EECS 151LB Project Report

Team 02: Averal Kandala, Seiya Ono

December 14th, 2018

1 Project Functional Description and Design Requirements

In beginning this project, our primary objective was to mould the key, given information about the requirements of the design into a three-stage RISC-V central processing unit (or "CPU") pipeline structure that would have an optimal critical path and be coherent with a certain forwarding strategy. The greatest constraint in this initial design phase turned out to be the need for synchronous read *and* synchronous write memories, resulting in four discrete sequential blocks — the program counter, the instruction memory abstraction, the data memory abstraction, and the write-back to the register file — that did not give us much leeway in selecting a pipelining strategy, since we wanted to maximize use of already-present sequential elements in pipeline divisions to avoid forwarding issues between stages later on. Because the synchronous register file write-back and progression of the program counter can happen simultaneously, we were able to encapsulate this functionality in one pipeline stage, and, from there, the two other stages defined themselves fairly neatly, with one "execute" stage beginning with the fetching of the next instruction from the instruction memory abstraction and ending at the input to the data memory abstraction and another "memory" stage spanning from the output of the data memory abstraction to the write-back data and address ports of the register file (the remainder of the register file is located within the "execute" stage).

This pipelining division yields a clear, immutable critical path in the "execute" stage, as data must progress through the instruction memory abstraction, the register file, and various sub-circuits (including the arithmetic logic unit, or "ALU") before reaching the data memory abstraction; a key decision made in this design was to continue with this structure, even with the knowledge that we would not be able to alter the critical path much in later phases, due to the simple fact that it lends itself nicely to a "branch not taken" assumption on jumps and branches and does not require much additional infrastructure to support forwarding for such control hazards and typical data hazards (which results in a relatively low cycles-per-instruction value, or "CPI"). Additionally, we felt it was important to de-complicate the rather convoluted memory hierarchy (from the perspective of the datapath), and therefore created a wrapper module for all memory elements which has different ports for specific uses (e.g. a certain set of address and data ports is only used for the instruction memory abstraction), and this allowed us to mostly straightforwardly insert this one module into our design without adding much peripheral circuitry or complication. And, to round out the design, we implemented two local controllers for the "execute" and "memory" stages, as well as one global controller (named `master_controller` in our code) to handle data forwarding and `NOOP`s (instructions without function) in the case of successful jumps and branches.

2 High-level Organization

Delving deeper into our design from the overview provided in the last section, the first stage of our pipeline, which contains the program counter and (nominally) the synchronous write-back to the register file, is essentially nothing more than an add-four block looped from the program counter register's output into a multiplexer (or "mux"), with the other input of this mux being the output of the ALU (representing the address to be jumped to in the case of a successful jump or branch). The output of this mux is fed back into the program counter register and is selected by a signal routed from the "execute" stage's controller by way of the "global" controller, which is high in the case of a realized jump or branch success in that stage (this is an indication that the next instruction loaded should be from the address now located at the ALU's output).

