have a 6-stage pipeline). However, when we began synthesizing the entire chip we realized that we were not going to be able to meet the timing constraints with only 6 stages.

First, we split the execute stage into two stages. During the first stage the ALU performs the computation, and during the second stage the branch is resolved (taking a branch is based on the result from the ALU).

In addition, we also split the memory stage into two stages. The first stage issues memory operations for load/store instructions, and the second stage waits until the memory operation is complete.

We will refer to the pipeline stages as IF1, IF2 (Instruction Fetch 1, 2), ID (Instruction Decode), EX1, EX2 (Execute 1, 2), MEM1, MEM2 (Memory 1, 2), and WB (Write Back).

In the datapath from Figure 10 the Control Flow module is responsible for resolving stall requests, mispredictions (flush requests), and interrupts (drain requests). All control flow requests are prioritized by age of the instruction. The oldest instruction's request is always satisfied first and younger instructions must obey that request before generating their own requests.

ID will generate a stall request whenever there is a data dependency on a register that cannot be resolved with data forwarding. We forward register values from the end of EX1, EX2, MEM1, MEM2, and WB to the beginning of the ID if the result has been computed. For example, a load instruction in EX1 cannot be forwarded because the value is not computed until MEM2. In contrast, an add instruction in EX2 can be forwarded because the value was computed during EX1.

Both memory stages are also able to generate a stall request. MEM1 will generate a stall if the instruction in MEM1 is a memory operation, but the memory controller is not ready to accept a new request. MEM2 will generate a stall if the

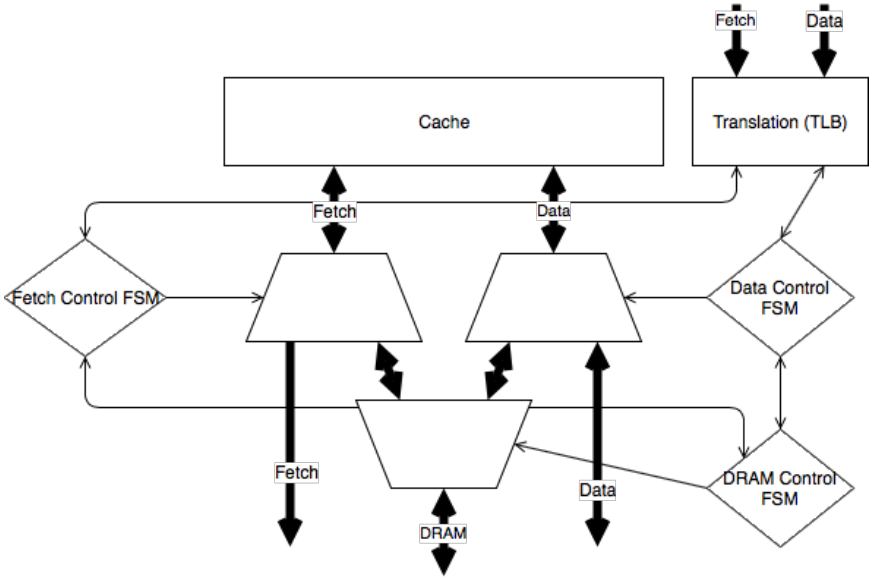

Fig. 9. Datapath for the Main Memory module used to generate DRAM requests and interface with the cache

