**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.

While we did not implement stalling, flushing, or forwarding in our SW pipeline, we were still able to use a few techniques to reduce the number of NOP instructions (and stalls for the HW pipeline). We first solved the problem of a read after write data dependency between the ID and WB stage. For example, if a register was going to be written to and was currently in the WB stage, it would only get updated after the following positive edge of the clock cycle. This meant that the instruction in the ID stage would be unable to properly fetch the correct value, leading to a data hazard and an additional NOP. So, by adding a comparison to see if the write address was the same as one of the read addresses, alongside ensuring the register write control bit was 1, we could multiplex the contents of the write register before it has been written to the register file. This was useful since it reduced our maximum number of read after write data dependency NOPs/stalls from 3 to 2 for consecutive instructions.

We also decided to move our branch determination for conditional branches from our ALU into a separate module in the ID stage. This would allow us to reduce the number of NOP instructions from 2 to 1 for unconditional branches, including instructions such as beq, bne, bgtz, and so on. Since the instruction was decoded in the ID stage, unconditional branches such as jump instructions would also require 1 NOP or stall, making both types of branching consistent. With both of these implementations, we were able to reduce both data hazards and control hazards by 1 NOP instruction, effectively reducing the instruction count for most types of instructions. With this, we would reach an improved execution time by also taking advantage of the multistage pipeline giving us a faster maximum clock frequency.

Testing the SW pipeline introduced its own challenges since we first had to update our previous assembly programs with the minimum number of NOPs depending on the control and data hazards. Every branch and jump instruction would require only 1 NOP, and data hazards could lead to either 1 or 2 NOPs, depending on if there was an instruction without a dependency between them. Debugging our waveforms became challenging since there could be up to 5 different instructions in our pipeline at the same time, which made it essential to trace through signals such as the program counter, register write address, and ALU output. There were many signals that had to be propagated through different stages, so it was crucial to identify which instruction was in what stage of the pipeline for every clock cycle. When we ran into bugs, we would develop a new assembly program and attempt to dilute the issue to the minimum number of instructions to easily identify and solve the issue. It was very rewarding to get to this point, since debugging our multistage pipeline processors has been some of the most interesting and rewarding problem solving, I have worked on in any of my classwork so far. With our software pipeline done, we were now ready to add hardware implementation for stalling, flushing, and forwarding!

HW Processor

Our hardware processor introduced new challenges for our design, but allowed us to improve our execution time even more by reducing the number of instructions by removing the NOP instructions and introducing forwarding and branch prediction. Before that though, we were required to design a hazard control module to determine when stalling would be needed, to imitate the functionality of the previously used NOP instructions. Alongside this, we had to update our pipeline registers to include a stall control, to delay writing to the register, and a flush control, to write 0’s for all of the bits in the pipeline register to wipe away the instruction due to a control hazard.

To implement stalling, we began by updating our pipeline register. We used the previously mentioned enable bit that was always set to 1 for the D flip flops in our pipeline register, and updated that to be driven by a stall control bit. If the stall control bit was 0, then the contents of the data input would be written to the pipeline register, which included data flow and control flow signals. To flush our pipeline, which would be needed to handle control hazards, we included a N-bit 2x1 MUX for each signal in the pipeline register, with the flush control as the select line. If the flush control select bit was 0, we would MUX the standard input to the pipeline from the previous stage. If the flush bit was 1, we would instead MUX 0’s in for each signal in the pipeline register, effectively clearing away the instruction in the previous stage. This was needed to handle control hazards since if we were buffering a branch no taken pipeline, and we ended up taking the branch in the ID stage, we would need to wipe away what was previously expected to be used that was fetched in the IF stage!

With our pipeline registers updated, we were now ready to create the hazard detection module that would drive the stall and flush control signals that would go to each of our four pipeline registers. The structure of our hazard detection module was very similar to our control decode module, in that it took in the ID stage opcode and function, alongside multiple register write addresses and control bits from the EX and DMEM stages, to determine if a hazard had occurred. In general, a data hazard could occur if the instruction in the ID stage was attempting to read from a register that has not yet been written, which is currently ahead in the pipeline in the EX or DMEM stage. A control hazard was determined by reading the opcode and function fields of the 32-bit instruction, to determine a stall initially. If a data hazard occurred, we would stall the IF/ID pipeline register and program counter register, alongside flushing the ID/EX pipeline. This would ensure the instruction in the ID stage would not get passed into the EX stage, and no new instruction would be fetched in the IF stage by stalling the program counter and IF/ID register. Control hazards only had to flush the IF/ID pipeline register since we moved the conditional branch module into the ID stage, leading to a delay of one cycle for both branches and jumps. With this complete, we were now ready to begin enhancing our hardware pipeline with forwarding and branch prediction.

