EECS 151 Project: RISC-V CPU

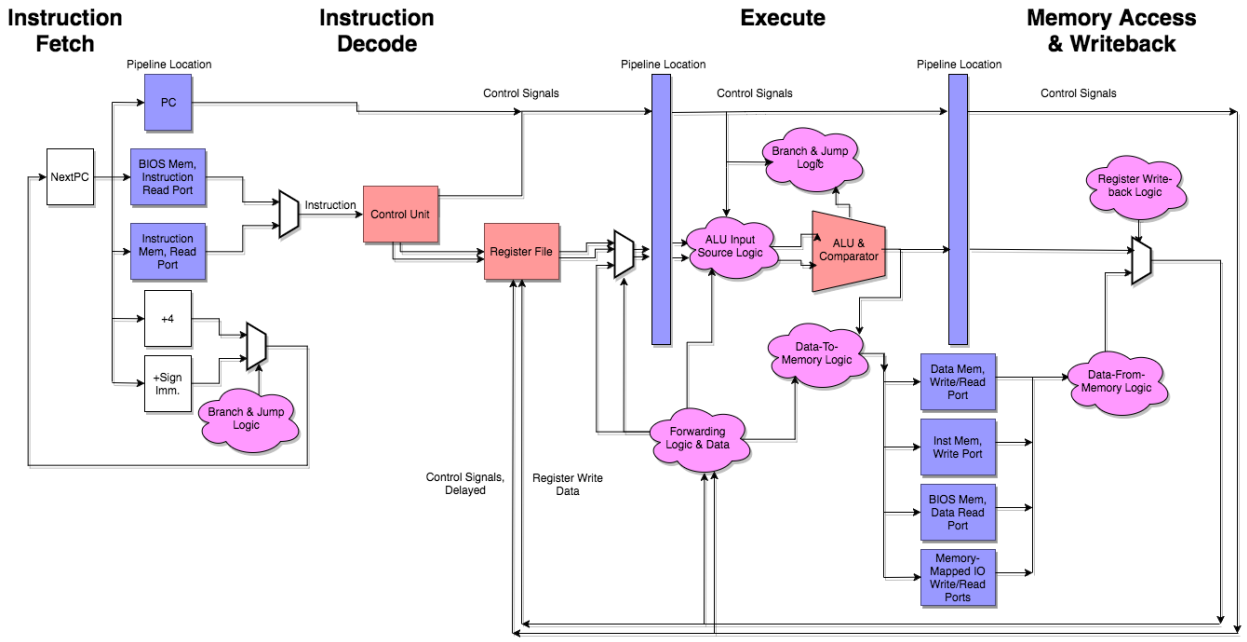
Allen Zeng

1 Project Functional Description and Design Requirements

The goal of the project is to implement a 3-stage RISC-V processor and peripheral circuits on the Xilinx Virtex-5 LXT ML505 FPGA. The target processor runs at above 50MHz. It is able to run a BIOS, load and execute programs, interact with user IO, output audio, and output video.

The implementation described here met those goals. Additionally, this implementation makes optimizations to the processor's clock speed and resources used on the FPGA.

2 High-level organization



A classic RISC pipeline can be modeled into 5 stages: Instruction (F)etch, Instruction (D)ecode, (E)xecute, (M)emory Access, (W)riteback.

There are three main parts to the CPU's memory hierarchy: the BIOS memory (BMEM), the Instruction memory (IMEM), and the Data memory (DMEM). Initially, the processor fetches instructions from the BMEM and executes a BIOS program. The BIOS is able to interact with user IO, write to IMEM, and write to DMEM. It is therefore able to store programs into the IMEM. The user is then able to instruct the CPU jump from the BIOS program to fetching and executing instructions stored in the IMEM. The DMEM can be used by any program to store data.

The BMEM, IMEM, and DMEM are implemented as Block RAMS on the FPGA. The Block RAMs are both synchronous write and read, so natural pipelining locations arise between the F-D stages and the E-M stages.

There is a degree of freedom about where to place the third pipelining location. This implementation places the separation between the D-E stages, parallel to the Register File's writing and after the Register File's reading. (The Register File is synchronous write, asynchronous read.) In the D and E stages, there is a long critical path through the Control Unit, Register File, and Arithmetic Logic Unit. So that location splits apart a large critical path. Additionally, this location was chosen because there is relatively little combinational logic in the M and W stages.