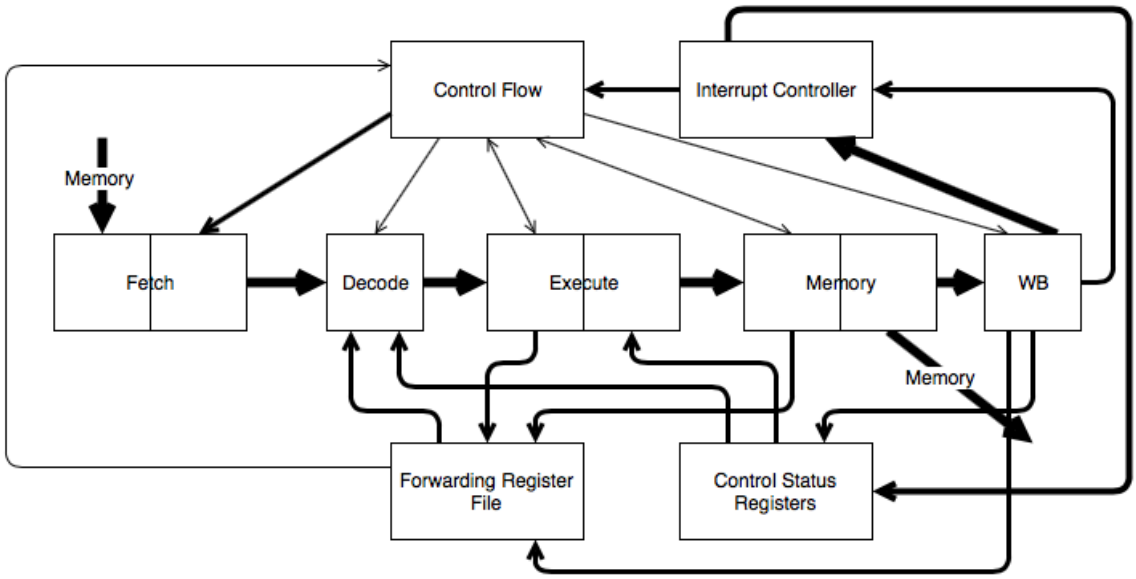

Fig. 10. Microachitectural pipeline for the core (8 stages)

instruction in MEM2 is a memory operation, but the response from the memory controller is not valid.

The control flow module predicts the next instruction to execute using a branch predictor with 128 entries of a 2-bit saturation counter in IF1. In addition, the control flow module is responsible for validating the predictions made and rolling back execution if necessary once the next instruction target is resolved in EX2. If the next instruction to execute was predicted incorrectly, we will flush all pipeline stages with instructions younger than the control flow operation and update the program counter value in IF1.

If there is a pending interrupt, IF1 will receive a flush request. This will prevent new instructions from entering the pipeline. Once the pipeline has been drained the program counter in IF1 will be updated to the base of the interrupt vector configured by the user and execution will proceed from there[5].

Since SYSTEM instructions[6] are not common, we do not forward partially computed state between those instructions like we do for registers. Instead, we require that there can only be a single SYSTEM instruction in the microarchitectural

[5]The correct cause, value, and error program counter will be set in the control status registers as appropriate.

[6]These include control status register instructions, and MRET.

pipeline at a time. This is enforced by generating a stall request whenever a SYSTEM instruction is in ID while another SYSTEM instruction is downstream in the pipeline.

Performance counters are updated when an instruction retires in the write back stage.

VI. Project Management

Table II shows both our proposed timeline[7] and what actually happened based on our weekly blogs posts. The biggest takeaway is we severely underestimated how difficult integrating the different Zedboard components would be.

We also did not anticipate how much effort it would take to synthesis the entire chip as an integrated unit. Our initial schedule included synthesis time for the individual components as they were being developed, but once we were ready to integrate them we encountered new challenges due to routing the entire integrated design on the Zynq-7020 fabric.

A. Team Member Responsibilities

Table II includes the per week responsibilities. As a high level summary, Reid was primarily responsible for internal interfaces and Teguh was primarily responsible for external interfaces. We shared secondary responsibilities for software and synthesis/integration.

B. Budget

Since we were creating a soft core, our budget requirements were fairly minimal. We did not expect the core to take a substantial part of the FPGA resources[8], so we picked our board based on the peripheral support available and the toolchain used for synthesis. We settled on the ZedBoard due to its VGA output, as well as a plentiful amount of Pmod connections which we used to connect the PS/2 keyboard. We also originally intended to connect a secondary SD card to another Pmod connection for bootloading purposes, but decided on using the Arm core on the Programmable System part of the Zynq-7020 to preload memory before our processor begins executing instructions. We bought all these parts at the beginning of the semester, and the scope of our project did not change in any way that required us to acquire more parts.