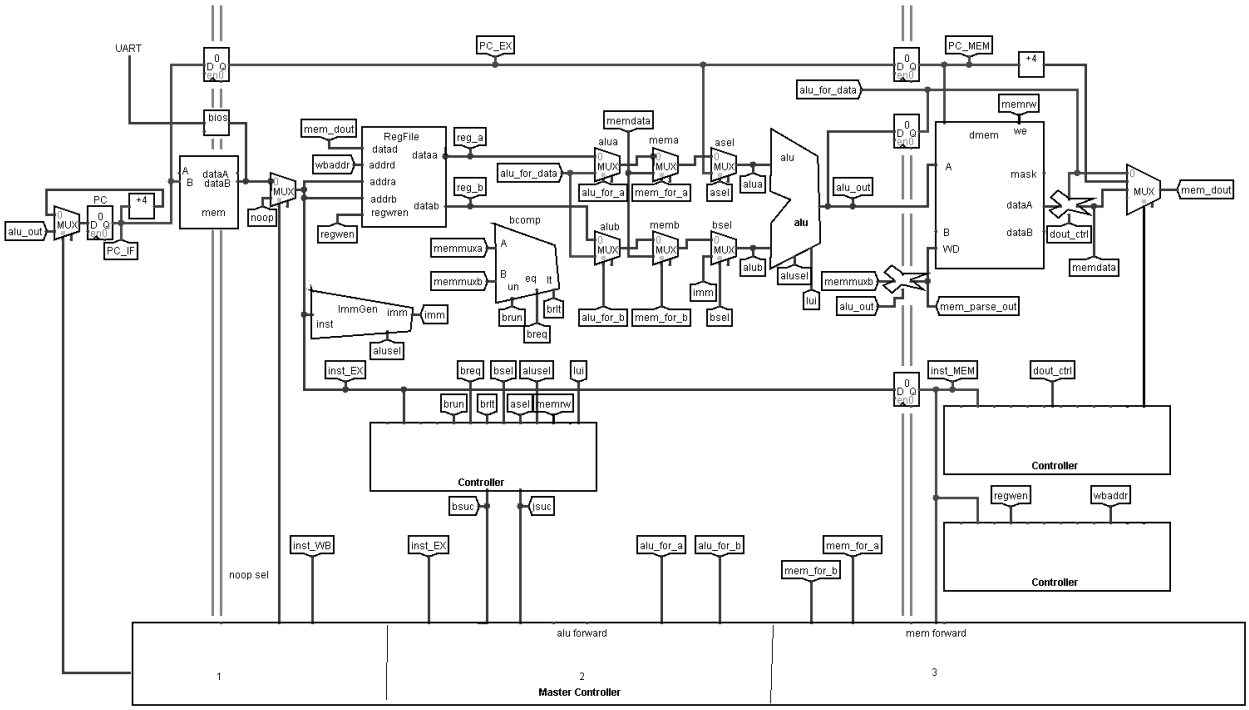

Figure 1: CPU Pipeline

Progressing into the "execute" stage, the current value of the program counter is provided to the (synchronous read and write) instruction memory abstraction (for the purposes of fetching the next instruction) and synchronized through an additional pipeline register for further use in the "execute" and "memory" stages. A mux then chooses between the resulting output of the instruction memory abstraction and a predefined `NOOP` signal (we used 32'b0, as this instruction results in nothing changing in our circuit), based on whether the provided instruction should be valid or not (this signal is called `noop_sel` in our code and is provided to the "execute" stage by the "global" controller). Then, with the instruction to be executed selected, it is used to retrieve register values from the register file and generate an immediate through an immediate generator module. This instruction is also fed through a register for later use in the "memory" stage. Following this, the register values yielded by the register file are filtered by a bank of muxes based on the type of instruction being executed and the possibility of forwarding scenarios presenting themselves. The "global" controller compares instructions in the "memory" and "execute" stages to identify data hazards in the `A` and `B` register positions and generates the control signals for the forwarding

muxes, which source data to forward from a synchronized version of the ALU output and the output of the data memory abstraction (which does not need to be synchronized further, as this is a synchronous read block). At the end of this cascade of muxes, the result of the forwarding muxes is provided to a branch comparator block, which evaluates whether a successful branch has occurred, and to the data memory abstraction, by way of the `memparse` block, which correctly aligns the data in the case of a "store byte" or "store halfword" operation. Here, control signals produced by the "execute" stage's controller allow two more muxes to select the inputs of the ALU (the `A` position chooses between the result of the `A` forwarding muxes and this instruction's corresponding program counter, and the `B` position chooses between the result of the `B` forwarding muxes and the immediate generated for this instruction). Then, the ALU operates on the inputs provided to it based on an `alusel` control signal from the "execute" controller and sends its output along to the data memory abstraction's address port, to be used in the next stage, and back to the first stage for the purposes of jumping or branching. As an additional note, we encapsulated input/output (or "I/O") functionality within our memory wrapper module, and were therefore able to sequester all peripheral circuitry to deal with high level inputs and outputs within this module (more on this in "Detailed Description of Sub-pieces").