The memory-mapped IO technique is used to handle user IO. The CPU interfaces with IO modules such as the on-chip UART, Tone Generator, AC97 Audio Controller, I2C Controller, and DVI Controller.

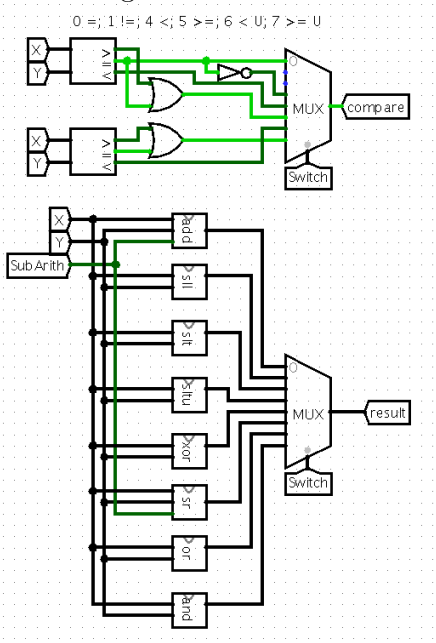
3 Detailed Description of Sub-pieces

CPU modules
The RISC-V CPU contains a Register File, Arithmetic Logic Unit (ALU), and Control Unit. There is no explicit Hazard Unit module, but it's logic is implemented on the CPU level in the appropriate sections.

The Register File is fairly standard. All registers are initialized to 0. Register `x0` is always set to 0. At each clock cycle, a write is performed if `write_enable` is true and `write_address` is not `x0`. There are two read ports, which are asynchronous.

The ALU contains a comparator portion and an arithmetic portion. The ALU takes in two inputs `x` and `y`, which are the operands; and inputs `switch` and `sa` (**S**ubtraction & **A**rithmetic Shift), which determines the operation performed. The `switch` corresponds to the `funct3` of the `instruction`.
The comparator makes signed and unsigned comparisons between `x` and `y`. It uses `switch` to determine which comparison result to mux to the output.
The arithmetic part applies all operations to the operands independently. It does the standard addition, subtraction, bit-shifting, and bit-wise operations. Then it uses `switch` and `sa` to determine which result to mux to the output. When adding, `sa` determines whether the operation is subtraction instead; and when shifting, `sa` determines whether the operation is arithmetic shifting.