C. Risk Management

The biggest risks we anticipated in our project were primarily I/O related. All the work that we have done in related coursework, particularly 18-447[9], assumed that these interfaces were neatly abstracted away, or omitted them as out-of-scope for the course. Unfortunately for us, without I/O our processor would be unable to demonstrate that it functions as desired. We decided to mitigate these risks by working on I/O early in the semester. We started by interfacing to external DRAM, and then added in VGA and PS/2 support.

[7]The proposed schedule starting on 4/2 is the adjusted schedule from the midpoint demo

[8]Our final utilization was 13% LUTs, 15% FFs, and 3% BRAMs, so this prediction was correct.

[9]Introduction to Computer Architecture

We originally budgeted a few weeks for the external interfaces at the beginning of the semester, but we quickly found a few weeks to be insufficient due to unfamiliarity with the tools and documentation that was not consistent with reality. One advantage we had when mitigating this delay is that both of us are equally comfortable working with all parts of the project. When one of person became stuck, we could take a step back and have the other person step in with a fresh set of eyes.

Our original project as scoped required 256MB of addressable memory, which was too large to fit on the FPGA fabric. Because of this, we did not have room for fallback in our memory module. In hindsight, we might have been able to implement a smaller working memory on the fabric initially while bringing up the rest of the processor, and then transition over to using the external DRAM once we found it to be absolutely necessary. On the other hand, this forced us to get our cache infrastructure to a good point early on, as the latency for hitting DRAM is much higher than on chip block RAMs.

For the VGA controller, we implemented this in a few stages, where certain features were disabled but would still be functional. We started with single bits controlling each pixel in the VGA output, then implemented text mode in the VGA controller, and finally added support for colors. For the keyboard, we initially started by outputting set 1 scan codes to the CPU by having a set of dedicated addressable memory that the processor would poll to get input, and as we fleshed out our interrupt system we converted the keyboard to delivering ASCII characters via interrupts. We chose to deliver ASCII instead of scancodes to simplify the software we had to run on the processor, as we didn't need extensive scancode support for any of our applications.

VII. Related Work

The main thing we wanted to find that others had done was interfacing with I/O, so we could hopefully integrate their code into our system and spend less time struggling to get our own interfaces working. lowRISC [5] is a similar project that we looked at to see if we could pull some code, and we ended up taking (and modifying) their PS/2 controller which they in turn took from opencores. We also tried to use an AXI controller from Brian Swetland's zynq-sandbox [6] but were unable to get it to function as intended so we wrote our own instead.

The architectural simulator we used was extended from the RISC-V simulator we had to implement as Lab 0 in 18-447. Parts of our core were originally based on the pipeline we created in 18-447, but by the end of the project, only the register file remained from the original 18-447 core.

There are other projects that we were aware of that implement soft cores [7], [8], but they had larger scope which was incompatible with our single semester time frame. We also know that a past group had tried a similar project in 18-545 with the MIPS ISA instead of RISC-V, but supposedly failed to meet their goal.

TABLE II
TIMELINE

Week	Reid		Teguh	
	Proposed	**Actual**	**Proposed**	**Actual**
2/12	Architectural Simulator	Architectural Simulator	VGA Prototype	VGA Prototype
2/19	Memory Subsystem	Memory Subsystem	DRAM Interface	DRAM Interface
2/26	CPU Core	Memory Subsystem	DRAM Interface	DRAM Interface
3/5	CPU Core	DRAM Interface	DRAM Interface	DRAM Interface
3/12	PS2 Interface	DRAM Interface	Boot Loader	DRAM Interface
3/19	Kernel Drivers	DRAM/Memory Integration	Boot Loader	VGA Interface
3/26	Minesweeper	Memory Validation	PIC	VGA Integration
4/2	Core	Core	Core	Core
4/9	Boot Loader	Core/Synthesis	PIC	Synthesis/PS2 Prototype
4/16	Microkernels	Synthesis/Infrastructure	PS2	Synthesis
4/23	Microkernels	Microkernels	Microkernels	PS2 Prototype
4/30	Presentation	Performance Counters Presentation	Presentation	PS2 Interface VGA Color Tetris
5/7	Demo	Kernel Drivers Minesweeper PIC/Timer	Demo	VGA Interface PS2 Interface Tetris

VIII. SUMMARY

In total, our project was amazingly successful. We were able to deliver all of the correctness features we desired and execute arbitrary programs with billions of dynamic instructions when running on the real hardware. As with all projects, there were features we wished we would have had more time to implement (see "Future Work"), but all of the key features that make a computer a computer where present in our system.