Finally, within the "memory" stage, not much is done besides the selection of the value to be written back to the register file (through what is effectively a mux) and the determination of the register file address to go along with this data (and whether to enable this write or not). The data written back is chosen by the "memory" controller, via the `wbsel` signal, from the previous output of the ALU, the output of the data memory abstraction (realigned by the `memsnip` block), and the value of the program counter corresponding to this stage's instruction, incremented by four. Meanwhile, the "memory" controller (which is split into one block, `mem_controller`, which provides the `wbsel` signal, and another, `wb_controller`, which provides the write-back address and register file write-enable signal) prepares the relevant ports of the register file for a write operation (or a lack thereof).

3 Detailed Description of Sub-pieces

In our CPU, the memory wrapper is by far the most elegant, non-standard submodule. Our intent in designing the memory wrapper was to create a universal interface that the CPU could use to abstract away any notion of the different memory blocks, as well as the many I/O functionalities. A simplified block diagram can be seen below, in Figure 2. From a high-level perspective, all the memory wrapper needs to function are the I/O ports, the program counter, two addresses, and a data/write-enable pair. The two address port model was chosen so that the memory wrapper could be used for both the fetching of the next instruction (the function of the instruction memory abstraction) and accessing/writing to the memory in the pipeline (the function of the data memory abstraction) — this allowed us to differentiate which addresses and control signals were used in each stage, which keeps the individual memory interfaces from getting tangled. For example, there is no way a write operation could take place when an instruction is being fetched, so `data_in` and `write_enable` were chosen to be signals reserved for when the pipeline was using the memory wrapper for the purposes of writing to memory. Furthermore, we made sure to apply a check when writing to the `IMEM` block because it is the only block that can be used both as instruction memory and data memory in the pipeline. To ensure that the CPU could only write to the `IMEM` block if the `BIOS` was currently being executed, we checked the program counter for this condition. Another problem we faced was the allocation of the two address ports. We got around this by using the `addra` port for accessing data, and the `addrb` port for fetching instructions. This way, the memory

wrapper would only ever get one address per port even within a three-stage pipeline. With our dual address port memory wrapper being able to handle simultaneous reading and writing, we could safely encapsulate the three different memory blocks within this module without having collisions and were able to keep the interface very simple (small note: the `addra` and `addrb` inputs and outputs of the `BIOS` memory block look inverted in Figure 2 because we used Figure 3 in the project specification, or "spec", as a reference).