Figure 1: ALU

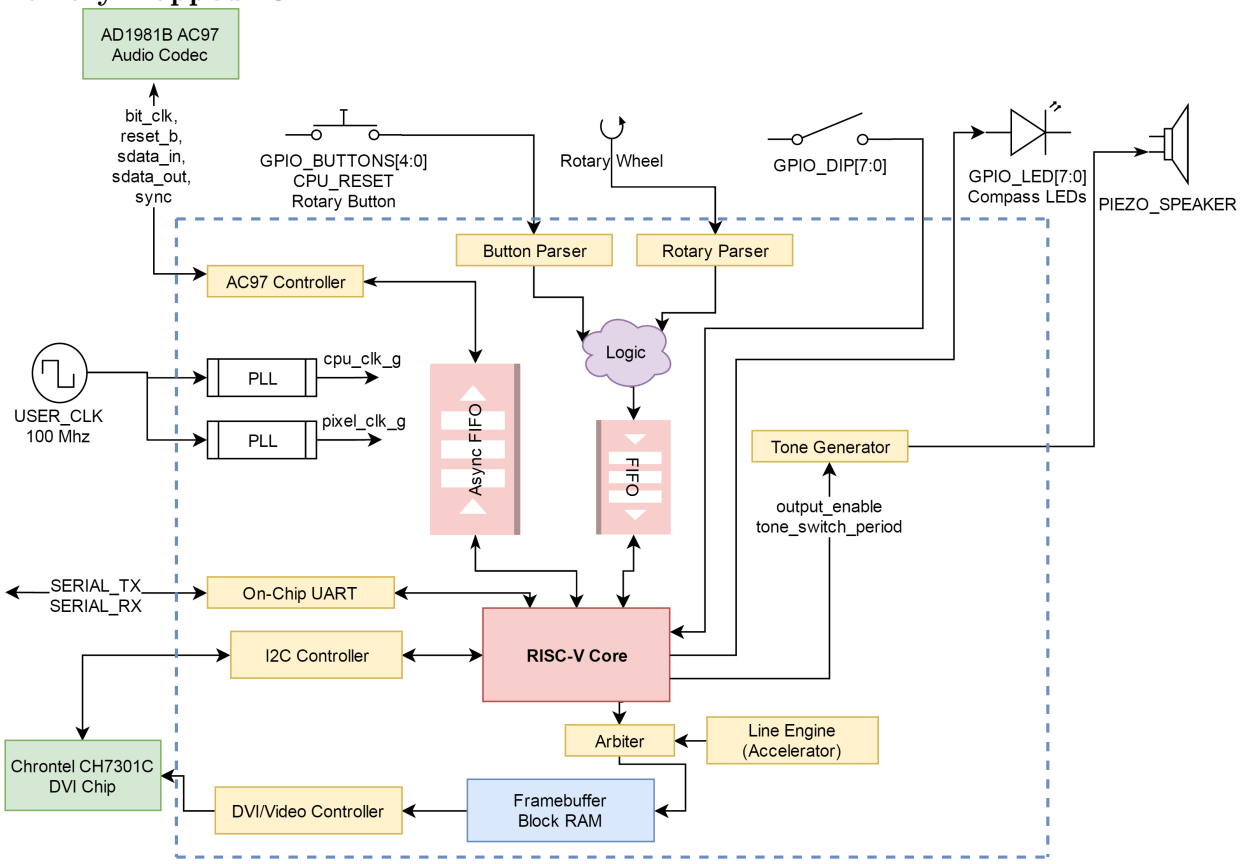


The Control Unit takes in an `instruction` and decodes it into many control signals. Most importantly, it determines the instruction type and constructs the appropriate Signed Immediate.

Due to the nature of the ISA, the bits of the immediate are spread into various locations of the instruction for different instruction types. For optimizing space, the traditional `shamt` signal for R-types is combined into being part of the Sign Immediate signals.
The Control Unit also determines the `aluSrcA` and `aluSrcB` signals. Those two signals determine the operands that are sent into the ALU. `aluSrcA` chooses among read-register 1's data (forwarded), read-register 2's data (forwarded), the corresponding `PC`, and 0. `aluSrcB` chooses between read-register 2's data (forwarded), and the Signed Immediate/`shamt`.

The Block RAMs used in the CPU were built using pre-existing Xilinx generators. The BMEM holds 2^{12} 32-bit words, and the IMEM and DMEM hold 2^{14} 32-bit words each. They are all synchronous read and write. This project's implementation does not require word and half accesses to be word or half aligned. But, accesses that are offset default to the closest lower alignment. For example, a store or load word at `0x4003`, `0x4002`, `0x4001` would default to accessing `0x4000`. And, for halfs, `0x4003` would access at `0x4002` and `0x4001` would access at `0x4000`. Bytes are aligned. Notably, the Block RAMs have an input port for write masking which is used in this implementation; but no read masking is provided, so there is a read mask operation in the CPU as part of load instructions. ("Data-from-memory logic" on the diagram.) The read masking implements signed and unsigned loading, and also has alignment behavior as described above.

Memory-mapped IO



The audio circuits and some of the IO circuits were carried over into the project from the lab with little changes. The I2C Controller was provided by the course staff, but the other circuits were implemented by the team in accordance to the project specifications.

The CPU interfaces with IO modules such as the on-chip UART, FIFOs, Tone Generator, AC97 Audio Controller, DVI Controller.

There is a serial line between the FPGA and the lab computer, and there are two UARTs one on either end that are used for communication. The on-chip UART interfaces with the CPU on the FPGA through a set of ready-valid handshaking signals. Data can be sent and received through the on-chip UART to the off-chip UART to lab computer. The on-chip UART's transmitter can take a byte of data written from the CPU and send it serially to the off-chip. Similarly, the on-chip UART's receiver can convert serially received data and present a byte to the CPU. Using the serial line, the FPGA can communicate with the lab computer. The baud rate is set to 115,200 in this project. The `screen $SERIALTTY 115200` is opened on the lab computer to talk to the FPGA. From there, `screen` can send ascii characters to the FPGA and can also receive and display characters. When running the BIOS program, the `screen` displays an input prompt and can run commands taken as input. For example, one can jump to another program, like `graphics`, and then subsequently `setup` the on-board Chrontel video-chip. Through the DVI Controller (below), the `graphics` program, and user commands on the `screen`, one can draw pixels to a video output.

The Tone Generator circuit switches an on-board Piezo speaker to produce mono sound. The tone generator takes an input half-period corresponding to certain audible frequencies. Every half-period clock cycles, the tone generator inverts its output signal in order to simulate a square wave. The square wave goes to the Piezo speaker to produce sounds.

