**Introduction**

During the Spring 2023 school semester, I took my first Computer Architecture class, CprE 381, at Iowa State University. In this class, we learned how to convert basic programs into assembly, design and analyze a MIPS processor, and compare different cache designs. In the lab portion of the class, we focused on implementing three different MIPS processors using VHDL and ModelSim, including a single cycle, 5 stage software pipeline, and 5 stage hardware pipeline processor. We started with the single cycle processor, and designed modules to increment the program counter, decode instructions, a register file, a sign extender, and an ALU. For the 5 stage processors, we broke apart each of these components into five separate stages, so that we could reduce the critical path latency and improve the maximum clock rate of our design. Near the end of the project, me and my partner had about a month of time left. Due to this, we decided to work on an extra credit project through the form of an FPGA wrapper with the Altera DE2 FPGA Development board. For more information on each of these designs, please look at the other tabs for this project.

The goal of this project was to learn more about implementing specific components of a processor, and analyzing the performance tradeoffs between each of our three processors. For the software pipeline, we inserted NOP instructions to remove the risk of data and control dependencies within our pipeline. This led to an increased number of instructions, which made our overall execution time larger. As we implemented stalling in the pipeline registers of our hardware pipeline, we were able to reduce the number of instructions used, with the tradeoff of our CPI increasing from a near average of 1. After adding forwarding we were able to reduce the average CPI of our hardware pipeline while retaining a faster maximum clock frequency compared to our single cycle design. By analyzing these choices, we were able to make educated design decisions on how to improve our HW pipeline design, and yield an improved performance compared to our first two processor designs.

Another goal of the project was to take ownership in our required work by taking it a step further with an extra credit project. As mentioned before, we had extra time near the end of the semester, so me and my partner Thomas worked on designing an FPGA wrapper for the Altera DE2 FPGA development board. With this, we were able to combine Verilog code from our previous digital design class, that I also was a teaching assistant for, and a mix of FPGA modules, from the previous library I designed, to make a robust wrapper. It was very satisfying to combine designs for clock dividers, button debouncers, and seven segment display interfaces that had already been extensively tested, and apply them to an interesting and challenging MIPS processor design. This was also a fantastic goal and achievement since we were finally able to see our processor run on real hardware, and not just the required simulations for the course and lab. For more information, please reference the FPGA tab on this page.

**Single Cycle Processor**

The first main project that we completed in CprE 381 was our MIPS single cycle processor. This processor would take 32-bit instructions, and was able to decode multiple R-type, I-type, and J-type instructions, including ALU arithmetic operations, conditional branches, unconditional branches, and memory operations. A full list of the 33 instructions and their respective decoded control signals can be seen below in the Single Cycle Controls spreadsheet. To implement this processor, we designed an ALU, register file, control decode, and sign extender. We were provided instantiated RAM modules to act as memory to interface with the provided testing toolflow. Using the open-source MIPS simulator MARS, and simulations from Quartus Prime and ModelSim, we would be able to load assembly instructions into our processor and verify expected behavior every clock cycle.

Each of our designed modules were written with VHDL, with a combination of structural, dataflow, and behavioral models. All of our code was managed with revision control by using Git, and we installed a VHDL plugin to use with VS Code as our text editor. As mentioned before, we used the open-source MIPS ISA Simulator MARS to simulate and test the assembly programs we would design to later test on our single cycle processor.

During the previous labs before the project, we were tasked with implementing a register file and basic ALU. The ALU could take an add/sub control, and we also included a 32 bit 2x1 multiplexor to choose between the contents of a second register or an extended immediate value, to dictate different ALU instructions between R-type and I-type. Since these were completed already, the main tasks in the single cycle processor project were to create a more integrated ALU, an instruction decode module, and a program incrementor module. The modules that I worked on were both the instruction decode and program counter incrementor modules, while my teammate Thomas worked on including additional functionality for our ALU, based on the added instructions.

The first module I worked on designing was our instruction decode module. This module would take in the upper 6 bits of each instruction fetched from instruction memory to determine what instruction we would run. If the opcode was a 0, we also were required to read in the function of the instruction, which was the 6 lowest bits of R-type instructions. Finally, we needed to read the RT address to identify certain branch instructions, including bgez and bltz. By using a process statement with these three inputs in the sensitivity list, I was able to create a branching case statement based on the opcode, then potentially reading the function or RT address depending on the opcode. Once we knew what the decoded instruction was, we were able to properly determine the control signals for each instruction for the ALU, data memory, and register file. The specific controls listed were created in a spreadsheet to manage better, and can be seen at the bottom of this page. Since I designed this module, it was my partner Thomas’s responsibility to test it with a VHDL testbench. Expected outputs and waveform results can be seen in the Single Cycle Report below.