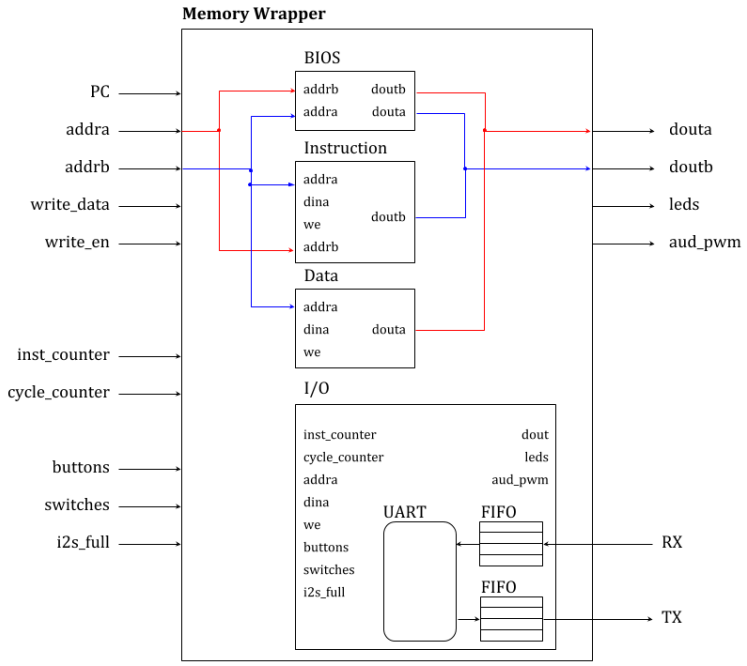

Figure 2: Memory Wrapper Internals

This implementation is made even more powerful by the ease with which memory-mapped interfaces can be added. Interfaces to UART, LEDs, buttons, switches, and audio were straightforwardly mapped by examination of the address provided to the port corresponding to the data memory abstraction. While it was not mentioned in the spec, we decided to implement transmission and receive FIFOs ("first in, first out" queues) for our UART to make it more robust to issues that might arise from frequent use. This proved useful to us when filling out implementation details, because we only had to look at FIFO `full` and `empty` signals to see if we could read or write to them.

As simple-to-use as our memory wrapper might sound, it is not without its flaws. The one subtlety our memory wrapper does not address is the fact that memory is byte-addressed in the RISC-V ISA. Our implementation of the memory wrapper did not initially handle operations on data smaller than one word (e.g. "load byte", or "store halfword"). To cover all of our bases, we created an address decoding (`memsnip` in our code) and write-enable encoding (`memparse` in our code) scheme that allowed us to continue to use the same memory wrapper without changing its internal circuitry. For any store instructions, we shifted the data by 0, 8, 16, or 24 bit positions, depending on the byte address, to properly align the data received by the memory wrapper with the masks of the memory IP blocks. Data coming out of the memory wrapper was decoded for load instructions by aligning its bottom-most bit (and sign-extending its top-most bit, if required)

to correspond to the instruction present in the "memory" stage of the pipeline. See Figure 3 for a visual representation of our address-parsing strategy.

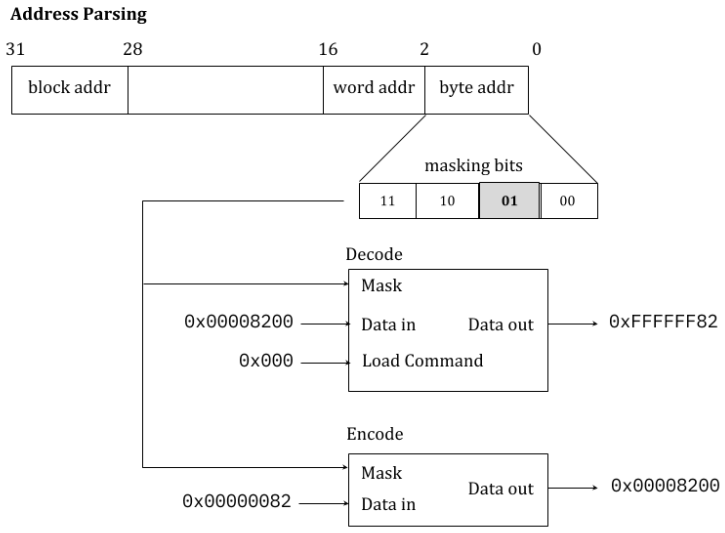

Figure 3: Address-Parsing Strategy for Loads and Stores

The other novel aspect of our design is our approach to pipelining and branch resolution. Specifically, the introduction of `NOOP` instructions during jump and branch operations is completed quite cleanly, and allows us to obtain what might be a theoretical minimum CPI for naive branch prediction. Because we partitioned the pipeline stages such that one stage contains all "execute" functionality, this "execute" stage knows immediately whether or not a branch is going to be taken, thanks to its branch comparator, which operates in parallel to the ALU. The branch comparator evaluates register values (that might have been forwarded) against each other through a "less-than" or "equal-to" comparison, and allows the "execute" controller to generate a "branch success" signal with this information. In the case that this "branch success" signal is high or a jump is being executed, the "global" controller kills the instruction following the branch or jump in the "execute" stage via the insertion of a `NOOP` and ensures that the next program counter value is the destination of the branch or jump instruction (as computed in the ALU). This allowed us to implement the full CPU without the need for any sort of stalling, with cycles only being wasted when a branch or jump is taken.