As shown in the figure above, a FIFO sits between the CPU and the GPIO buttons. The buttons include the compass buttons and the rotary wheel. Button presses are parsed into single-cycle pulses and enter the FIFO. They are held until the CPU has time to process them. This way the CPU does not have to waste resources constantly watching for physical inputs.

An asynchronous FIFO is connected between the CPU and the AC97 Controller. The AC97 Controller operates on a Bit clock given from the on-board AC97 Audio Codec, separate from the CPU clock. So, the async FIFO takes in audio data written from the CPU and is read by the AC97 Controller.

Both FIFOS are implemented as a circular queue with a read and a write pointer. The FIFO is empty when the read pointer catches up with the write pointer, and the FIFO is full when the write pointer catches up with the read pointer. In order for the async FIFO to work across the two different clock domains, additional logic is implemented. The write pointer runs on the CPU clock and needs to be compared to the read pointer to generate the empty signal to the AC97 Controller. Similarly, the read pointer runs on the Bit clock and needs to be compared to the write pointer to generate the full signal to the CPU. To exchange these signals across clock domains, the pointers are converted from binary to gray code, synchronized, and converted back into binary in the other clock domain.

The AC97 Controller continuously reads audio data, in terms of tone frequencies (audio pitch), from the asynchronous FIFO and converts it to control signals sent serially to the AC97 Audio Codec. The Audio Codec plays notes through a headphone line. To control the Codec, the Controller sends 48,000 frames per second with 256 bits per frame. Each frame is synchronized using a sync signal and the contents are controlled by a tag at the beginning of the frame. The frame here is used to control master volume, headphone volume, PCM-out volume, and audio pitch.

The CPU is able to output video data through the DVI Controller and the Chrontel chip. This implementation uses the VGA protocol for timing, and connects to a 1024×768 60Hz refresh rate monitor through a DVI cable. The VGA frame is 1344×806 running on a Pixel clock of 65MHz. Synchronizing signals are asserted to start each video frame and horizontal line, and a data enable signal is asserted while the pixel clock is running through the visible region. While in the visible region, data from the video framebuffer is fetched and sent to the Chrontel chip through the data bus. Vertical and horizontal counters and combinational logic determine the synchronizing signals and the data signals. The video framebuffer is implemented as a Block RAM that is written to by the CPU. As the pixel clock runs through the visible region, data is fetched from the framebuffer and outputted as a pixel. Because the framebuffer is synchronous read, the data is fetched one cycle ahead to be in time for the data bus output. The `graphics` and `ac97_visual_piano` programs were able to draw to a monitor once the DVI Controller was implemented.

Table 1: Project PC and Data Addressing

PC Address [31:28]	**Function**	**Access**	
4'b0001	Instruction Memory	Read	
4'b0100	BIOS Memory	Read	
Data Address [31:28]	**Function**	**Access**	
4'b00x1	Data Memory	Read/Write	
4'b001x	Instruction Memory	Write	If PC[30]
4'b0100	BIOS Memory	Read	
Data Address	**Function**	**Access**	
32'h80000000	UART Control	Read	
32'h80000004	UART Receiver Data	Read	
32'h80000008	UART Transmitter Data	Write	
32'h80000010	Cycle Counter	Read	
32'h80000014	Instruction Counter	Read	
32'h80000018	Reset Counters to 0	Write	
32'h80000020	GPIO FIFO Empty	Read	
32'h80000024	GPIO FIFO Read Data	Read	
32'h80000028	DIP Switches	Read	
32'h80000030	GPIO Compass LEDs	Write	
32'h80000034	Tone Generator Output Enable	Write	
32'h80000038	Tone Generator Tone Switch Period	Write	
32'h80000040	AC97 FIFO Status	Read	
32'h80000044	AC97 FIFO Send Sample	Write	
32'h80000048	AC97 Volume Control	Write	
32'h80000100	I2C status	Read	
32'h80000104	I2C register read data	Read	
32'h80000108	I2C register address	Write	
32'h8000010c	I2C register write data	Write	
32'h80000110	I2C slave address	Write	
32'h80000114	I2C controller trigger	Write	
32'h900xxxxx	Video Frame Buffer	Write	

4 Status and Results

All of the basic functionality described in the project specification are implemented and are working. The `mmult` program completes with the correct checksum. The software-based `music_streamer` is able to parse the FPGA's button and rotary inputs, transition between states, and play music. The `graphics` program draws black-and-white pixels and lines to a 1024×728 VGA output. The AC97 Visual Piano program, `ac97_piano_visual`, integrates all parts of the project. It takes in keyboard inputs through the FPGA UART and converts them into notes. Audio notes are sent to the AC97 headphone output, and a visual representation of the note is sent to the VGA output.

