

Beryl: A 32-bit Out-of-Order ARM PC

Pete Ehrett, Brian Jacobs, Amanda Marano

December 14, 2015

This document and all associated resources, code, and designs may be utilized without limitation for future instances of 18-545: Advanced Digital Design Project at Carnegie Mellon University. They may be used with permission for other academic or nonprofit research purposes. They may not be used for any purposes other than academic purposes or nonprofit research without the explicit written permission of the authors.

Permission is granted to Professors Nace and Lucia to post this report on the course website.

Contents

1	Project Overview	5
2	Toolchain	6
2.1	Vivado (and Porting a Project from ISE)	6
2.2	Perl Scripts	7
3	Core Architecture	7
3.1	Amber: an in-order ARM processor and system infrastructure	7
3.2	Beryl: out-of-order core	11
3.2.1	Fetch	12
3.2.2	Decode	12
3.2.3	Dispatch	13
3.2.4	Execute	16
3.3	Chrysoberyl: superscalar core	17
3.3.1	Fetch	17
3.3.2	Decode	17
3.3.3	Dispatch	17
3.3.4	Execute	18
3.4	Dendrite: MorphCore-like ARM core	18
3.5	ISA and Implementation Notes	19
4	Display Controller	21
4.1	HDMI	22
4.2	VGA	23
5	Keyboard	25
5.1	Hardware Interface	25
5.2	PS/2 Interrupts	26
6	Testing and Verification	27
6.1	Software	28
7	Results, Status, and Future Work	29
7.1	Project Status	29
7.2	System Performance	29
7.3	Future Work	31
8	Lessons Learned	31
9	Personal Notes	33
9.1	A Note from Pete	33
9.2	A Note from Amanda	34
9.3	A Note from Brian	35

List of Figures

1	Original Amber system diagram.	8
2	Original Amber 23 pipeline.	9
3	The modified Beryl system diagram.	11
4	Beryl fetch stage.	12
5	Beryl dispatch stage.	13
6	Beryl execute stage.	16
7	Available characters in TEXT_MODE.	22
8	A test pattern in TEXT_MODE.	23
9	LCD Interface Circuit [4]	23
10	VGA Pinout and Wire Colors	24
11	An example of an R2R DAC.	24
12	The results of our VGA efforts	25
13	PS/2 on an Oscilloscope	26
14	Relative benchmark performance of Amber and Beryl.	30
15	Relative resource utilization of Amber and Beryl.	30
16	Normalized speedup between Amber and Beryl.	31

1 Project Overview

Rather than choosing the 18-545 “default project” of implementing an 80s video game in FPGA hardware, our team decided we wanted to do something more related to modern processor architecture and reconfigurable computing.

After some initial research, we were intrigued by Khubaib et. al.’s MorphCore paper[2], which details a processor architecture capable of dynamically switching between two processing modes based on workload requirements. The first uses the processor as a single, superscalar, out-of-order core to take advantage of instruction-level parallelism in programs. The second carefully splits the processors resources to produce multiple in-order cores, providing superior performance when thread-level parallelism can be exploited. Although this processor design was very successful in simulation, the paper notes that it was not synthesized in physical hardware. Our goal was to design and synthesize a MorphCore-like processor (or at least to get as close as possible) on an FPGA and use it in a full computer system with display output and keyboard input.

From the outset, we recognized that this was an extremely ambitious goal – either one of designing and building the processor core or making a complete, reasonably modern computer system could be a semester-long project by itself. Therefore, we created a series of intermediate goals such that our project could be successful even if we didnt have sufficient time to create the full MorphCore. These were as follows:

1. Make a complete computer around a simple i.e. in-order single-core processor capable of executing arbitrary code, with UART-based I/O.
2. Extend the I/O capabilities of the system to include direct output to a standard monitor and direct keyboard input.
3. Design, implement, and verify an out-of-order processor core for the system.
4. Make this out-of-order core superscalar.
5. Modify the superscalar design to be capable of splitting processor resources into multiple cores, a la MorphCore.
6. Boot a simple OS on our MorphCore.

We divided this work into the following sub-tasks, split among our team:

1. Design or otherwise acquire a simple in-order processor core to use as a starting point.
2. Verify the existing core, whether designed ourselves or acquired elsewhere.
3. Synthesize this core to FPGA and demonstrate basic code execution.

4. Set up UART I/O for the system and demonstrate with the in-order core.
5. Design and implement a display controller to connect to the core.
6. Build an interrupt handler and hardware interface for generic keyboard input.
7. Boot a simple OS kernel on the complete in-order system.
8. Modify the in-order core design to out-of-order and implement.
9. Verify the out-of-order core.
10. Extend the out-of-order core to a superscalar design and implement.
11. Verify the superscalar out-of-order core.
12. Modify the superscalar design to enable splitting processor hardware resources into multiple cores upon keypress, software interrupt, etc. and implement.
13. Verify the MorphCore.

2 Toolchain

Our toolchain for this project used Vivado for everything from simulation to synthesis. We originally made an effort to avoid using Vivado for too much of the process in favor of tools which we were more familiar with. In particular, we wanted to use VCS for simulation. This proved to be a bad decision, although it did not cost us more than a day or two's worth of work.

2.1 Vivado (and Porting a Project from ISE)

Rapidly, it became too complicated to use VCS and Vivado for the same project. There were two major issues: the first was that they had different conventions for 'include and 'define directives which required nontrivial changes to the entire Amber core to allow both tools to properly evaluate them. The second was that the Xilinx IP that we were using for the multiplier and memory modules was not easily simulateable without Vivado. Ultimately, we determined that it was best to try to keep everything as internal to Vivado as possible to prevent cross-tool issues.

Another issue which we ran into was that the Amber project was originally built using ISE. Converting the project into something which would even simulate in Vivado involved removing a lot of the macro-based conditional code and replacing all of the IP cores, which we did not think it was worth extracting from the ISE version of the project.

Over the course of the semester, issues with Vivado made seemingly simple tasks take hours and days. These issues were compounded with the issues arising from porting the

project from one tool to another. Rather than one monolithic list of warnings and complaints about using Vivado for development, we have included such warnings during the parts of the report where they were most relevant.

Amanda and Brian did a majority of their work in the lab. This was because a lot of their work involved interacting directly with the VC707 board, and so could not be done remotely. Pete, however, was able to do a lot of work by using VNC to remotely access Vivado and work from outside the lab. This helped us multiplex the available computer resources, especially after having one of our computers fail midway through the semester.

2.2 Perl Scripts

One of the most valuable tools we ended up using was Perl, especially for producing .COE files to initialize memory. Our ARM assembler produced files in a .mem format which had to be converted into Xilinx' .COE format. Doing this took a couple of hours of Perl hacking, and left us with a tool which could let us easily convert several different file formats, including the .PPM image format, into memory layouts for our core.

This was especially useful when we were working on the HDMI display module. We wrote a Perl script which used various linux utilities to produce a set of bitmaps of characters and then map them into a .COE file. This would have taken substantially longer to do by hand, but was a few days work using a relatively simple script.

3 Core Architecture

3.1 Amber: an in-order ARM processor and system infrastructure

The Amber project[1] is a complete in-order processor design supporting the ARMv2 ISA. Its completeness and relative simplicity made it a good choice to use as a baseline for later modification and performance comparison. Its surrounding system is based on the “wishbone” bus standard, making it compatible with various other modules including UART, Ethernet, certain memory controllers, and so forth.

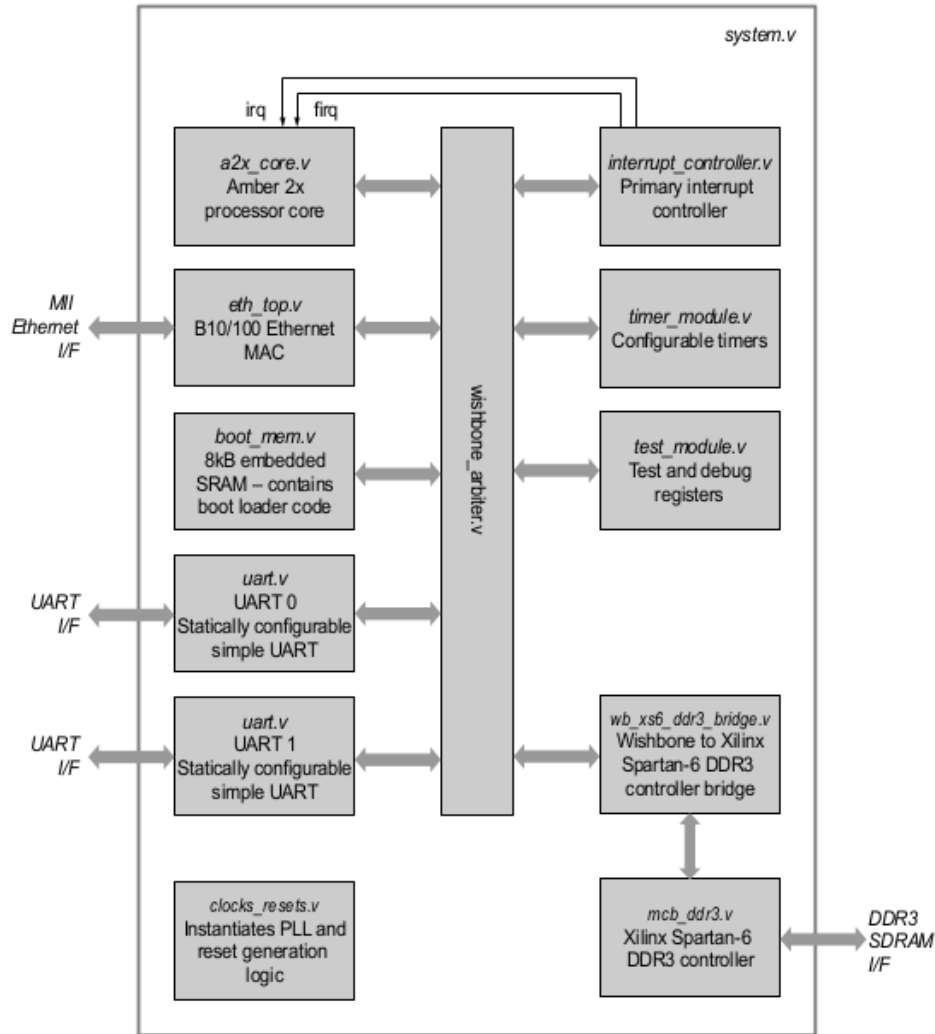


Figure 1: Original Amber system diagram.