For our I2S and HDMI controllers, we followed the waveforms exactly to the respective datasheets, making sure that the clock signals produced by our controllers were correct at even the smallest scale. This meant that we had to take into account various rounding errors to ensure that the clock edges lined up perfectly. Aside from the need for this attention to detail, the implementation of these controllers and other additional peripheral circuits (like the FIFOs) was generally straightforward and only really required an in-depth knowledge of the communication protocol or principle in question.

4 Status and Results

On completion of this project, we can happily declare that our design was able to meet almost all design specifications and requirements. For the purposes of the first few checkpoints, our core design principles, which centered around developing a simple high-level design which would be robust to future changes, took us quite far (see Figure 4). Our initial choice of accepting a relatively long critical path in the "execute" stage of our design, in exchange for simplicity in branch resolution (with a "branch not taken" strategy), allowed us to achieve a fairly low CPI of 1.113 with a reasonable upper limit on our design's clock frequency (see Table 1). Furthermore, our decision to abstract our memory structure away within the memory wrapper module enabled us to seamlessly handle integration of I/O circuits for later project checkpoints.

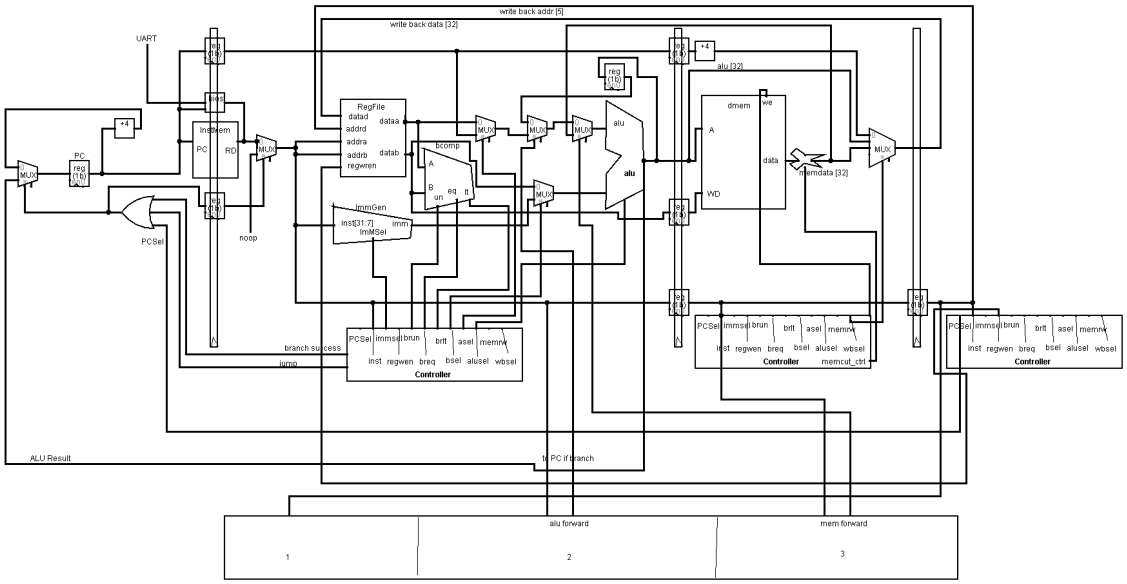

Figure 4: Initial CPU Design

Our peripherals all seemed to work decently well when running their respective test cases. However, this was not quite the case with our I2S controller. The testing of basic functionality, such as tone generation, went seamlessly, but, when testing output sine wave functionality using the `i2s_piano` test bench, we could not get our board to produce a clean tone. While trying to debug our problem, we put together another test bench, the `i2s_basic_piano` test bench, which sends corresponding frequencies as a square wave (as opposed to a sine wave). This worked fairly well, and we were able to produce clean sounds by typing on our keyboard.

