RTL through OS

Reid Long, Teguh Hofstee
Group B9
March 8th 2018

Project Description

Our mission is to demonstrate mastery of computer engineering by designing and implementing a single core RISC-V processor and developing a basic multitasking kernel to run on the processor. We want to experience the thrills of debugging a complex system with neither a ground truth software implementation nor a ground truth hardware implementation. Furthermore, we want to develop a unique skill set that spans both kernels and processors in order to increase our value in the event of nuclear holocaust or zombie apocalypse[1].

Design Requirements

A full specification of the processor and the kernel can be found on the team website[2]. Relevant documents are linked in the references section. Any requirements listed formally in this section supercede any explicit or implicit requirements from the RISC-V User-Level ISA, the RISC-V Priviledged architecture, or the Pebbles Kernel specification.

Processor Requirements

- P1: The processor shall support a custom subset of the RV32IMAS instruction set architecture
    - P1.1: The processor shall be consistent with the RISC-V User-Level ISA v2.2
    - P1.2: The processor shall be consistent with the RISC-V Priviledged Architecture v1.10
    - P1.3: If the processor receives an EBREAK, it shall halt execution
        - P1.3.1: If the processor receives user stimulus (push button) it should resume execution.
    - P1.4: The processor should only implement Load Reserve/Store Conditional instructions out of the Atomic extension
    - P1.5: The processor shall support U-Mode and M-Mode
    - P1.6: The processor shall implement virtual memory translation as a 2-level hardware walked page table consistent with the specification in RISC-V Privileged Architecture specification
- P2: The processor should support debug logs via the Zedboard ARM chip
- P3: The processor shall load the kernel RAMdisk from an SD Card
    - P3.1: The processor shall idle the core until the kernel RAMdisk is loaded into main memory
- P4: The processor shall support reading and writing VRAM via load/store instructions
    - P4.1: The processor shall only support direct-mapped VRAM operations
- P5. The processor shall support fetching instructions from main memory
    - P5.1: The processor should not support fetching instructions from VRAM

[1] https://www.usenix.org/system/files/1311_05-08_mickens.pdf
[2] http://www.ece.cmu.edu/~ece500/spring18/teamB09/website/

- P6: The processor shall generate a Load/Store Fault if the software requests a misaligned load/store
- P7: The processor shall implement a subset of the Control Status Registers[3]
    - P7.1: The processor shall implement **mcycle** which will report the number of cycles since boot
    - P7.2: The processor shall implement **minstret** which will report the number of instructions retired
    - P7.3: The processor shall implement **ustatus** and **mstatus** which will report control flags[4] like interrupts enabled/disabled
    - P7.4: The processor shall implement **mtvec** which will be the base address where all traps/exceptions/interrupts will be delivered
- P8: The processor should implement several performance counters
    - P8.1: The processor should implement an instructions fetched performance counter
    - P8.2: The processor should implement performance counters for Forward/Backward branches retired/predicted correctly/predicted incorrectly for a total of six performance counters
    - P8.3: The processor should implement performance counters for data/instruction cache hits/misses for a total of four performance counters
    - P8.4: The processor should implement performance counters for Store Conditional successes and rejections for a total of two performance counters
    - P8.5: The processor should implement exception/interrupt delivering performance counters distinguishing between keyboard interrupts, exceptions, and timer interrupts delivered in M-Mode and U-Mode for a total of six performance counters
    - P8.6: The processor should implement performance counters for forward/backward jumps that are retired/predicted correctly/predicted incorrectly for a total of six performance counters

Kernel Requirements

- K1: The kernel shall implement the Pebbles kernel specification (Version 9-20-17)
    - K1.1: The kernel shall make the appropriate corrections to ensure consistency with the RISC-V architectural model
    - K1.2: The ureg_t struct shall be defined to include the architectural state of a RISC-V processor[5]
    - K1.3: The kernel should implement the task_vanish system call[6]

---

[3] See http://www.ece.cmu.edu/~ece500/spring18/teamB09/website/Architecture.pdf for additional specifications