The project actually includes two cores: Amber 23 and Amber 25. Amber 23 includes a 3-stage pipeline Fetch (with a unified instruction/data cache), Decode, and Execute and tends to be rather slow due to its inefficient handling of load and store operations. Amber 25 improves upon this by moving to a 5-stage pipeline, adding Memory and Writeback stages after Execute. It also splits the cache structure, with an instruction cache in the Fetch stage and a separate data cache in the Memory stage. Neither core has any mechanism for branch prediction.

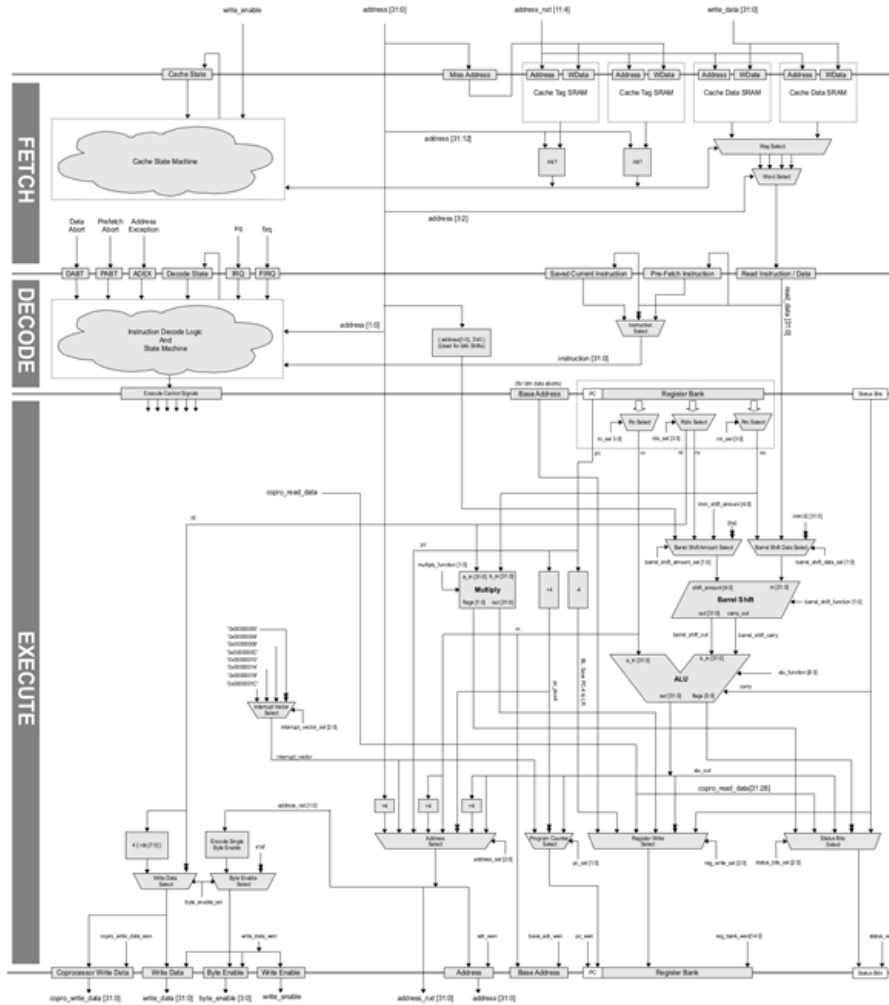


Figure 2: Original Amber 23 pipeline.

Our decision to use this core as our starting point proved to be both a blessing and a curse. On the one hand, it was useful to have a reference design to test against and modify rather than starting completely from scratch. However, as we tested and revised, we discovered a plethora of undocumented idiosyncrasies, inexplicable design decisions, ISA omissions, etc. that made us wonder if we'd have done better just to create our own core from the get-go.

First, we had an extremely difficult time synthesizing the supposedly-synthesizable Amber project to our VC707 board. The original project used a Xilinx Spartan-6 as an FPGA target, so presumably, the Verilog should have been compatible with Xilinx tools. It wasn't.

One issue which slowed us down by more than a week was the delicate network of 'include statements that held the Amber project together. To start off, they all were written without filepaths, requiring a script to be written to guess the path of a file from its name and replace the short name with the relative path in the 'include directive. This was necessary to get

VCS working, although we then discovered that Vivado could not handle relative paths, and instead needed unqualified filenames.

It then came to light that the project would only build if certain files were included in a particular order. This order was implicitly followed by ISE, but not by Vivado. In particular, some things which were ‘define’ed needed to be undefined at certain points in the build order, but not in others. We had to sift through the build order to figure out what needed to be included where to ensure that things were not defined too early.

After a few weeks of tracking down bugs in ‘ifdefs, we managed to finally get the project to simulate and synthesize in Vivado. Of course, simulating and synthesizing correctly required fixing issues with how previous Xilinx IP was included in the project, and replace it with our own.

Second, we encountered a huge number of problems related to how Amber handles multi-cycle instructions including SWP, LDM, STM, pre- and post-indexed memory operations, and so forth. Amber uses a state machine in the Decode stage to effectively convert each of these into micro-ops, but the core documentation made no reference to this part of the Amber design and the state machine itself was quite buggy. We ultimately removed this from Decode and moved it into any applicable execution units instead.

Third, we found that Amber does not support the ARM MRS and MSR instructions. These are critical for interrupt handling in a modern OS, but instead, Amber uses long-obsolete variants of the TEQ and TST instructions to replicate this functionality. It took us an unfortunately long time to recognize this problem, which caused us significant headaches when attempting to properly set up keyboard input.

Fourth, the Amber cores Decode and Execute stages perform multiplication extremely inefficiently. Upon decoding a MUL or MLA instruction, Decode enters a portion of its state machine where it waits for the multiplier unit in Execute to finish. Executes multiplier is based on the Booth multiplication algorithm and takes 34 cycles to return a result, plus another cycle to add another register value if the instruction is MLA instead of MUL. The multiplier is also non-pipelined, so the entire core must stall whenever MUL or MLA is being executed.

Our revised system keeps the same Wishbone-centric structure as Amber and makes the following revisions:

1. Heavy modifications to the core.
2. Add a PS/2 controller and link to the interrupt unit.
3. Remove the two UART controllers and the Ethernet MAC, as these are superfluous for our intended demo.

4. Add an HDMI controller with a memory-mapped frame buffer.
5. Revise the memory subsystem to either use the DDR3 SODIMM on our VC707 board or, if the capacity is not too constraining, the Virtex 7s block RAMs. We ultimately selected the latter, for reasons detailed later.

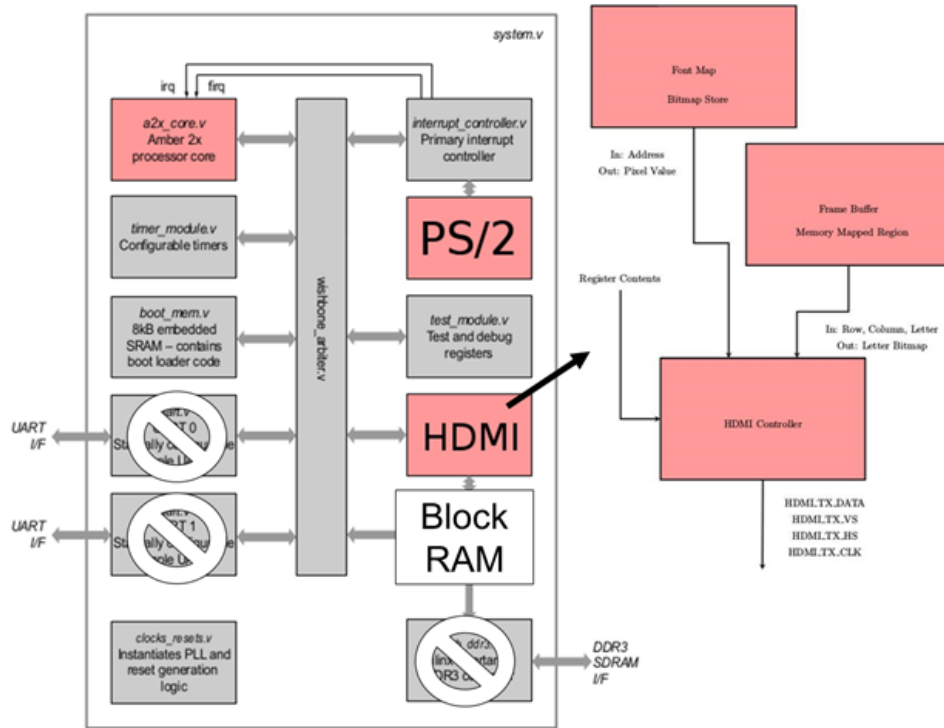


Figure 3: The modified Beryl system diagram.