The problem with the `i2s_piano` stuck with us when trying to run the `i2s_visual_piano`. The sounds were as noisy as before, but, coupled with that, the waveforms that were supposed to show on screen were also a noisy mess, reflecting the problem that was prevalent in the `i2s_piano`. In an effort to robustly test our HDMI controller and prove its functionality, we wrote an external script that could send a hand-drawn bitmap file from the user's computer to the CPU, which parsed it to draw on the screen. This worked surprisingly well through the slow UART, and we were able to draw on the screen through our CPU. With this functionality proved, we can say with confidence that our HDMI controller is working well, despite the `i2s_visual_piano` not displaying what we would like.

For the final checkpoint, involving optimization of our design, the main inhibitor to further

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	LUT Flip Flop Pairs (53200)	Block RAM Tile (140)
z1top	2223	1836	303	54	881	2207	16	314	36
b_parser (button_par...	52	67	0	0	33	52	0	29	0
comptutor (Riscv151)	2094	1608	296	54	811	2094	0	252	36
i2s_control (i2s_contr...	39	101	7	0	25	39	0	18	0
i2s_fifo (async_fifo)	38	60	0	0	18	22	16	14	0

Figure 5: FPGA Resource Utilization

optimization was essentially the critical path we had identified at the very beginning of our design process. While we decided against addressing this issue by modifying the pipeline structure of our design (as this would simply require too much additional effort for an uncertain gain in performance), we began tackling this problem by seeing how far our current design could take us (in terms of clock frequency, since CPI will remain the same for the same design) and iteratively improving it from there. On simulation at frequencies exceeding 68 MHz (we had initially been running our design at 50 MHz), we began to see possible setup time violations involving the path starting from the output of the instruction memory abstraction, passing through the "execute" stage and ALU, and ending at the program counter register. This path contains the entirety of the "execute" stage, with a bit more logic added, so it makes sense that it would be the constraining path of our system. To improve the delay of this path slightly, and push our clock frequency over 70 MHz, we identified that wire delay was a major contributor to the overall delay of this path and attempted to lower this by eliminating the use of `alusel` in our immediate generator and instead having this block determine the type of immediate to be generated on its own from the instruction (already provided to this block), by replicating some of the logic of `alusel` within the immediate generator. This change, as well as modifying the combinational logic involved in filtering the outputs of the register file (with respect to forwarded data) to use a `case` statement, instead of muxes implemented through behavioral Verilog, allowed us to scale our machine's workable clock frequency up to 71.875 MHz (with the same CPI, since our pipeline structure did not change). This resulted in a decent `mmult.c` execution time figure of merit of about 0.97 seconds, which is listed in the summary of performance metrics provided below in Table 1.

Table 1: Design Optimization Performance Metrics

Metric	Performance
`mmult.c` Instruction Count	62,599,914 instructions
Cycle Count for `mmult.c`	69,702,504 cycles
CPI for `mmult.c`	1.11346 cycles per instruction
Initial Clock Frequency	50 MHz
Initial Clock Period	20 nanoseconds
Initial `mmult.c` Execution Time	1.394 seconds
Optimized Clock Frequency	71.875 MHz
Optimized Clock Period	13.913 nanoseconds
Optimized `mmult.c` Execution time	0.9698 seconds

5 Conclusions

The biggest lesson we learned from this project would be the importance of a well-thought-out design. We took the first checkpoint of the project very seriously, and spent a lot of time thinking about whether or not our design would be sufficient for the specifications of the project. In hindsight, we should have spent more time thinking about not just the logic and timing requirements of the circuits we were proposing, but the constraints that we were imposing on ourselves by choosing said design. One notable such constraint included Vivado™'s synthesis and routing of our code, which was not always optimal and introduced extra wire delay which impacted our critical path, as described earlier. Though our design was streamlined, modular, and well-documented, this issue caused minor headaches down the line when we found that a majority of the delay on our design's critical path was coming from wire delay which could have been avoided by using fewer wires and fewer local modules.

If we were to do this project all over again, we would, for the most part, approach it quite similarly, as we feel we did a fairly decent job in conceptualizing our design. The one thing that, as a team, we did not do that well was reading and understanding the project design specification. As simple as it sounds, much of our confusion during the early stages of the project and resulting wrangling with Vivado™could have been avoided if we had just read the project outline more carefully.