[4] See RISC-V Privileged Architecture specification

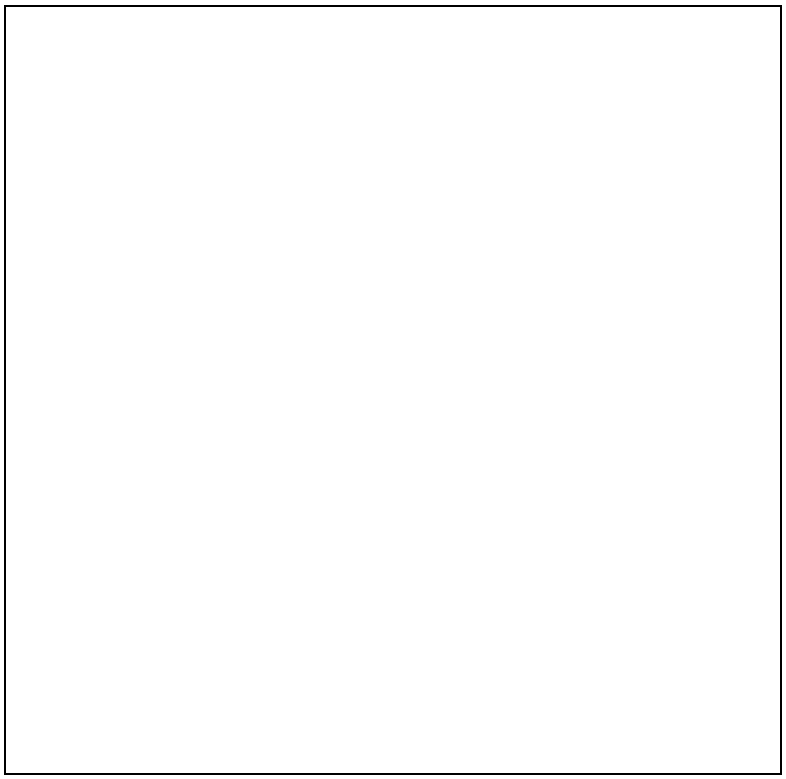
[5] The Pebbles specification is x86 centric and the ureg_t struct includes explicit references to x86 architectural state which we will not need in our system.

[6] This system call is typically optional

    - K1.4: The kernel should implement the getchar system call[7]
    - K1.5: The halt system call shall trigger an EBREAK to the processor
- K2: The kernel should boot within 30 seconds (30% tolerance[8])
- K3: The kernel shall support a 2-level hardware walked page table consistent with RISC-V Priviledged Architecture v1.10
- K4: The kernel shall support a PS2 keyboard driver
- K5: The kernel shall support a VGA console driver
    - K5.1: The kernel shall always direct-map VRAM

Functional Architecture[9]

The entire system is running on top of a ZedBoard interfacing with DRAM, SD Card, PS2, and VGA. The processor implements a custom subset of RV32IMAS which is the architectural contract with the kernel. The kernel implements the Pebbles system call spec as the application binary interface contract with the user mode programs. See Figure 1 below for a high level block diagram of the system design.



[7] This system call is typically optional
[8] The high tolerance is because there is a human in the loop. As long as the performance is somewhat reasonable, the human will consider the behavior acceptable. It is also acceptable for the processor to boot faster than is specified; however, our initial models indicate that this will be unlikely.
[9] See the System Architecture section for additional details of the subsystems

Design Constraints/Tradeoffs

Correctness

Our design is motivated by two key metrics. The primary metric is optimizing for correctness. While correctness seems like an obvious constraint, in computer architecture it is particularly critical since the complexity of a chip is many orders of magnitude larger than what we could possibly test over the course of a semester. We carefully distinguish between “true correctness” [10] with “effective correctness” which is what we are targeting. Our effective correctness metric represents correctness within the allowable operation of our processor. In particular, we are not implementing additional instructions that are not necessary to run the test binaries we have selected. Furthermore, we do not expect the processor to run for more than 20 Billion instructions between reboots (based on the simulations of a full kernel validation suite). Based on these restrictions, it is indistinguishable for our processor to contain a flaw if executing 1 Trillion instructions compared to a fully correct processor (if such a thing exists).