The next module I designed was the fetch module, to appropriately update the program counter for a following instruction, conditional branches, and unconditional branches. We began by designing a register to hold the program counter, which was a 32 bit value that could be asynchronously reset and included a write enable bit. The fetch module would take in an input from the decoded control module to multiplex between a PC + 4 address, branch address, or jump address, which were all calculated separately based on the requirements of the MIPS ISA. Other inputs included the jump address and branch determination to handle both unconditional and conditional branches. As before, Thomas was responsible with testing this module.

After Thomas was done completing the ALU, it was my responsibility to test it! This was an awesome opportunity to test something that I had not designed, which I got lots of practice from on my co-op as a Systems Engineer at Collins Aerospace. Our ALU would take in two inputs to use as arithmetic operands, which could be received either from our register file or as a 16 bit extended immediate value. Depending on what type of ALU instruction we had, the immediate could be extended as either sign-extended or zero-extended. For example, ADDI instructions were sign-extended but logic instructions like ANDI were zero-extended. We used more control signals to act as a select line for a multiplexer between each of our ALU submodules to dictate the correct output. Each unit under test inside of the ALU included branch determination, an adder, logic operations, and a shift module. Each of these modules earned their own testbench, which included error flags and automated error checking based on the inputs and expected outputs. We also were able to create a custom .DO file for ModelSim to automate compiling our source files, adding waveforms, and fitting the screen to them all. We even figured out how to color code them to make viewing the waveforms easier for our TA.

After all of our individual modules were tested, we were ready to wire them up and instantiate them together in a top-level processor module. We would include each of our designed modules from the previous sections and in our first 2 labs, alongside the provided memory module for the instruction and data memory. The largest challenge was keeping track of all of the internal signals, since this was the most involved digital design module me and my partner had designed up to now. To help with this, we designed a top-level schematic connecting each module, and specifically labeled each signal on that schematic. This was especially useful since we could then reference this schematic to determine what signals were left to connect. After connecting our processor, we were ready to begin simulating assembly programs.

To test our processor, we would simulate assembly programs to run code for a Fibonacci sequence of bubble sort. Alongside this, we were provided unit cases and other tests to verify the robustness of our design. During all of these tests, we were able to debug and verify the functionality of ALU operations, control flow, and memory operations. It was crucial to ensure that instructions such as JAL and JR would function correctly, since instructions like these required additional hardware to multiplex inputs to the register file. It was also especially helpful to have another custom DO file to automatically load the generated waveform from our toolflow and add in all of the relevant waveforms, including but not limited to the target read and write registers, ALU output, and program counter. Connecting and verifying each of these modules gave a wholistic view on computer architecture and allowed for more complexity that I was looking for in my first digital design class. Next, we were ready to begin designing our first multistage pipelined design.

**5 Stage Processor**

SW Processor

Next, we were able to begin designing our 5-stage software implemented processor. As we learned in class, we are able to prevent data and control hazards always by stalling our processor. To get more familiar with designing the other components of our 5-stage processor, we would begin with a software implementation where we would insert NOP instructions directly into our assembly programs to prevent the aforementioned hazards from occurring. With the software implementation, this would then increase the overall number of instructions ran in the program, while keeping a CPI near 1, and ideally reducing the cycle time due to breaking up our components between five stages.

At the start of the project, we worked on designing the pipeline register files that would be placed between each of the 5 pipeline stages, which included:

* Instruction Fetch (IF)
* Instruction Decode (ID)
* Execute (EX)
* Data Memory (DMEM)
* Write back (WB)

Each of these stages would require at least some of the previous stages data flow and control flow signals. For example, the decoded controls in the ID stage would need to be propagated through EX, DMEM, and WB to ensure that the register write control bit is sent through our pipeline and is written on the correct cycle and instruction. For a more detailed list of these controls and which signals propagate through, please refer to the 5 Stage SW Controls spreadsheet. For each of the propagated signals, we would use a synchronous vector of D flip flops for each signal with a write enable and asynchronous reset. We decided to include a write enable so that we could eventually use it to control stalling for our hardware pipeline. In total, we designed four pipeline registers, that would be placed between the stages IF/ID, ID/EX, EX/DMEM, and DMEM/WB. With this complete, we were now ready to update other components to reduce the number of NOP instructions required.

Read after write register file (max NOP for data hazards from 3 to 2)

Moving conditional to decode stage (1 less NOP)

HW Processor

**FPGA Wrapper**