3.2 Beryl: out-of-order core

Our first step toward MorphCore was to completely restructure Amber to support out-of-order execution. Our design is based on Tomasulos standard OoO algorithm. As instructions are decoded, they are inserted into reservation stations where they wait on any not-yet-ready operands before being dispatched to their respective execution units. A tag store assigns this instruction a “tag”, and the register file waits on this tag before updating its data for the instructions destination register. After being executed, an instruction broadcasts its tag and result on a tag bus, which connects to each of the reservation stations, the register file, and the tag store. This makes the tag available again for another instruction, updates the saved state of any other instructions waiting on this data, and updates the architectural register state of the processor.

Our Out of Order core uses a 4-stage pipeline – Fetch, Decode, Dispatch, and Execute – as opposed to Ambers 3 or 5 stages:

3.2.1 Fetch

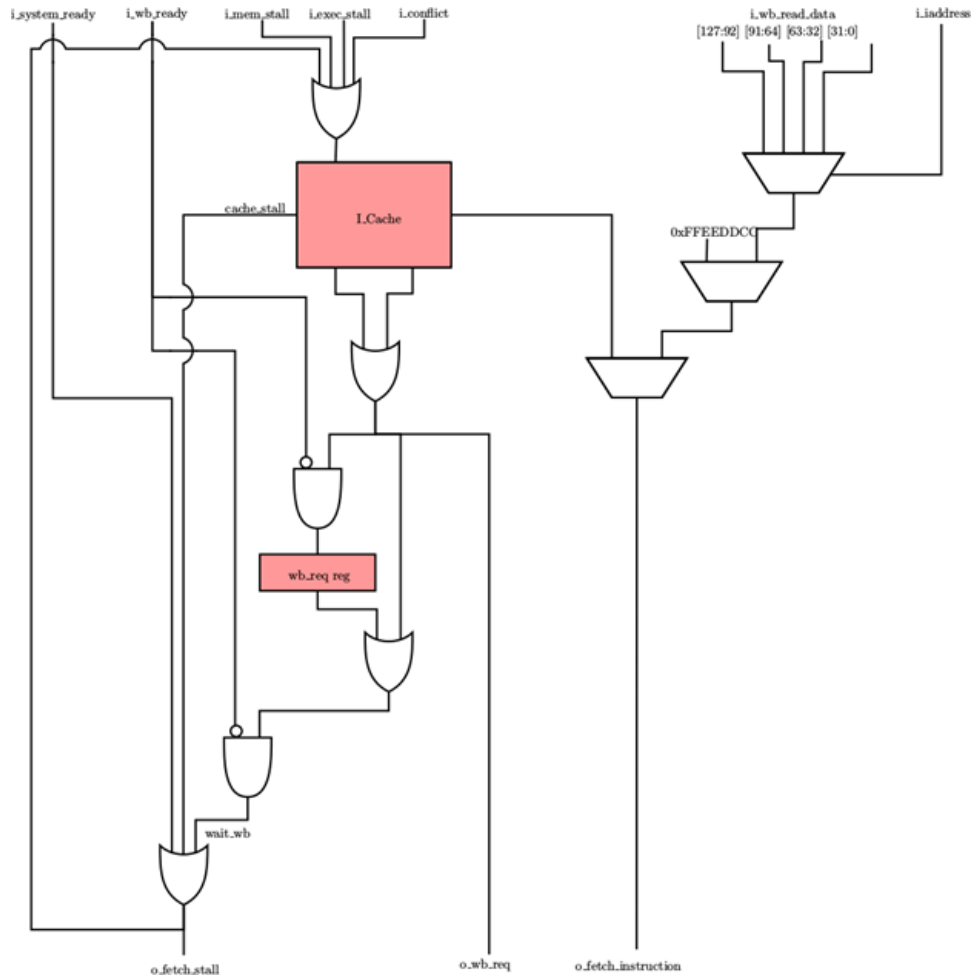


Figure 4: Beryl fetch stage.

This stage operates very similarly to Ambers Fetch. Instructions are fetched from memory, and a L1 instruction cache is employed to accelerate access to recently-used instructions. The key difference in our design is that we must flush the output of this stage when a branch is taken. This is done by setting the instruction output to a Beryl-specific NOP instruction (unused in the ARM ISA), `0xF0000000`.

3.2.2 Decode

Portions of Beryls Decode stage are similar to Ambers, but, as noted above, we removed the state machine for handling multicyle instructions. We also added logic to implement proper MSR and MRS instructions with a dedicated SPSR and set of CPSRs. Removal of the explicit FSM also eases interrupt handling, as we need not worry about remembering whether or not an IRQ or FIRQ has been received while were in the midst of a multi-cycle

instruction; we can immediately inject a command to branch to the appropriate interrupt vector.

Since this stage is essentially just a big blob of combinational logic, including a structural diagram would not be particularly useful.

3.2.3 Dispatch

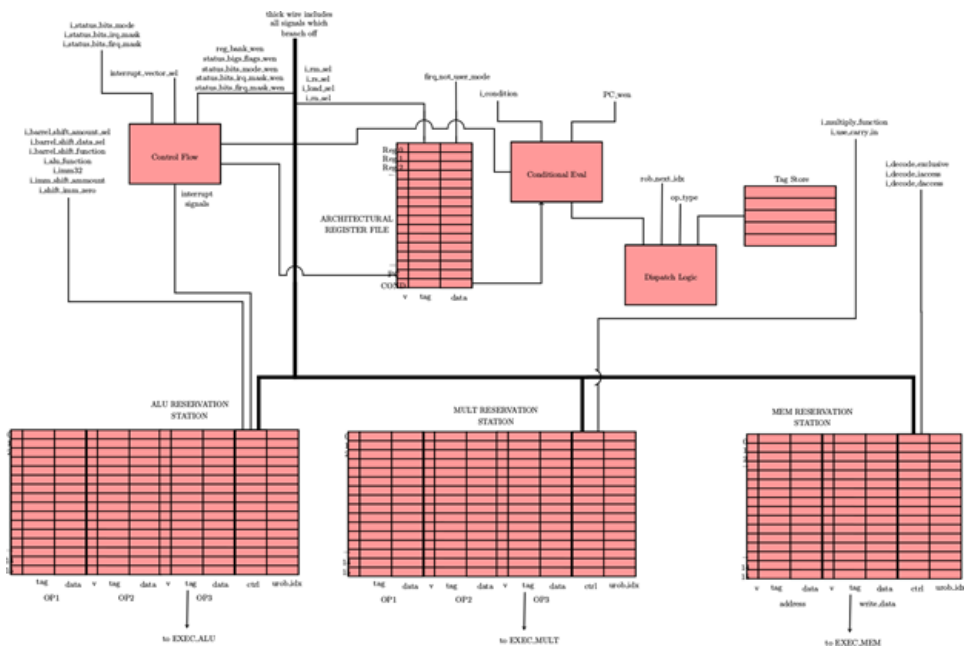


Figure 5: Beryl dispatch stage.

The Dispatch stage is the heart of the out-of-order core design. It is comprised of three main modules plus interconnection logic: the tag store, the register file, and the reservation stations (one each for ALU, multiplication, and memory operations).

Tag Store

The tag store tracks which tags are currently available for assignment to newly decoded instructions. By extension, it also tracks how much space is available in each of the three aforementioned reservation stations.

Any instruction with a destination register is assigned at least one tag corresponding to its designated execution unit. In the case of memory instructions, up to two tags may be assigned, one for address computation in the ALU and one for the memory operation itself.

Tags are 6 bits in length. The upper two bits designate the type of operation to which the tag is assigned; 00=ALU, 01=multiplier, and 10=memory. The “11” tags were originally reserved for other execution units to be added later (e.g. a floating-point unit), but ultimately, these were also assigned to the ALU since both memory and arithmetic instructions may need to dispatch an operation to the ALU reservation station. The low 4 bits of a tag comprise a unique identifier.

A tag is only retired, i.e. made available for assignment to new instructions, after its current instruction has been retired. This ensures that no two operations can conflict with each others tags, and is also part of the mechanism for tracking when a given reservation station is full.

If insufficient tags are available for a given operation to proceed, the core will stall.

Register file

The register file contains the standard set of 15 ARM registers (R0-R14) plus the program counter (R15), a separate R8-R12 for use in the processors FIRQ (fast interrupt) mode, and a separate R13-R14 for each of the SVC (supervisor), IRQ (interrupt), and FIRQ modes.

Each register is now implemented as a Verilog struct with a Valid bit, 6 Tag bits, and 32 Data bits. If a register is the destination for a decoded instruction, the Valid bit is set to 0 and the Tag bits are set to the tag assigned by the tag store. Each cycle, the registers data is updated and it is set to Valid if the registers current tag matches a tag on the output of any of the execution units.

If a register is invalid but another incoming instruction also writes to that register, the register remains invalid and its tag is updated to that of the new instruction. As discussed below, any instructions waiting on this register value will already be waiting on the proper tag, so updating the register files tag will not break correctness. Register-renaming, then, lets us avoid stalling for “false” write-after-write dependencies.

Reservation stations

The reservation stations record information about pending ALU, multiply, or memory operations and sniff the tag buses for data that was not yet ready when an instruction was decoded. They select a single instruction per cycle to dispatch to each of the execution units; older instructions are prioritized. If the newly decoded instruction is ready to dispatch but nothing else is yet ready, it is dispatched immediately.