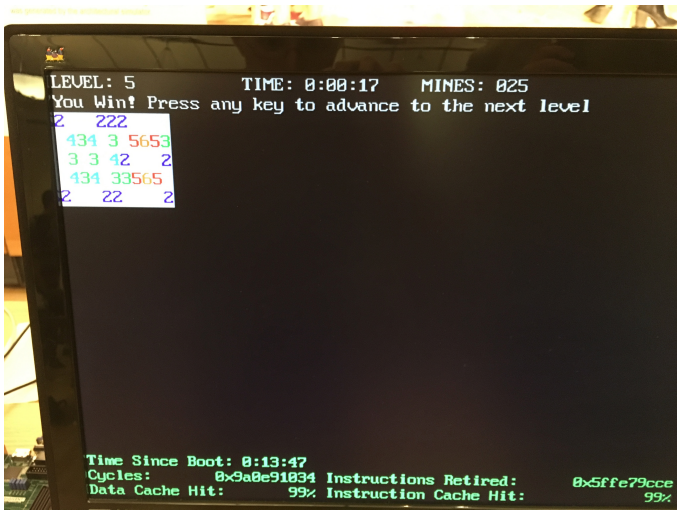

Fig. 11. Minesweeper running during demo, May 11, 2018

For most groups, the primary contribution to society is the final demo. While our demo is exciting[10], we don't expect average engineers to synthesize our soft-core on their own Zedboard. Our contribution to society mostly involves the infrastructure and validation techniques we implemented and discovered throughout the process. Our scripting infrastructure for Vivado could save future students innumerable

[10]For some definition of the word exciting

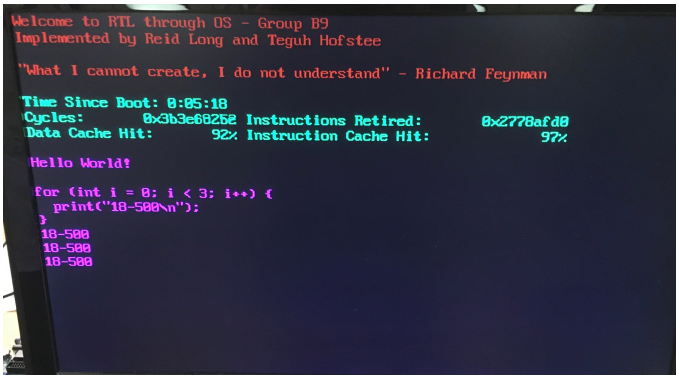

Fig. 12. Simple keyboard echo-loop running during demo, May 11, 2018

hours wasted poking at the Vivado GUI, and our verification infrastructure could be applied to other architecture projects, perhaps even outside of the academia. Futhermore, we have build robust interfaces for common Zedboard I/O (e.g. DRAM) which could save future students weeks of effort.

Additionally we gained insight as to why processors are designed the way they are today, and collided head on with performance and integration problems that are not commonly discussed.

A. Future Work

While we were able to hit all of our (adjusted) goals, there is still substantial room to improve our final project. After taking a brief recovery period[11], we plan on continuing to develop both the processor and our software suite. The biggest action item is to finish implementing the virtual memory system (including writing the tests). We considered implementing this the week before the demo, but instead prioritized polishing

[11]We've been working almost continuously since January, so it will be nice to have a day or two off.

the features we already had and writing better demo kernels[12] One of out biggest takeaways from the project is that a memory subsystem should be multi-issue and feature internal pipelining in order to make it easier to meet timing on the design. Reimplementing the memory system should be easier the second time since we already have a robust test suite, so we will also pursue this as an optimization to increase the clock frequency of our design. Finally, we would like to improve the stability of the VGA system.