One tip we would definitely stress to someone about to start the project would be the importance of learning to use debugging tools like ModelSim™, TCL scripts, Vivado™test benches, and C assembly dump files. Additionally, while we had ample experience using different GNU tools and understanding different scripting languages, these are skills not many have. The time we spent reading through and understanding how the debugging and simulation architecture was set up proved to be unimaginably useful. In the end, we were able to write basically any kind of test to thoroughly evaluate all parts of our project at any time, and to any level of precision. This helped us resolve our very last CPU bug, that was hidden deep within the ALU, through iterative testing of our design, starting from high-level tests, like a smaller matrix-multiplication program, and ending at low-level tests of specific functions (in this case, our design did not handle the generation of immediates for shift operations correctly).

There are two suggestions that we would like to make to improve the experience of this project. Firstly, we would like for the `Makefile` in the `software` directory to generate two different `.coe` files — one that is to be sent to the FPGA and one that is to be loaded into the IP blocks inside Vivado™with the proper offset (which can be pulled from the `.ld` file). The second suggestion would be a reorganization of the skeleton code to make sure that ModelSim™can properly look into the `frame_buffer` and `rgb2dvi` IP blocks (in the HDMI controller) so that students are not forced to use the Vivado™simulation.

Finally, we would like to express our sincere gratitude to our GSIs, George Alexandrov and Ali Moin, as they helped us through many a tough time this semester and made what would have been an otherwise painful experience remarkably enjoyable (even if it was still incredibly difficult). It is not an exaggeration to say that we were inspired to give this project our all by their collective demeanor and overall competence.

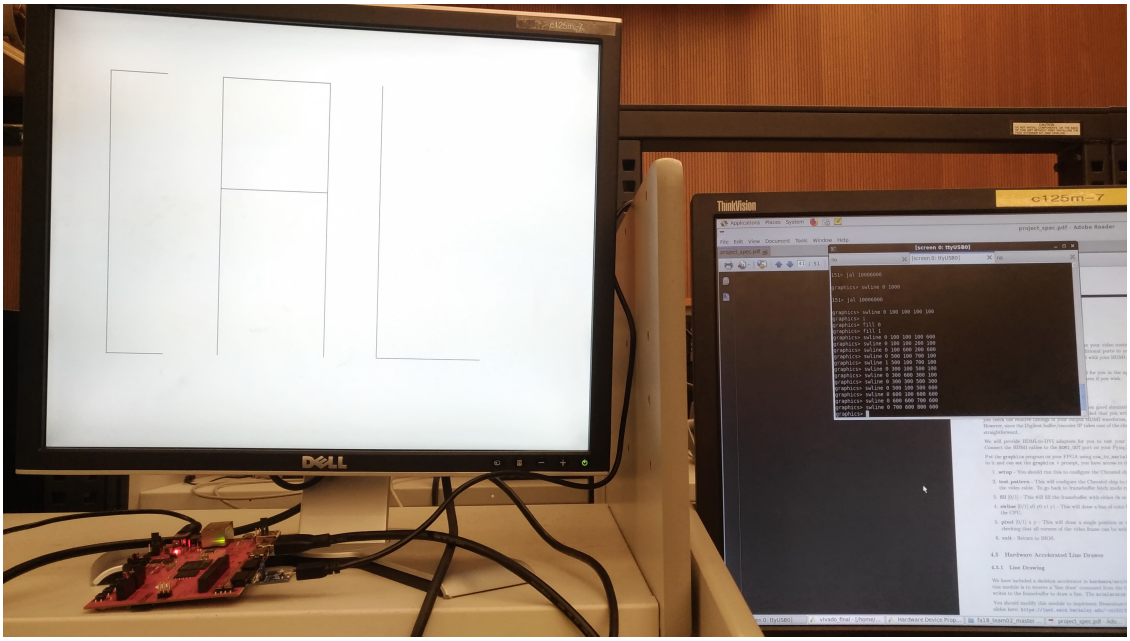

Figure 6: Our Design Showing Some School Spirit