Originally, the reservation stations were designed symmetrically. Each had 16 slots for waiting instructions, and each waited on all operands (in terms of the architectural register values) before being dispatched. Ultimately, however, this plan was revised. We decided it

would be more resource-efficient to handle memory address computation in the ALU instead of duplicating these resources in the Memory stage, particularly since certain pre- or post-indexed memory instructions in the ARM architecture also write back the computed address to a register. Thus, when a memory instruction is decoded, it is normally dispatched as two separate operations: one for address computation in the ALU, one for the “actual” load, store, or swap.

Because some memory instructions dispatch an operation both through the ALU and through the memory unit, we decided to double the size of the ALU reservation station, to 32 slots. The formerly reserved “11”-prefixed tags were then assigned to the ALU, in addition to the “00” tags.

Finally, the memory reservation station actually operates as a queue rather than as an out-of-order reservation station. This is due to the extreme logic expense and interface complexity of address comparison between pending memory operations, rolling back speculatively executed loads or stores, etc.. Given our systems relatively large instruction window (compared with the latency of any given instruction), this does not pose as significant a performance penalty as it might in another system.

Branch targets are computed and resolved immediately if their operands are available. If not, the core stalls until the next valid instruction fetch address has been calculated. This also applies to any ALU or memory operations whose destination register is the program counter (R15).

In the ARM architecture, most instructions may choose to set status flags (Negative, overflow, Carry-out, and Zero) based on their results. All instructions support predicated execution based on these flags. If a conditionally executed instruction is decoded but the status flags have been invalidated by an earlier instruction which sets these flags, the core stalls until the new set of flags is available. If the condition evaluates to “false”, the new instruction is thrown away and not inserted into any reservation station.

3.2.4 Execute

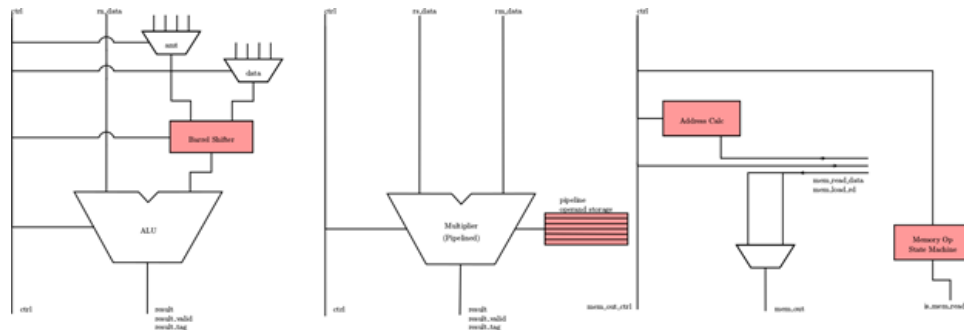


Figure 6: Beryl execute stage.

Our Execute stage is comprised of three separate substages: ALU, Multiply, and Memory.

ALU

The ALU stage is responsible for all non-multiplication integer operations. It also handles memory address computation, including pre- and post-indexing.

Like Amber, the ALU stage includes both a standard ARM barrel shifter unit and a general-purpose integer ALU. All instructions are retired in one cycle; that is, a result is available on the cycle after the instruction is dispatched to the stage.

Multiply

The Multiply stage handles MUL and MLA instructions. Unlike Amber, we take advantage of the dedicated hardware multiplication units on our Virtex-7 FPGA chip and instantiate a 6-cycle, fully pipelined multiplier instead of Ambers 34-cycle unit. If the instruction is MLA, the selected register value is added combinationally to the multipliers result at the end of the pipeline.

Memory

The Memory stage executes LDR, STR, and SWP instructions. LDM and STM are also supported in our design, but implementation of these is incomplete. The Memory unit is not pipelined both for simplicity and to ensure that certain operations remain atomic. LDR instructions are finished in two cycles, STR in three. SWP acts like an LDR followed by an STR, and a stage-internal state machine forces these two halves of the swap operation to be atomic. Similarly, LDM and STM cause the Memory stage to enter a different state that atomically loads or stores a series of registers before allowing any other memory operations

to proceed.

The original Amber core interfaces with a memory controller for a DDR3 SDRAM chip on the developers Spartan-6 prototyping board. We originally attempted to do something similar with the DDR3 SDRAM SODIMM on our VC707 board, but the complexity and latency of this were not worth the increased DRAM capacity. Instead, we chose to use the large amount of block RAM available on our Virtex-7 FPGA as our main system memory. This provided significant gains in memory operation latency, and those latency improvements allowed us to remove the data cache without any negative performance impact (thus dramatically reducing our core area and slightly shortening our critical path).

3.3 Chrysoberyl: superscalar core

Our second step towards MorphCore was to make our out-of-order core superscalar. Structurally, this design is extremely similar to Beryl, but adds various execution units and logic to enable fetching multiple instructions per cycle (as detailed below).

3.3.1 Fetch

The Chrysoberyl Fetch stage is mostly identical to Beryl's Fetch. The key difference is that, since memory fetches are 128 bits (4x 32-bit instructions) wide, we fetch all 128 bits at once and pipe each 32-bit instruction to a duplicated Decode unit. This provides a theoretical maximum instructions-per-clock value of 4.0.

3.3.2 Decode

In Chrysoberyl, instruction decode happens in exactly the same way as in Beryl. The sole exception to this is interrupt handling. If an IRQ or FIRQ is received, the 0th Decode stage processes the interrupt and injects a command to branch to the appropriate interrupt vector into the Dispatch stage. The other three Decode stages set their output to NOP.

3.3.3 Dispatch

The Dispatch stage is expanded to incorporate careful inter-instruction dependency checking:

First, the tag store is expanded to be capable of providing 4 times the number of destination tags per cycle to newly decoded instructions. The operands of each instruction in the set of four we fetched and decoded are checked against the destination registers of each earlier instruction in the set. If any match, that operand waits on the new tag for the applicable instruction in the fetch window instead of on the tag in the register file. If two instructions in the fetch window write to the same register, the register file waits on the tag of the later

instruction.

Second, the condition codes of any branch instructions in the fetch window are immediately evaluated. If any branch is taken, all later instructions in the window are ignored but all earlier instructions are dispatched normally. Fetch addresses are forced to be 128-bit-aligned. Upon branching, this means that the next instruction address will be the calculated program counter value modulo 4 (the number of instructions in the fetch window). The Dispatch stage also outputs a mask to the fetch stage showing which, if any, instructions following this address should be ignored.

If any instruction in the fetch window forces the core to stall, all later instructions in the window will also stall but all others are dispatched normally.

The interfaces to the ALU and multiply reservation stations are expanded to support both inputting and dispatching 4 of any type of operation at once. The interface to the memory station is expanded to support inputting 4 memory instructions simultaneously, but it is still constrained to dispatching only one instruction per cycle as no performance would be gained without also quad-porting the memory (a prohibitively costly endeavor). The reservation stations are not expanded in size in the Chrysoberyl design since testing with Beryl showed that they rarely filled to even 1/4 of their maximum capacity; however, they are easily expandable if need be.

Finally, since ARM expects that the program counter for each instruction (when it reaches the Execute stage) will be PC+2 but we increment the PC by 4 each cycle, we add a set of subtractors to the Dispatch stage to ensure that each instruction observes the expected architectural value of R15.

3.3.4 Execute

The execute stage is modified to include four of each of the ALU and multiplier units. As noted above, it continues to have just one memory unit. One instruction may be dispatched to and retired from each ALU or multiplier unit per cycle (given the pipelining of the multiplier and the single-cycle latency of the ALU). Expansion of the tag buses allows retirement of up to nine instructions simultaneously, as opposed to Beryls three; changes to this effect are also made in the tag store, register file, and reservation station modules.

3.4 Dendrite: MorphCore-like ARM core

Once the superscalar design was complete, we moved on to creating a MorphCore-like design. We say “MorphCore-like” since the original MorphCore was based on x86 instead of ARM, our pipeline is significantly different, and rather than splitting the superscalar out-of-order core into multiple in-order cores, we subdivide the OoO components (reservation stations, tag store, execution units, etc.) and assign each of them to one of four smaller,

non-superscalar, out-of-order cores.

Ultimately, the changes from Chrysoberyl to Dendrite are the following:

1. Add three separate Fetch stages in parallel with the first Fetch stage. These remain inactive until a mode switch is requested by the software. Since instruction memory in our design is read-only, there are no cache coherence issues.
2. Multiplex the output of each new Fetch stage with a 32-bit section of the superscalar instruction window from the original Fetch stage. The selector for these muxes is the current operating mode of the processor (single-core or multicore).
3. Decode stage 0 continues to handle all interrupts, whether in superscalar single-core mode or non-superscalar multicore mode.
4. Add three more register file modules to Dispatch, one for each of the newly split cores. Their inputs are always tied to the outputs of specific execution units.
5. Modify the tag store and reservation station logic to allow division of these structures resources among the four smaller cores.
6. The memory reservation station will remain largely the same; it will act as a queue for all memory operations. It will input an additional operand, carried through to the Memory execution units output, to designate which core generated a given memory request.
7. The execution units themselves require no significant changes. Since the Memory stage is unified among the cores, there would be no cache coherence issues even if the data cache were re-inserted into this stage.
8. A mode switch from single-core to multicore mode (or vice versa) is accomplished with a special software interrupt. Instruction fetch and decode are stalled until all reservation stations are clear and the memory queue is empty, at which point register file 0 is replicated across the other three register files (in the case of a single-multicore switch) and the core either splits or recombines.