B. Lessons Learned

If you are considering pursuing a similar project in the future, there are several things you should know. The first is that building a solid, well tested base before adding features is critical. We spent over four months before we had the system assembled and able to generated a distorted "hello world" message; that was time well spent. Once we achieved that milestone, we were able to rapidly iterate on arbitrary features without introducing new bugs since we had both a solid implementation to build off of and a robust verification suite to ensure that new bugs were identified before they became the foundation for future work.

The second lesson is that the memory controller should be multi-issue. One of our largest challenges was meeting timing on the Zedboard. After multiple iterations of optimizations within the pipeline, we were able to identify the critical path as internal "clock-to-out" propagation delay within the memory controller. During the initial development period several months before we were synthesizing the full chip, we prototyped memory controller logic and determined that this critical path would not be an issue; however, when adding the rest of the microarchitecture to the project, Vivado was unable to route our signals and still meet timing. Having a multi-issue memory controller would have allowed us to pipeline internally and likely achieve faster clock frequencies.

This leads into the third lesson: prototyping is important, but be careful about committing to design decisions based on prototypes. While having a collection of prototypes is important for both grades (having something to show at midpoint demos) and being able to build a robust system, we trusted the results of our prototyping too much. Subsystem prototypes by design do not include the complexity inherent in the full system which means the synthesis tools are unable to account for the logic that will be present in the full system.

The fourth lesson is that both software and hardware are important for an amazing demo. Designing hardware is an exercise in managing complexity. We spent the majority of our time working on developing and refining the datapaths, control finite state machines, and developing robust testing infrastructure to validate our processor. We kept procrastinating on writing an exciting set of programs to show off to the public. This led to our demo software being less polished and less flashy that it could have been if we spent more time on it. Having a working, well validated product is important, but it is also nice to have a cool demo. The project schedule should include time for writing all of the software necessary for a cool demo.

The fifth lesson is that the Vivado GUI does not work for large scale projects. We highly recommend groups invest in TCL scripting for setting up a Vivado project and running synthesis.

As a final caution to future students, we advise you to think carefully before pursuing a similar project. It will require a substantial investment in energy and time, but if you are willing to accept the challenge and succeed the final result will be well worth the investment. The proudest moment of my[13] life was on April 23, 2018, when our processor finally said "hello world!" after 4 months of empty struggling.

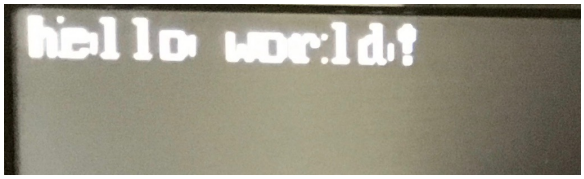

Fig. 13. April 24, 2018

SOURCE CODE

Our project includes work that could potentially be used by other students in violation of academic integrity for some courses. We are open to sharing our work with anybody who is not currently taking, or planning on taking 18-447 at Carnegie Mellon University. Please contact the authors for access to the repository. In particular, we think future students in 18-500 working on similar projects would benefit from our DRAM interface and our scripting infrastructure to interface with Vivado.

REFERENCES

[1] James Mickens, "The Night Watch," November 2013 https://www.usenix.org/system/files/1311_05-08_mickens.pdf
[2] http://www.ece.cmu.edu/ ece500/spring18/teamB09/website/
[3] https://riscv.org/specifications/
[4] https://riscv.org/specifications/privileged-isa/
[5] http://www.lowrisc.org/
[6] https://github.com/swetland/zynq-sandbox
[7] http://zipcpu.com/
[8] https://github.com/freechipsproject/rocket-chip
[9] Reid Long, Teguh Hofstee "RTL through OS: Design Proposal," March 8, 2018 http://www.ece.cmu.edu/ ece500/spring18/teamB09/website/DesignProposal.pdf

[12] A demo kernel is in contrast to our benchmark kernels which do not include a nice GUI, but are more stressful on the processor.

[13] Reid Long