In order to achieve correctness we are focusing on the simplest processor possible and the simplest kernel possible. Unlike modern processors, we are targeting a simple, single core, in order, single issue processor. However, we are introducing some complexity into the design in order to improve performance (our second key metric, see below). Instead of implementing a single cycle processor, we are going to design a pipelined microarchitecture with 6 stages. A more standard microarchitecture would be a 5-stage pipeline (IF, ID, EX, MEM, WB); however, our preliminary work on the memory subsystem revealed that the clock-to-out propagation delay on reading from memory is fairly large[11].

Based on our experiences in 18-447 working on a simple RISC-V processor with a similar microarchitecture to what we are planning on building, the decode stage is on the critical path. It follows that introducing additional propagation delay in the clock-to-out part of the instruction fetch stage will be problematic and likely the critical path in our design since this propagation delay will carry over into the decode stage. This has motivated our design to feature a 6-stage pipeline (IF1, IF2, ID, EX, MEM, WB). We are accepting an increase in complexity to hopefully ensure that we will be able to deliver a processor running at 100MHz (the native frequency of the Zedboard Programmable logic). When synthesizing the full chip it is possible that our assumptions about the decode stage or the instruction fetch stage will need to be adjusted. Our future decisions will be motivated by the same factors as this initial design decision (desire to manage complexity while also ensuring reasonable performance).

---

[10] Industry grade, consistent with the chips shipped by Apple or Intel

[11] The key contributors to this clock-to-out propagation delay involve the multiple levels of comparators necessary to validate the tag bits are a match to identify which of the cache sets is the hit.

Performance

Our secondary metric is performance. Our performance constraint is based on the user experience. If it takes many minutes to boot the kernel, then the user is going to give up and assume the kernel is broken or unuseable. When the user is interacting with the kernel by playing a game or simply typing on a shell, the user expects a certain level of responsiveness. We would like to run our kernel games at 60 Frames per second. We also have identified a reasonable time-to-boot of 30 seconds. While 30 seconds is a long time by modern standards, we believe a user is willing to tolerate 30 seconds of boot time.

Since the number of instructions that need to be executed is relatively constant, we must optimize the processor in order to archive a sufficiently high IPC[12] and a sufficiently fast clock period if we want to meet our performance targets. Our calculations are summarized in table 1 below.

Binary	Instruction Count	IPC	Clock Period (ns)	Runtime (s)
Tetris (60 frames)	792,000	0.70	240	0.271
Tetris (60 frames)	792,000	0.70	10	0.011
Kernel Boot	900,000,000	0.70	240	308 (5 minutes)
Kernel Boot	900,000,000	0.70	10	12.857
Full Kernel Test	15,000,000,000	0.70	240	5142 (85 minutes)
Full Kernel Test	15,000,000,000	0.70	10	214 (3.5 minutes)

Table 1: Performance estimates

Our estimates for IPC are based on pessimistic results from 18-447 RV32I processors. These results are not necessarily consistent with the microarchitecture we are planning on designing; however, as far as we can tell 18-447 processors are the most similar to our desired microarchitecture. Our microarchitecture will be different than standard 18-447 processors since we are implementing a many more features. Furthermore, we will experience cache misses which will have a negative effect on our IPC. As we progress through the project we will update these calculations based on our measurements.

We provide two estimates for clock period to reveal the likely extremes we will be able to design within. A clock period of 240ns represents the memory latency of DRAM on the Zedboard. A

[12] Instructions per cycle

clock period of 10ns represents the “native” frequency of the Zedboard’s programmable logic. Ideally we will archive the 10ns clock period; however, it is possible that our design’s critical path after place and route will be too long and will limit us to a slower clock period.

Evaluation

Our evaluation will focus primarily on ensuring the correctness of our design (the primary metric). Since our performance evaluation is subjective and based on human interaction we will evaluate that indirectly by measuring runtime of the correctness evaluation of both the kernel and the processor.

Validating a kernel and a processor simultaneously introduces substantial complexity into our testing methodology. In contrast to most other projects (both within the course and in the real world), we do not have a ground truth processor nor a ground truth kernel. Furthermore, neither a ground truth processor nor a ground truth kernel exist[13] for the specific subset of the RISC-V architecture we are attempting to implement.