3.5 ISA and Implementation Notes

The supported ISA for the current Beryl core (and its derivatives) is as follows:

Instruction	Description	Execution Unit	Implementation Notes
ADC	Add with Carry	ALU	Waits on status flags in addition to register operands
ADD	Add	ALU	

AND	Logical AND	ALU	
B	Branch	Special	Handled immediately without explicit dispatch
BIC	Bit clear	ALU	
BL	Branch and Link	Special	Handled immediately without explicit dispatch
CMN	Compare negative	ALU	Set flags, no register write
CMP	Compare	ALU	Set flags, no register write
EOR	Logical XOR	ALU	
LDR	Load register	ALU, MEM	
LDRB	Load register byte	ALU, MEM	
MLA	Multiply with Accumulate	MULT	
MOV	Move register	ALU	
MRS	Move status register to register	Special	Handled immediately without explicit dispatch. Uses dedicated CPSR and SPSR in dispatch stage
MSR	Move register to status register	Special	Handled immediately without explicit dispatch. Uses dedicated CPSR and SPSR in dispatch stage
MUL	Multiply	MULT	
MVN	Move not	ALU	
ORR	Logical OR	ALU	
RSB	Reverse Subtract	ALU	
RSC	Reverse Subtract with Carry	ALU	Waits on status flags in addition to register operands
SBC	Subtract with Carry	ALU	Waits on status flags in addition to register operands
STR	Store register	ALU, MEM	

STRB	Store register byte	ALU, MEM	
SUB	Subtract	ALU	
SWI	Software interrupt	Special	Handled immediately without dispatch
SWP	Swap	MEM	
SWPB	Swap byte	MEM	
TEQ	Test equivalence	ALU	Sets flags, no register write
TST	Test	ALU	Sets flags, no register write

4 Display Controller

We decided to memory-map the frame buffer as another area in memory. In Amber and Beryl, memory is organized into areas by macros which define a beginning and ending address to each section. Boot mem (also known as instruction memory), data memory, and all memory mapped I/O were mapped to different address ranges by these macros.

We decided to map the frame buffer to a large area after data memory, much like a memory mapped I/O. This division of memory areas helped our design be more modular, because each range could be defined as a separate Block RAM IP in Vivado and accessed by the starting range offset, where the starting address of that range became address 0x00 in the BRAM.

Both the boot memory and data memory were assessed separately by the wishbone by the Amber design, which supports modular design in their case as well, and we decided that adding the frame buffer BRAM to the wishbone would create the most consistent design. However, the wishbone slaves were controlled by a three bit signal, which only afforded eight slaves that were already all assigned. Therefore, we changed the function of slave 4, the UART 1 signal to that of the frame buffer memory interface.

Originally there were two UART interfaces, UART 0 and UART 1, and we were never intending to ever use more than one UART interface for our design. Because UART 1 was already hooked up to the wishbone in the same way as every other slave, including boot mem and data mem, rerouting the signals to the frame buffer was relatively simple.

In simulation, writing to and reading from the frame buffer was functioning correctly and only required specifying a certain address range to access when using the STR or LDR instructions in the assembly code. However, when running the same code on the board, the screen was not getting written to correctly, signifying a clocking issue. The HDMI monitor requires a very precise 27.027MHz clock, and otherwise the signals will not get transferred correctly.

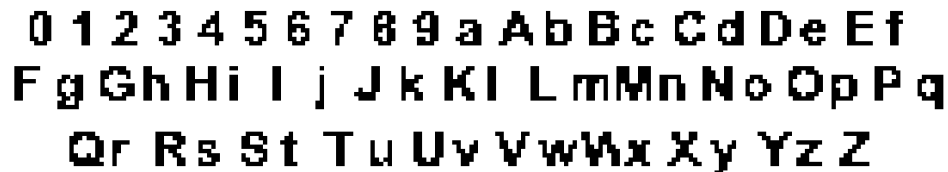
We eventually discovered that our performance gains in Beryl were enough that we could clock the entire system in time with the HDMI monitor, which solved the initial timing problems and afforded us a solid color screen, but no other detail from the buffer. However, at this point now we noticed an incorrect reading from memory. We would write a 2 every other pixel to memory in the assembly code, but we would get 0, 1, and 2 out of memory when the default value inside memory was A.

4.1 HDMI

The HDMI controller was divided into two major modes of operation, TEXT_MODE and PIXEL_MODE. The part of the module which interfaced with the HDMI chip on the VC707 was taken from Team SDK's code[3]. We originally attempted to implement this ourselves, but after running into some bugs with the I²C interface, we decided it would be easier to just use an existing solution. The part of the module which generated a color for each pixel, either by reading from the frame buffer or by reading from the font ROM.

Our screen is 720×480 pixels, and HDMI has a color depth of 12 bits per color. This works out to $720 \times 480 \times 3 \times 12 = 12,441,600$ bits of memory to store an entire frame in memory. In order to reduce the memory footprint of the frame buffer, we build a color map between 4 bit strings and full HDMI colors. We then stored these 4 bit strings in memory instead of the full colors. This saved us 11,059,200 bits of memory.

To display text, we essentially used an array of bitmaps corresponding to letters as a sprite map. From there, it was just a matter of keeping track of what pixel of what sprite was being drawn to the screen during the current clock cycle.



0 1 2 3 4 5 6 7 8 9 a A b B c C d D e E f
F g G h H i j J k K l L m M n N o O p P q
Q r R s S t T u U v V w W x X y Y z Z

Figure 7: Available characters in TEXT_MODE.

In order to generate these bitmaps, however, we wrote a Perl script which converted any system font which was accessible through `imagemagick` into a set of bitmaps, and then into a .COE file. To make it easy to perform this conversion, we output the bitmap files as .ppms, which is a bitmap file format that stores color data as human-readable ascii numbers.



Figure 8: A test pattern in TEXT_MODE.

4.2 VGA

It is worth mentioning that we started our adventure in video output with VGA. Reading previous reports it sounded like HDMI was more trouble than it would be worth to get working. Unfortunately, the Virtex-7 board does not have a VGA port. Our solution was to co-opt some of the GPIO pins which had been previously driving an LED display (see figure ??). This turned out to be substantially more trouble than we were expecting. Even though we didn't end up using VGA, we still describe our efforts here in the hopes that someone on a later project finds it useful.

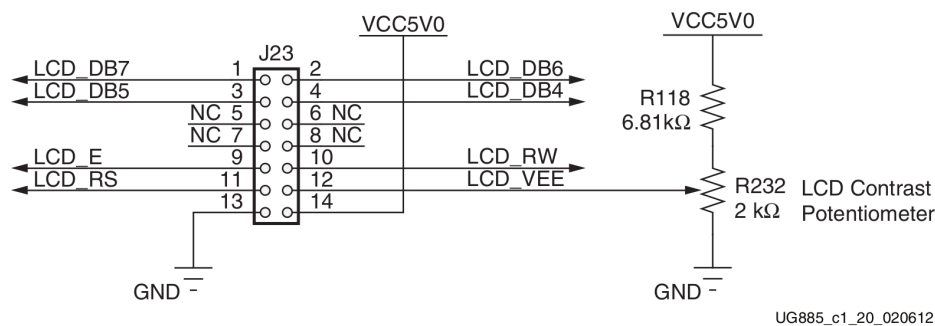


Figure 9: LCD Interface Circuit [4]

VGA is an analog standard with 15 pins, some of which are unused. The three color channels are on pins 1 through 3, and consist of analog voltages between $0v$ and $0.7v$. There

is also a vertical synchronization signal (pin 14), a horizontal synchronization signal (pin 13), grounds for HSYNC and VSYNC (pins 5 and 10), and a 5v pin, 9. It is worth noting that each of the color channels have their own grounds as well, in the form of the red, green, and blue return pins, 6, 7, and 8. The other pins are used for various I²C communications with the monitor. We did not use them.

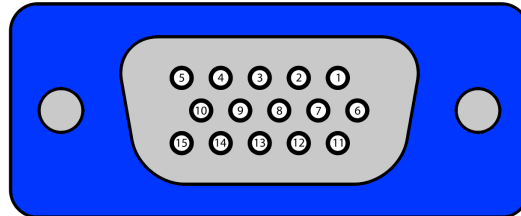


Figure 10: VGA Pinout and Wire Colors

In order to connect the vc707 GPIO pins to a VGA monitor, we cut open a VGA cable and determined which internal wires were connected to which ports. In figure 10, we show the colors of the wires and the pins they were connected to. Keep in mind that the pins in the diagram are labeled from the female side. In order to convert between the digital GPIO outputs and the analog inputs required for the VGA color channels, we used an R2R resistor network¹ as a digital to analog converter. We tied all of the grounds together, and tied all of the color return pins to ground as well.

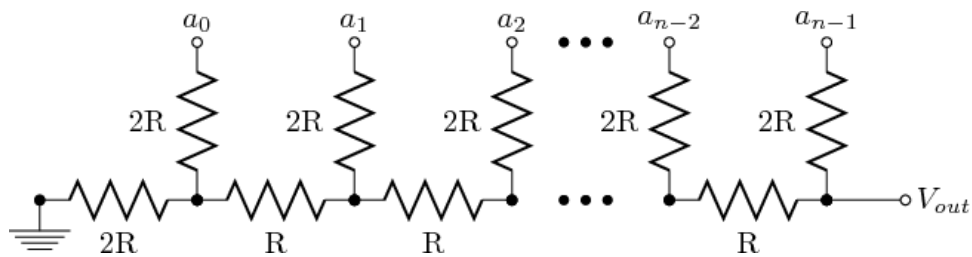


Figure 11: An example of an R2R DAC.