Optimization Results

Originally, one of the worst critical paths occurred where the data that was coming out of the Memory stage (including IO data). The data went through a long chain of MUXes, the ternary operator `?`. Fortunately, that chain of MUXes was shortened into a more tree-like structure and the CPU frequency was pushed up to 80MHz. The new critical path was elsewhere in the CPU. The optimization seemed to work because the `mmult` program ran with the correct checksum. But actually, an IO bug was also introduced which caused unexpected behavior when running non-CPU-only programs. After fixing the new bug, the clock speed was lowered back to about 48MHz.

```
wire [31:0] data_from_memory_raw = (memWriteW2 && memToRegW && (addressW2 == addressW)) ? dinaW2
                                 :((partitionW == 4'b0001 || partitionW == 4'b0011) ? douta_dmem
                                 :((partitionW == 4'b0100) ? doutb_bmem
                                 :((addressW == 32'h80000100) ? {30'b0, i2c_rdata_valid, i2c_ctrl_ready}
                                 :((addressW == 32'h80000104) ? {24'b0, i2c_rdata}
                                 :((addressW == 32'h80000010) ? cycleCounter                    // For Memory IO, reading the counters
                                 :((addressW == 32'h80000014) ? instructionCounter              // For Memory IO, reading the counters
                                 :((addressW == 32'h80000000) ? {32'd0, uart_data_out_valid, uart_data_in_ready}
                                 :((addressW == 32'h80000004) ? {24'd0, uart_data_out}
                                 :((addressW == 32'h80000020) ? {31'b0, empty}
                                 :((addressW == 32'h80000024) ? {24'b0, button_c, button_n, button_e, button_s, button_w, rotary_push, rotary_event, rotary_left}
                                 :((addressW == 32'h80000028) ? {24'b0, gpio_dip[7:0]}
                                 :((addressW == 32'h80000040) ? {31'b0, ac97_fifo_full}
                                 : 32'd0 ))))))))))));

/* * * Old Code Above, New Code Below * * */

reg [31:0] data_from_memory_raw = 0;    // Declared as reg for below always, but acts as a wire
always @(*) begin
    if (memWriteW2 && memToRegW && (addressW2 == addressW)) data_from_memory_raw = dinaW2;
    else if ((partitionW == 4'b0001) || (partitionW == 4'b0011)) data_from_memory_raw = douta_dmem;
    else if (partitionW == 4'b0100) data_from_memory_raw = doutb_bmem;

    else if (addressW[31:12] == 20'h80000) beginif (addressW[11:4] == 8'h10) begin
        if (addressW[3:0] == 4'h0) data_from_memory_raw = {30'b0, i2c_rdata_valid, i2c_ctrl_ready};
            else if (addressW[3:0] == 4'h4) data_from_memory_raw = {24'b0, i2c_rdata};
            else data_from_memory_raw = 32'd0;
        end
        else if (addressW[11:8] == 4'h0) begin
            if (addressW[7:0] == 8'h00) data_from_memory_raw = {32'd0, uart_data_out_valid, uart_data_in_ready};
            else if (addressW[7:0] == 8'h04) data_from_memory_raw = {24'd0, uart_data_out};
            else if (addressW[7:0] == 8'h10) data_from_memory_raw = cycleCounter;
            else if (addressW[7:0] == 8'h14) data_from_memory_raw = instructionCounter;
            else if (addressW[7:0] == 8'h20) data_from_memory_raw = {31'b0, empty};
            else if (addressW[7:0] == 8'h24) data_from_memory_raw = {24'b0, button_c, button_n, button_e, button_s, button_w, rotary_push, rotary_event, rotary_left};
            else if (addressW[7:0] == 8'h28) data_from_memory_raw = {24'b0, gpio_dip[7:0]};
            else if (addressW[7:0] == 8'h40) data_from_memory_raw = {31'b0, ac97_fifo_full};
            else data_from_memory_raw = 32'd0;
        end
        else data_from_memory_raw = 32'd0;
    end

    else data_from_memory_raw = 32'd0;
end
```

The tree could be branched out further, but at this point the critical path is elsewhere. Note that because the I2C control signals were part of the old critical path, they are evaluated sooner in the new logic. Part of the bug that arose was because there were no `else data_from_memory_raw = 32'd0` in the inner if-statements. The wire was left unassigned, causing undefined signals.