To reduce the number of stall cycles needed for data hazards, we designed a forwarding module that would be placed across the ID, EX, and DMEM stages. We decided to forward three different locations, including both ALU inputs and the DMEM data input. The forwarding cases were determined the same way that they were for the hazard detection module, but we now added extra multiplexors in front of both ALU inputs and the data memory module to MUX between the normal pipeline datapath and the forwarded contents from an instruction in a further stage. As we implemented these forwarding paths, we turned updated our hazard detection module so that we would not stall the pipeline when we could instead now forward it. To reduce the number of flushes for our branch instructions, we included a check for branch not taken in our hazard detection. Instead of always flushing our conditional branches for both taken and not taken, we included a check so that we would not flush the instruction in the IF stage if a branch was not taken, since it would be the proper PC + 4 instruction, instead of at the branch address. We could have made a more complicated predication unit to reduce the CPI of branch instructions, but this was a beneficial way to reduce flushes without extra hardware.

After we tested and completed our hardware processor, we completed a performance analysis that compared each of our three processor’s instruction count, CPI, and maximum clock frequency so that we could find the execution time of each respective processor. We were able to analyze the positives and negatives of each processor, and how we could improve them more based on a standardized set of assembly programs. For example, the critical path and maximum clock frequency of our single cycle processor was much lower than our pipelined processors, so adding more hardware to our critical path would be a huge detriment to the single cycle’s execution time. We could have also added another stage to handle the load of some of the hardware between the ID and EX stages to increase the maximum clock rate of our hardware pipeline. Also, for the hardware pipeline, we could have added more forwarding paths for the branch conditional in ID stage, reducing the number of stall cycles for conditional branch instructions. A more detailed report of this analysis can be seen below in the list of the provided documents.

**FPGA Wrapper**

By the time we completed our hardware pipeline and performance analysis, we still had around a month of the course left before the assignments were due. With this extra time, me and my partner decided to work on an extra credit project in the form of an FPGA wrapper. We used the Altera DE2-115 FPGA Development board, which is the FPGA board used for the first digital logic class that I was a teaching assistant for in Spring 2021. The goal of this project was to take advantage of multiple I/O modules that me and my partner had previously designed for other projects and apply them to a wrapper to synthesize our hardware pipeline processor. Since we were using an Intel FPGA, we used Quartus Prime to synthesize and configure our FPGA.

The first thing we set out to do was employ multiple different I/O modules to get our processor running on an FPGA. This included a pushbutton debouncer and seven segment display decoder. The DE2 board that we were using had 8 seven segment displays, meaning that we could display the full 32-bit values in our data flow, including the ALU output, data memory output, and fetched instructions in our pipeline. We used the debounce buttons to have smoother signals for our asynchronous processor reset and a manual clock. We decided to have the option for a manual clock so that we could increment the clock cycles at our own rate to verify the correct values were flowing through our pipeline. We also included a clock divider module, so that we could divide the on board 50 MHz clock down to 25 MHz, to satisfy the maximum clock constraint of our hardware pipeline, and down to 5 Hz, so that we could still read the outputs on the seven-segment display. We also added a counter to verify the amount of clock cycles, and added that as a MUX output to the seven-segment display. Finally, we included a latch to the clock signal so that when the final instruction was output in our processor pipeline, we would stop from running the program again or fetching anymore instructions from data memory. Overall, each of these modules helped to provide a useful interface with our previously simulated 5-stage hardware pipelined MIPS processor.

The next challenge for our wrapper was in figuring out how to properly initialize the instruction and data memory contents from our memory modules. Quartus Prime has a feature where you can designate the name of a Intel Hex file for the initial loading, which will be done upon reconfiguring the FPGA when loading the bitstream file through Quartus Prime. So, we decided to split the memory module into two separate modules, one for the instruction memory and one for the data memory, and specify two different Intel Hex files imem.hex and dmem.hex, respectively. We were able to generate these intel hex files from the MIPS ISA simulator MARS, by taking advantage of our previously used assembly programs for our simulation verifications. We learned that to reset the memory we had to fully reconfigure the FPGA board, which gave us issues after testing the data memory, since the initial contents would be written over through the program. After this was done, we were able to verify the same tests ran in our pipeline simulations on our FPGA, and verify the final clock cycle count, ALU output, and memory accesses both for the instruction memory and data memory. This was a very rewarding experience and a great way to apply my knowledge of the course and expand on my FPGA skills with another set of tools.