Since we only had access to the 7 GPIO pins from the LCD connector, and needed one each for horizontal and vertical sync, we were left with only 5 pins with which to generate color channels. We ended up allocating two pins each to red and blue, and one for green. The end result was somewhat disappointing. The colors were fairly ugly and limited in range. Since the connection was electrically messy and temperamental, and the colors were lackluster, we decided to scrap our VGA approach and switch to HDMI.

¹Image from https://en.wikipedia.org/wiki/Resistor_ladder

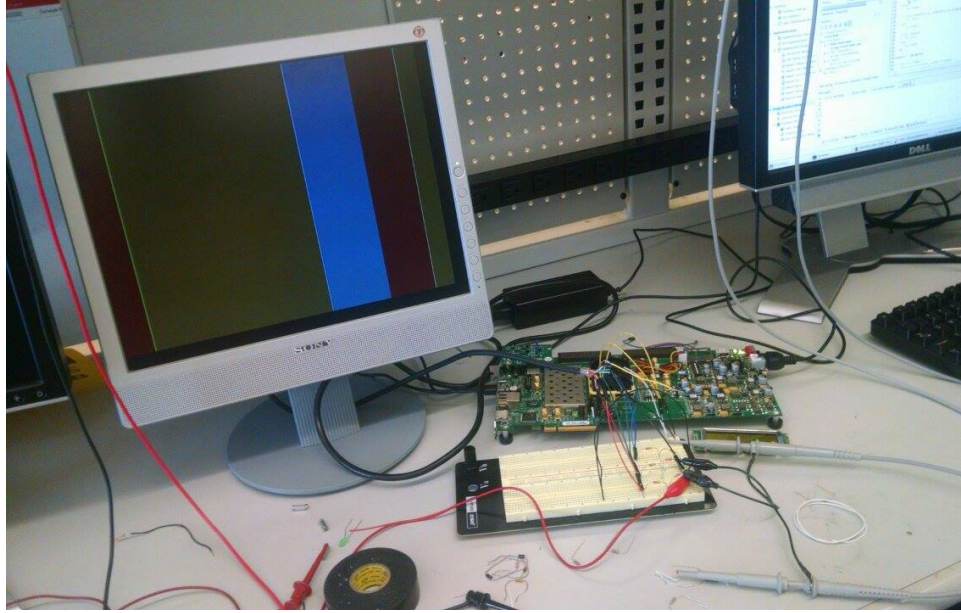


Figure 12: The results of our VGA efforts

5 Keyboard

5.1 Hardware Interface

For input to our system, we decided that we would use a PS/2 keyboard. The PS/2 standard is, like VGA, pretty simple. Also like VGA, the virtex-7 board did not have a PS/2 port. PS/2 operates by sending packets which are clocked by the device. There are thus two wires, one for the clock and one for data. For a keyboard, packets are 11 bits long, and consist of a start bit, which is always 0, 8 data bits, an odd parity bit, and a stop bit, which is always 1.

By default, the clock and data bits are both high. They can be pulled down by the host to communicate with the device, although we did not bother to implement this. The voltage that PS/2 operates at is 5v. In about 15 minutes of wiring, we managed to get useable output from a PS/2 keyboard. The signal can be seen on an oscilloscope in figure 13.

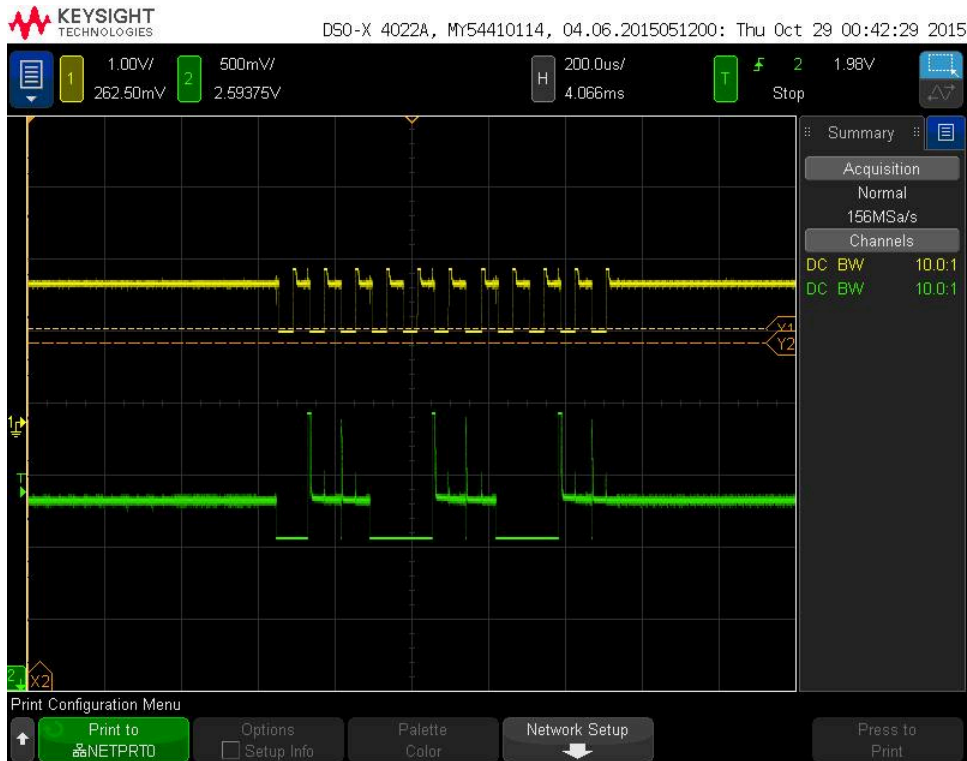


Figure 13: PS/2 on an Oscilloscope

Unfortunately, everything went downhill from here. At some point during the semester, we stopped being able to read values from the PS/2 keyboard across the GPIO pins. We could still read values which we set manually, by forcing the PS/2 lines to ground, but every effort to read values sent by the keyboard itself was met with failure.

We had a plan for determining what was wrong with the PS/2 connection, but unfortunately higher priorities prevented us from debugging it before the demo. Our suspicion is that something changed the current draw from the keyboard and this brought it outside of the operating range of the GPIO pins, but we honestly don't know.

5.2 PS/2 Interrupts

In standard ARM architecture, when an interrupt gets triggered the status bits, held in the current program status register, or cpsr, get saved into the saved program status register, or spsr, so that the current mode can be switched to IRQ mode (the shortened name for interrupts). Then, the register file gets banked onto the stack so that when the interrupt returns, the registers can be popped off the stack and execution can resume as normal. Additionally, the new program counter gets saved to 0x18, which is the ARM standard for IRQ interrupts. At 0x18, there is an address for the program to jump to where the IRQ handler is located. At the end of the handler, the IRQ handler code must unbank the registers and re-instate the spsr into the cpsr so execution in the previous mode can resume before returning from

the interrupt handler.

In the Amber design, the designers removed the `spsr` and `cpsr` as separate register entities and saved the status bits and condition bits (which are all saved in the `cpsr`) into separate program signals that are paired with the current program counter. Unfortunately, this made writing an interrupt handler very confusing. Eventually, after looking through the instructions implemented by the Amber core, we realized that there is an instruction called `TEQ` that can assign specific bits to the current `cpsr` that isn't used by most ARM convention because of the danger of writing directly to status bits of the processor.

The general layout of the interrupt handling in the Amber system that was eventually carried over to the Beryl system used the wishbone as the main data line. There is an interrupt controller module that is connected to external signals, including the UART connections, the timers, the general `IRQ` and the `FIQ` (fast interrupt). If any of these signals is detected, a signal would be sent through the wishbone, along with any data necessary for the core to use during handling, to the system module where it is sent directly to the core. In Amber, the designers decided to send the signals first to the decode stage in order to be analyzed and then sent to the execute stage in easily managed signals. Then, in the case of an `IRQ`, the mask, the signal which determines in execute the type of interrupt being handled, determines the next program counter to set and how to change the status bits. The types of interrupts are `IRQ`, `FIQ`, timer, UART, and in execute, a software interrupt, or `SWI`.

We determined that setting up `PS/2`, at the verilog level, involved hooking up a signal to the interrupt controller module that collected external signals from the GPIO pins on the board that were hooked up to the `PS/2` connector on the keyboard. At that point, once that signal also triggered the output `IRQ` signal, the rest of the chain would be executed properly, and all that needed to be written was an assembly handler that could be loaded by `.coe` file into the boot memory BRAM. However, the `PS/2` keyboard runs on its own external clock. Having two clocks in a module, one to collect data and one to send data out on, caused some timing issues with Vivado and with the module. Additionally, while we were able to collect correct data waveforms on the oscilloscope, we were unable to send correct keyboard stroke information through the GPIO pins. This led to our inability to have a complete keyboard hookup for the final demo.

6 Testing and Verification

The original Amber design downloaded from OpenCores contained a directory of assembly tests, as well as a directory of SystemVerilog tests that were used to verify the hardware design. Originally we created a text file that describes what instructions each test uses and how robust each test is, and how useful the functionality would be in our final design. Tests like those for the ethernet mac could safely be ignored, while a favorite of ours, `add.S`, was used for all initial pipeline testing for Amber in order to get the design up and running before modifications.

Later in the semester, additional tests were written that use the small instruction set more robustly, such as those that use all arithmetic instructions, all bitwise operations, branching instructions, comparisons, and conditional execution. Separate tests were written for the barrel shifter, the multiplier that was added later, and memory access instructions. Towards the end of verification, larger test cases that were being prepped for potential demos were run on the core to verify long-term execution capability and accuracy.