Processor Validation

Our processor validation will focus on both randomized coverage tests[14] and targeted edge case tests. Our goal is to achieve 100 million lines of assembly in the randomized tests and to have 100 targeted edge case test programs to cover the 70 instructions we are implementing. In order to enable our large scale validation efforts without hindering other project progress we will implement a architectural simulator which will be able to generate golden register dumps. We will be able to simply write the test program and run it through the architectural simulator to generate the golden register dumps. Then we can execute the test on our processor and compare the results to the golden register dump to identify a test passing or failing. Since validating the entire memory image would require an excessive amount of storage[15] we will instead perform a checksum of the relevant[16] memory regions and storing the result in a register.

Kernel Validation

We have access to 150 correctness binaries for a Pebbles specification, x86 kernel. We will port these binaries to our custom RV32IMAS architecture and then execute them on the processor. Based on our simulation estimates, running all of these binaries in succession will require executing approximately 15 billion instructions. This will severely tax our processor, but will provide a high level of confidence that both the kernel and the processor are correct (or at least effectively correct).

---

[13] This motivated our desire to build an architectural simulator for our specific architecture to help facilitate easier validation and verification.
[14] These randomized tests are generated with a test generate designed in 18-447. We will be able to extend this generator to be compliant with the RV32IMAS architecture our processor is implementing.
[15] This would require 512MB of “golden” memory for each test program
[16] Defined by the test based on what regions of memory the test is attempting to modify

System Architecture

Processor Architecture

We are implementing a single core processor which implements a custom subset of the RV32IMAS specification (User Mode v2.2, Privilege Mode v1.10).

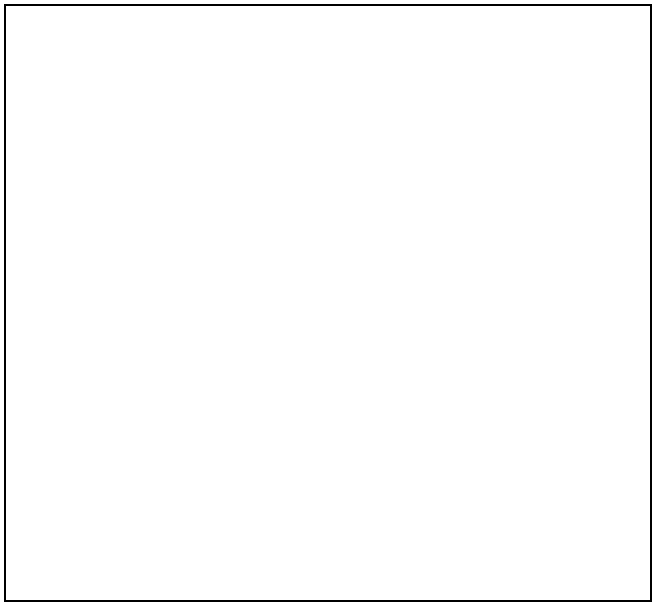


Figure 2 above shows a component breakdown of the processor high level architecture. Our key contributions of custom RTL will be in the Memory subsystem, the Core, and the Programmable Interrupt Controller. In addition, we are planning on using a custom VGA controller[17] and a custom timer device[18]. We would like to use and off-the-shelf IP component for interfacing with the DRAM, interfacing with the PS2 keyboard, and for interfacing with the SD card; however, if we are unable to find and integrate a suitable component, we will resort to writing custom RTL to interact with either/both.

Memory Management Unit Architecture

The memory subsystem features a 32-bit virtual address space with 512MB of physically address DRAM. Virtual address translations are provided with a classical 2-Level hardware walked page table. There is a single level translation lookaside buffer to amortize the cost of virtual memory translation. All addresses on the core are virtual with translation happening in the memory subsystem which interfaces directly with DRAM when necessary.