I tried to further optimize the CPU, and thought I would be able to push to 60MHz. (The Placement-and-Routing report stated it reached 60MHz with no timing violation.) But then the optimizations turned out to introduce some bugs. I then fixed the bugs, but the clock dropped

back down to 52MHz.

Without a way to further reduce the critical path except for adding in more pipeline stages (thereby redesigning major components of the CPU), I decided to optimize the existing design for area and resource usage instead. Reducing resource usage mostly involved making small changes and simplifying combinational logic. Mostly notably, the traditional R-type shifting instruction's `shamt` signal was combined into the other instruction's Signed Immediate logic.

The current design runs at 52.054MHz. Without more pipelining stages, this is to be the fastest clock speed I could reach. All checkpoints run at that clock speed, because the critical path goes from the Write stage of the datapath back to the Execute stage and through the ALU. Because the critical path of the entire project is part of the CPU, this limits the system's overall clock speed. The peripherals implemented in Checkpoint 2 and 3 are not part of that path.

For `mmult`, the cycle count is `0427936a` and the instruction count is `03bb32eb` for a CPI of 1.1135.

Device Utilization Summary on the next page.

Device Utilization Summary

Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	2,255	69,120	3%
Number used as Flip Flops	2,255		
Number of Slice LUTs	3,869	69,120	5%
Number used as logic	3,810	69,120	5%
Number using O6 output only	3,442		
Number using O5 output only	229		
Number using O5 and O6	139		
Number used as Memory	37	17,920	1%
Number used as Dual Port RAM	24		
Number using O6 output only	8		
Number using O5 and O6	16		
Number used as Shift Register	13		
Number using O6 output only	13		
Number used as exclusive route-thru	22		
Number of route-thrus	251		
Number using O6 output only	250		
Number using O5 and O6	1		
Number of occupied Slices	1,684	17,280	9%
Number of LUT Flip Flop pairs used	4,962		
Number with an unused Flip Flop	2,707	4,962	54%
Number with an unused LUT	1,093	4,962	22%
Number of fully used LUT-FF pairs	1,162	4,962	23%
Number of unique control sets	116		
Number of slice register sites lost to control set restrictions	164	69,120	1%
Number of bonded IOBs	59	640	9%
Number of LOCed IOBs	59	59	100%
IOB Flip Flops	17		
Number of BlockRAM/FIFO	60	148	40%
Number using BlockRAM only	60		
Number of 36k BlockRAM used	60		
Total Memory used (KB)	2,160	5,328	40%
Number of BUFG/BUFGCTRLs	3	32	9%
Number used as BUFGs	3		
Number of PLL_ADVs	2	6	33%
Average Fanout of Non-Clock Nets	4.68		

5 Conclusions

This project has been a great learning experience.

Looking back, there are were some things that could have been done differently. When implementing Checkpoint 1, I implemented the basic functionality of the CPU with the Control Unit, Register File, ALU, BMEM, IMEM, and DMEM. I wrote basic assembly tests to test all instructions independently and dependently, and attempted to move on to IO modules. Once I implemented the IO modules, I encountered many strange bugs in trying to run the BIOS and programs. It turned out that although I believed our assembly tests were comprehensive, they lacked some edge cases. I was especially stuck on the data-hazards associated with load-then-store and store-then-load operations.

Additionally, there was a timing error in our DVI Controller's testbench. I had implemented the DVI Controller's timing both combinationally and as a state machine. It turned out neither was working well because the testbench's timing resolution was not precise enough. The resolution caused the testbench waveforms to round the signal lengths, causing the timing model to output errors. After increasing the testbench's resolution, the work-in-progress combinational implementation passed the testbench. I kept that implementation because it was working, and also it uses less hardware than the state machines.

Aside from those difficult bugs, the rest of the project was fairly straightforward.

I wanted to extend the CPU to be a 5-stage processor. But unfortunately there was not enough time to do so since the semester is over and the project deadline is approaching. Given the chance to redo the project, I should have planned for a 5-stage to begin with. Now that the 3-stage is implemented, transferring it to a 5-stage processor is fairly involved. New data-hazards would arise and need to be resolved. But, I would be able to reuse the (now comprehensive) assembly testbench that was created for the 3-stage processor. Certainly if there were more time in the semester, I would have extended the project and implemented extra functionality.