6.1 Software

There were two types of demo code written for the Beryl core: those that explicitly used the frame buffer for drawing to the screen and those that only used arithmetic computation. Our original ambitious demo was going to be running a full Linux OS, like the Amber core claimed to do. However, we quickly downscaled our plan to original demo code, that would hopefully be written in C and compiled to run on the Beryl core.

While the compilation from C toolchain worked with no errors, the ARM assembly code assembled created a complicated stack structure at arbitrary virtual memory addresses that didnt translate well to our baremetal core and had to be modified heavily. Eventually, we just wrote code in ARM assembly by hand. The code written by hand was better able to be modified to access specific memory addresses, such as the frame buffer, and to use certain registers that were easier for us to check on the waveforms.

Our demo plan was to draw a fractal, probably the koch snowflake, to the frame buffer. Demo code was written that used Bresenhams line algorithm to draw one line between two previously decided points on the screen based on the monitors resolution. However, while the data was written correctly to the frame buffer, the line did not get drawn to the screen correctly. The issues surrounding the frame buffer were described previously in the display controller section and will not be described again here.

Because the frame buffer was not functional, we decided to stick with arithmetic operations in order to capitalize on the performance gains of our processor modifications. We decided on a couple easily implemented mathematical operations, fibonacci number generation and matrix multiplication. Both programs were written in assembly and ready to be executed on the core, but we only ended up with time to integrate an HDMI interface for the fibonacci number generation register files. We decided that having such a visual for matrix multiplication would be confusing, but we were too time-crunched to create a separate interface for it. We would have liked to be able to have a user-input a choice of which demo program to run, and a separate visual interface for each, but unfortunately that was not completed in time.

7 Results, Status, and Future Work

7.1 Project Status

Unfortunately, we weren't able to complete the full morphing Dendrite core before demo day. Although Chrysoberyl is extremely close to completion, other aspects of the system had to be prioritized shortly prior to our demo date. Nonetheless, our Beryl core is fully functional and yields massive performance gains over Amber, as detailed below.

The HDMI display controller allowed the display of both bitmaps from the frame buffer and arbitrary text output, although we restrict it to producing only a written display of the values in the architectural registers. PS/2 is very close to being functional. If we had debugged whatever problem was preventing us from reading values over the GPIO pins, we probably would have been able to use a PS/2 keyboard during the demo.

It would likely have been a fairly significant stretch to get a full operating system running on our core, since running an OS is not a particularly fault-tolerant application for a CPU, but we were definitely able to run arbitrary ARM assembly on the core without running into any issues.

7.2 System Performance

We analyzed the performance of our Beryl system with a set of benchmarks designed to exercise different portions of the processors arithmetic, branching, and memory capabilities. We chose not to include multiplication-related tests in our results since any performance gains in these (versus Amber) would be more due to our use of a more efficient multiplier than to our out-of-order architecture. Indeed, some simple multiplication-heavy tests showed somewhere between a 4x-10x performance gain on Beryl, which is unrealistic to expect from a move to out-of-order execution alone.

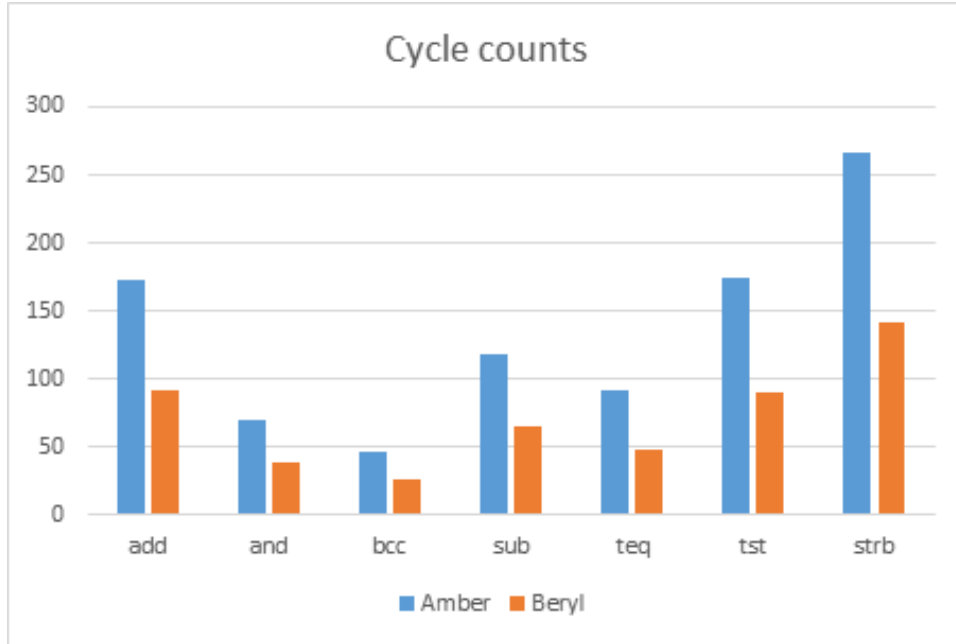


Figure 14: Relative benchmark performance of Amber and Beryl.

Considering cycle counts alone, we see an average 87% performance gain across the board when moving from Amber to Beryl. Post-synthesis analysis with a VC707 board target yields the following results for power consumption, logic area, and critical path:

	Amber	Beryl
Est. Power Consumption	0.429W	0.514W
LUT Utilization	6.91%	12.59%
Slice Register Utilization	7.60%	4.51%
Max Attainable Clock	34.39MHz	35.47MHz

Figure 15: Relative resource utilization of Amber and Beryl.

There is a surprisingly low increase in power consumption from Amber to Beryl given the observed performance improvement. The critical path shortening from Amber to Beryl is due partly to careful design and partly to removal of the no-longer-important data cache (which also accounts for the decrease in slice register utilization). All this, however, comes at a $\sim 82\%$ increase in required LUTs on our Virtex-7 FPGA.

Incorporating the clock speed differential into our previous cycle count results, we obtain the following graph showing the overall performance gains from Amber to Beryl:

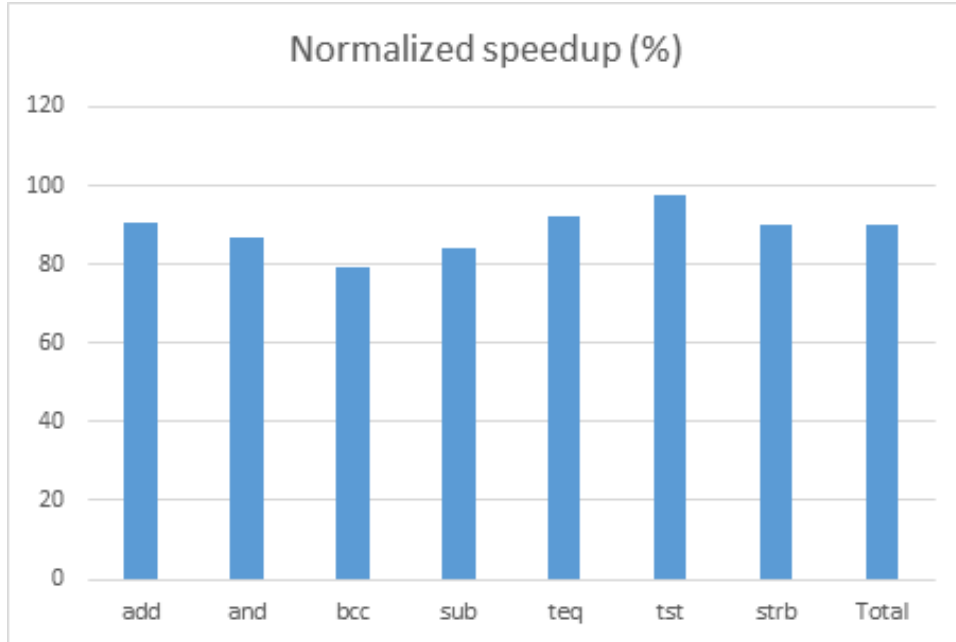


Figure 16: Normalized speedup between Amber and Beryl.

In total, we see performance gains from 79% to 97% from the move to out-of-order processing, with an average improvement of $\sim 90\%$. We expect this number would increase significantly by moving to Chrysoberyl; however, the additional gains would be limited by our instruction cache structure and handling of control flow operations. Chrysoberyl's more-complex dispatch logic would probably also lengthen the critical path of the core. Substantial performance gains could also come from adding a proper branch predictor and optimizing the pipeline to shorten the design's critical path.

7.3 Future Work

We definitely intend to continue working on this project after this semester. The MorphCore concept has a lot of promise, and our designs are close enough to fruition that we are inspired to continue our efforts. Chrysoberyl, in particular, could probably be completed in no more than a few days of intense work. Beyond finishing the Chrysoberyl and Dendrite cores, we hope to add a branch predictor, optimize the pipeline to shorten the critical path, improve our I/O subsystems, and ultimately boot a small OS kernel via our system running on physical hardware.

8 Lessons Learned

By far our most important lesson learned was to never underestimate the complexity of a toolchain. Figuring out how to use all the capabilities of our VC707 board was extremely

involved, particularly with HDMI. The boards relative lack of GPIO pins and “standard” connectors such as VGA and PS/2 made creating I/O interfaces with the outside world quite difficult; if a future project needs these, wed strongly recommend either using a different board, borrowing a known-working HDMI implementation from a previous project, or designing the project so it doesnt depend on these protocols in the first place.