[17]The off-the-shelf components we were able to find were overly complex and we have experience from 18-240 implementing our own VGA drivers

[18] This should be trivial to implement with a simple counter and comparator

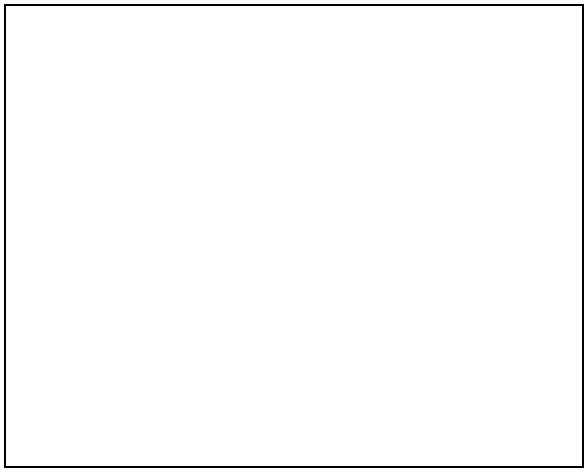


Figure 3 above, shows a high level outline of the memory subsystem. One notable feature is that instruction fetch cannot access VRAM and the VGA driver will not be able to access MainMemory. No reasonable program would need to fetch instructions out of VRAM, and this restriction enables many simplifications in the memory controller. In particular, having only two sources of reading/writing to both of our VRAM and MainMemory ensures that we can use the native dual-ported BRAMs on the Zedboard for both the VRAM and our Cache.

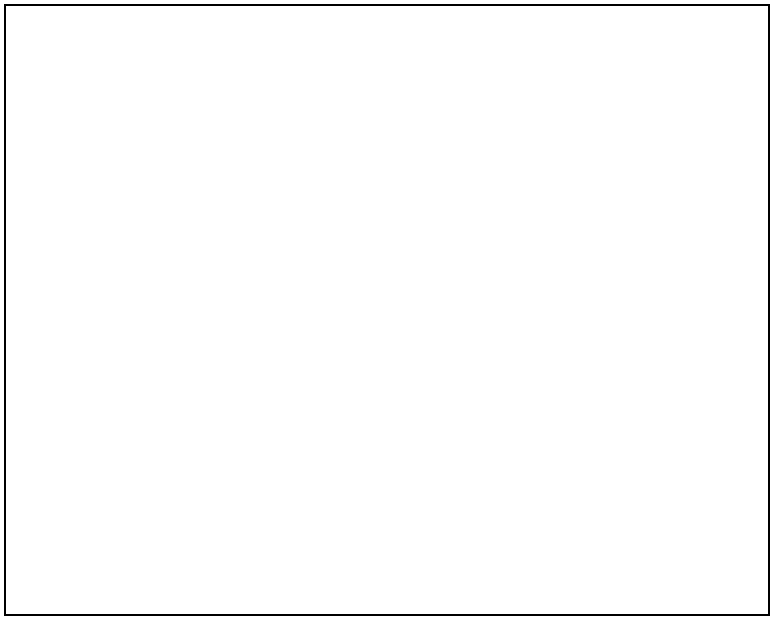


Figure 4 above shows the high level architecture of the MainMemory module within the memory hierarchy. This diagram elides many of the datapath level details of our implementation; however, it captures the essence of the design. We have three controllers within MainMemory. The Instruction Fetch and Load/Store controllers can operate in parallel fulfilling memory requests for their respective stages in the pipeline; however, in the event of a cache miss the Instruction Fetch or Load/Store controller will trigger the DRAM controller to take over and update the cache.

In order to enable virtual address translation to happen in parallel with reading from the cache, we are using a virtual indexed[19], physically tagged cache. This ensures that we can deliver a single-cycle memory read on a cache hit.

Another detail this diagram reveals is that our virtual memory translations go through the cache. This means that we do not need to perform a cache flush when configuring page tables since the translation will find the most recently written values in the cache (or will trigger a cache miss and load the cache with the appropriate data from main memory).

Core Architecture

As discussed in our design tradeoffs, the core will feature a 6-stage pipeline. When the design is more mature, we are willing to adjust based on the timing constraints of the full chip, but our initial estimates indicate that the critical path will be through the decode stage. Thus it is unacceptable to introduce additional clock-to-out propagation delay in the instruction fetch stage. We have resolved this issue by adding an additional pipeline stage to instruction fetch to reduce the likelihood of a timing violation in the decode stage.

We observed that the write back stage is very simple and likely will have a short critical path. This has enabled us to treat the writeback stage as the MEM2 stage analog of the IF2 stage since it is unlikely the write back stage will be the critical path, our initial estimates indicate that it will be permissible for our microarchitecture to push additional clock-to-out propagation delay from the MEM stage into the WB stage.

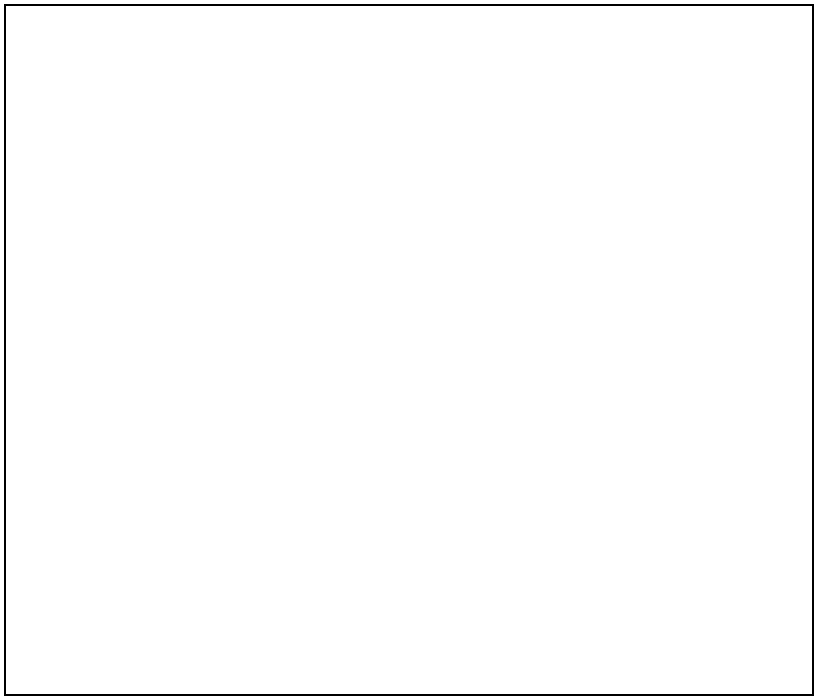
We also will implement a simple branch predictor and data forwarding within the pipeline in order to achieve an acceptable IPC which is necessary to meet our performance goals (discussed above in design tradeoffs).

Kernel Architecture

Our kernel shall implement the Pebbles System Call specification. This specification contains a mere 25 system calls; however, those system calls include Tasks[20], Threads, Memory Management[21], and console input/output. Figure 6 below provides a conceptual software stack for the kernel where communication flows in the vertical direction. Explicit communication arrows have been elided in this model since the true communication graph in a kernel reveals

[19] Technically the cache is physically indexed, but the index is from the virtual page offset which is equivalent to the physical page offset. The translation is the identity operation.
[20] Analogous to a Process
[21] Necessary to implement user-mode malloc and automated stack growth

that most components are able to talk directly to each other, but not in a conceptually meaningful way.



Project Management

Timeline

One of the strengths of our team is that both members are comfortable with both the hardware and the software aspects of the project. This enables us to have extreme flexibility when resolving issues since we can easily swap roles to provide a fresh set of eyes on a complex problem. This timeline includes expected assignments of who will be working on which aspects of the project; however, we are willing to make responsibility adjusts as we discover which aspects of the project are the most complex and thus need more time than initially anticipated.

The following schedule (Table 2) is color coded to indicate which aspect of the project the specific task associates with. Blues represent software while oranges/reds represent hardware.

Week	External Milestone	Internal Milestone	Reid's Task	Teguh's Task
2/12			Architectural Simulator	VGA Interface
2/19			Memory Management Unit	DRAM Interface

2/26	Design Presentations		CPU Core	DRAM Interface
3/5	Design Proposal		CPU Core	CPU Core
3/12	Spring Break		PS2 Interface	Boot Loader
3/19		Working Processor	Kernel Drivers	Boot Loader
3/26		Working Demo	Minesweeper	PIC
4/2	Midpoint Presentation		Virtual Memory/TLB	Tetris
4/9			Kernel	Kernel
4/16	Carnival			
4/23		Working Kernel	Kernel	Kernel
4/30	Final Presentation		Demo	Demo

Table 2: Gantt Chart

Budget

Our project is extremely low-cost since many of the components are being borrowed from lab at no cost to our budget. Table 3 provides the bill of materials for the entire project.