More importantly, though, Vivado was the one of the most stubborn and buggy pieces of software weve ever encountered. It was extremely slow, sometimes failed to launch properly due to Java VM errors, often crashed silently and for no apparent reason while running simulations, and the list goes on. When it worked, it was a very useful and full-featured toolkit for simulation, synthesis, IP generation, design analysis, and the like. But getting it to actually work properly was not worth the hassle. We wish wed had access to an Altera board with comparable FPGA capabilities to our VC707 so we couldve used Quartus II instead, but we shouldnt look the proverbial gift horse in the mouth. Despite all the bugs, thanks are due to Xilinx, and professors Nace and Lucia, for providing the VC707.

In addition to the issues we had with a lack of hardware ports, we also ran into an issue early in the semester where our board would fail checksum checks after writing a bitstream. It turned out that this was because there was some sort of hardware issue with the board, although we did not discover this without many hours of poring over the bitstream generation and downloading process looking for other possible human error.

As far as the design/implementation process is concerned, we learned to never trust that other people who wrote code youre working with made sane design decisions. Amber was full of problems ranging from broken ‘include dependencies, an opaque and undocumented FSM in their Decode stage, bugs with data dependencies from memory operations, a hideously inefficient multiplier, and more. It was helpful to have a starting point and not have to build everything from scratch, but whenever you use someone elses code, dont trust it – verify, verify, verify!

On the project management end, we aimed really, really high. Aiming high pushed us forward its probably why we got as much done as we did and had a reasonable demo but we couldnt do everything. We didnt fully appreciate how difficult it would be to get the toolchain working and to synthesize the original Amber core to our board, which put us behind schedule from the outset. However, we still finished a substantial portion of our initial (and overly ambitious) goals because we set stepping stones along the way at demoable points. We were able to design and construct a full computer system around a modern, out-of-order, 32-bit ARM processor with HDMI output and (almost finished) keyboard input, which we feel is a substantial accomplishment in its own right.

9 Personal Notes

9.1 A Note from Pete

Initially, I was responsible for architecting, designing, and implementing our three processor core variants, while my teammates would work on developing other system components (HDMI, PS/2, UART, demo software, etc.). The work would be pipelined in such a way that I could finish a core design one week, implement it the next, do some initial debugging, and then pass it off for more-formal verification. This plan went out the window around the second or third week of class when Murphy reared his ugly head. Vivados obstinacy, combined with hardware problems, a dead VC707 board, issues cleaning the original Amber code, and a dying lab computer, meant there I was needed more to help debug these problems before impending midterm demo deadlines than to finish building the processor cores.

In the end, I architected and designed all three of our processor cores, implemented Beryl and got most of the way done with Chrysoberyl, made an almost-working HDMI interface for our board before the handwriting recognition team generously provided us with their code, and did a lot of debugging of the original Amber 25 core/system while we tried to get it synthesized on our VC707 board. Due to project time constraints, I ended up doing the verification of my Beryl core prior to our final demo. I also supported the debugging of other system components/HDMI, benchmarked Beryl, made our poster, and wrote up a ton of documentation and design notes about the processor cores. Towards the beginning of the semester, I spent about 12-15hrs/wk on this project, increasing to 20hrs/wk by midsemester. This remained fairly steady until the last two weeks, at which point I was in lab pretty much continuously anytime I didnt have to be at another class.

One of the biggest challenges I personally faced with the core was finding a good balance between creating a super-detailed design and just starting in on the implementation. I probably ended up spending too much time on the design phase, but this front-loaded work made implementation far easier after coming up with an extremely thorough design for Beryl, for instance, it only took me about a week to get the implementation into an at least partially functional state (although debugging and verifying it took significantly longer). In retrospect, my biggest impediments to completing Chrysoberyl and Dendrite were either that I was blocking on teammates to finish certain components or that I became too involved in trying to deal with broken system integration, HDMI components, etc. before demo days/status meetings. But when deadlines approach, its hard to stay focused on components that either arent due as imminently or arent as critical for a demo.

The one thing Id change about this courses structure can be summed up in four little words: Kill Vivado With Fire. Perhaps its better than past years ISE, but thats not saying much. If you, future 545-taker, can swing it, use something Altera-based instead of the hideously buggy toolchain we were forced to endure. And make sure that if you take on an ambitious project, like trying to build three increasingly complex versions of a CPU and put them in a full-fledged computer system, you include demo-able stepping stones along the way.

On the whole, though, this has been a very worthwhile course. I've learned a lot about the digital design process, working with industry-standard tools and hardware devices, and both the software and hardware structure of ARM-based systems. I'm really glad we ended up seeing the Beryl core running inside a mostly-complete computer, and the performance gains we observed were highly satisfying. I'm also grateful to our professors for letting us pursue such an exploratory project rather than take the default build-a-video-game approach.

9.2 A Note from Amanda

During this semester, my original job was to be the head of verification and the ARM architecture expert that the others could go to during the semester. I also assigned myself as a floating helper to latch on to whomever needed extra eyes at any given point during the semester. At the beginning I was tasked with understanding the test assembly given to us by the Amber team, and the scope of the instructions that were implemented by the core. Unfortunately, the Amber core we got from OpenCores was poorly documented, so that meant more time for us to read through the code itself in order to understand the implementation. Originally, I was also supposed to oversee the structure of the new core and oversee real-time verification for pipeline stages, but unfortunately I was not given access to the core by my teammate until it was already verified by him. I did along the way write assembly tests for him to use when debugging, including a multiplication test, a memory access test, and more complex arithmetic and bitwise operations.

By the end of the semester my main responsibilities were head assembly-code writer and demo code head, ARM architecture expert, and HDMI integration support. I also wrote up a guide on interrupt handling in ARM for us to use for integrating the PS/2 keyboard, which unfortunately was not able to be completed on time. The guide is also written earlier in this report in the interrupt handling section. Brian and I both handled HDMI integration, where I was responsible for testing potential demo code and the frame buffer integration, and he was responsible for the bitmasks for the library and setting up the frame buffer BRAM. I ended up writing assembly code for the fibonacci sequence that was used in our final demo, a 3x3 matrix multiplication, and various frame buffer assignment programs that assigned vertical colored lines and connected two predetermined points with a line for frame buffer testing, along with all of the small test programs.

I definitely spent more time on this project at the end of the semester than at the beginning, due to some group tension and my busy interview schedule in September and early October. However, I think I spent at least 10 hours a week on this class, and by Thanksgiving that number was probably closer to 15 hours a week.

Overall, I have mixed feelings about this class. I am a hardware engineer, primarily, and I took this class because it was in my concentration, and because I really enjoy studying computer architecture. However, because of some tension within the group, I felt hindered in actually doing work I was interested in, and my learning opportunity for this semester was limited to toolchain operation. Additionally, the group tension really did a number on

my nerves and my overall attitude towards the class, and it made me not as productive as I could have been, and I do regret that that happened. And while we chose a really cool project topic, I think that we could have gotten more done with a more even distribution of the work and a more concrete idea of what our demo would be in each stepping stone case. However, I really liked having the opportunity to take what I was learning in another class, in this case Embedded Systems, and apply it to this project by writing code and parsing through the verilog, but it would have been nice to actually write some verilog and to be more involved in low-level design decisions.

Overall, I think that this class was kind of a wash. I think that it had the opportunity for cool exploration and inventiveness, but it was kind of passed over me. I am happy with our final result, though. Im also really happy some of my best friends from other majors came to see our demo.

9.3 A Note from Brian

At the beginning of the semester, I had reservations about working on such an architecture heavy project. I hadn't yet taken 18 – 447, and the general thrust of the course is definitely towards implementing older, well documented systems versus open-ended designs for modern processors. Nevertheless, it was an incredibly valuable experience designing in an open space.

I ended up spending between 10 and 40 hours a week on this project, with a typical number being between 15 and 20. Having not taken any computer architecture before starting a computer architecture heavy project, I was committed early on to spending extra time catching up on concepts like Tomasulo's algorithm. I feel like I learned a lot from the experience, although I didn't apply it particularly much to the parts of the project I worked on.

One thing which was somewhat frustrating was that most of the work that I did over the semester, on PS/2, VGA, and the graphics mode of the HDMI output controller ended up not being used in the final project. This is a side effect of not having a strong idea of what our demo was going to look like until fairly late in the semester. I also spent a lot of time determining that we didn't want to use ethernet or UART.

That being said, the work which I did which did make it into the final demo was very satisfying. The text rendering module provided opportunities to think about hard problems optimizing the storage of the character bitmaps in memory and efficiently moving them from memory into registers to be printed to the screen. I spent a lot of happy time learning about compression-efficient encodings of sparse matrices, although ultimately the solution was to just use a bunch of memory, since we had plenty to spare by demo time.

My advice for future students is that if they are doing an open ended project like this, they should be sure to have a demo in mind as early as possible. We did not actually have the idea for our final demo until after the in-class demo, and then had to scramble to get something presentable working. Also, do not be afraid to write quick scripts in a language

like Perl or Python to solve data format problems, or produce highly structured pieces of verilog which are not amenable to the methods of parameterization and generate statements.

References

- [1] Amber arm-compatible core :: Overview. <http://opencores.org/project,amber>. Accessed on: Dec 14, 2015.
- [2] Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 305–316, Washington, DC, USA, 2012. IEEE Computer Society.
- [3] Team SDK. Team-sdk-545. <https://github.com/kkudrolli/Team-SDK-545>, 2015.
- [4] Xilinx. *DISCLAIMER SCHEM, ROHS COMPLIANT vc707 EVALUATION PLATFORM*, 1.0 edition, 4 2012.