Table 3: Bill of Materials

Item	Price	Source
Zedboard	$0	Lab
PS2 Keyboard	$0	Lab
VGA Monitor	$0	Lab/Personal
PS2 PMOD	$8.99	Digilent
PS2 SD Card Reader	$9.99	Digilent
SD Card	$0	Lab
Total	$18.99	

Risk

One of the biggest risks in our project has already created a severe problem for our timeline. In particular, the DRAM interface is proving exceptionally difficult to integrate into our memory management unit. While our initial estimates and intuition assumed it should be a fairly common to communicate with DRAM from the FPGA logic, we have been unable to find a working tutorial or achieve any sort of communication with the DRAM. We are actively mitigating the risk and researching additional options to increase our flexibility.

Other high risk aspects of the project include all other external interfaces. These include VGA, PS2, and SD in addition to the DRAM interface. Our primary risk mitigation technique for these risks is to start integrating them into the system as soon as possible. This approach has proven invaluable for the DRAM interface since we have encountered the problem early enough in the progress that we still have flexibility to design workarounds and research additional options.

Beyond the interface concerns, we also are attempting a project with a high degree of complexity. Effectively managing this complexity is likely going to be a challenge especially since neither of us have tackled a project of this scope before[22]. In order to manage the complexity and mitigate this risk, we are planning on utilizing high quality software development techniques like pair programming, peer review, and frequent communication.

The final risk in our project is the unknown. There are many components in our system that we have never implemented before. Our past experiences indicate that one never truly understands something until one attempts to implement it. In order to mitigate this risk, we are planning on targeting the largest unknowns first. This will hopefully allow us to discover unknown complexity early in the project while we still have the ability to cleanly pivot to alternative implementations or designs.

Related Work

Depending on the lens, our project is either one of a kind or highly redundant. From a certain perspective, designing a computer and developing an Operating System are already solved problems. Anybody can buy an x86 processor from Intel and many modern mobile processors feature ARM's RISC instruction set architecture. Furthermore, there are many widely available operating systems including Linux, Windows, and macOS.

The annual releases of modern operating systems and modern processors are not yearly rewrites of the entire system. The complexity in state-of-the-art processors and operating systems exceeds what can be accomplished in a year by a team of world class experts, thus most releases patches on top of legacy code. It is not tractable for us to attempt to duplicate the work of both Intel and Microsoft have done over the last several decades in a single semester.

---

[22] We have each completed 18-447 and 15-410; however we did not complete both of those classes in the same semester.

Instead, we are targeting a more reasonable goal of a reduced operating system and a reduced subset of a RISC-V processor.

While it may appear that our project is strictly inferior to the work produced by large corporations since we are delivering fewer features, we have one key advantage. Our product is not hindered by the enormous amount of bloat that both modern operating systems and modern processors have. For some people, what we consider bloat is actually a feature. Some developers need exotic instructions or specialized system calls to deliver cutting edge performance; however, there exists some people[23] who would prefer to have a simple processor running a simple operating system. Simple technologies can make it easier to verify systems with respect to security and correctness. Considering this context, it follows that our project could easily be a prototype for a future startup company that specializes in processors and kernels that deliver superior security guarantees without sacrificing performance for most common applications[24].

References

RISC-V Specifications: https://riscv.org/specifications/
RISC-V User Level Specification:
http://www.ece.cmu.edu/~ece500/spring18/teamB09/website/riscv-spec-v2.2.pdf
RISC-V Supervisor Specification:
http://www.ece.cmu.edu/~ece500/spring18/teamB09/website/riscv-privileged-v1.10.pdf
Pebbles Kernel Specification:
http://www.ece.cmu.edu/~ece500/spring18/teamB09/website/kspec.pdf
RISC-V FPGA Size: http://ieeexplore.ieee.org/document/8056766/

[23]Several of our security conscious professors/friends have indicated they would be very receptive to a simple, secure processor, especially one without the Intel Management Engine.
[24] Less common applications would likely have a performance sacrifice