Tyler Curtis A20304662
Caleb Seabolt A20343732

Lab 2: Single-Cycle RISC-V Microarchitecture Implementation in HDL

Section 1: Introduction

The main goal of this lab is to develop a deeper understanding of a Single-cycle Risc-V processor. This will give us the base knowledge to understand more complicated architectures like multi-cycle or pipelined. To start we are given the working code for a limited instruction set for a single-cycle RISC-V Microarchitecture. The objective of this lab is to implement additional instructions to this architecture by expanding upon the given code. Below is a table with all instructions, those highlighted are already implemented.

Thus far all labs completed in this course have built upon each other. Lab zero consisted of a review of digital logic that led to creating a register file. This not only reminded us what a register file is but it gave us a deeper understanding of what is actually going on inside. This was a good segway into lab 1 which was an instruction-level simulation for this same RISC-V single-cycle CPU. Implementing the instruction required a high level of understanding of each instruction, and that is where this lab really picks up. Not only do we need to understand what each instruction does. We need to understand how it interacts within the control unit and the Data path. Currently, in the lecture we are learning about Multi-cycle CPUs, and pipelined CPUs, being able to understand how these instructions are implemented within these different sections is crucial in understanding these more advanced architectures.

add	addi	andi	and	auipc
beq	bge	bgeu	blt	bltu
bne	jalr	jal	lb	lbu
lh	lhu	lui	lw	or
ori	sb	sh	sll	slli
slt	slti	sltiu	sltu	sra
srai	srl	srli	sub	sw
xor	xori			

Section 2: Baseline Design

The first step in adding additional instructions to the architecture is understanding how each instruction flows through the control unit, and data path. This allowed us to make the essential expansions to allow for more complex instructions, or to add them into the existing code simply. For example, many (if not all) of the R-type instructions were already implemented correctly in the ALU. Which came with additional support from the get-go. We were just left with adding the control unit logic.

However other instructions could just simply be added logically. We needed to expand to include them. One of the first steps in this process was increasing the bits for the ALU control bits from 2 to 3. This allows us to add more operations that require a more complicated ALU arithmetic. Other processes needed the implementation of various muxes to perform as expected. Our objective when programming all of this was to start with the easier instructions and progress through to the more complicated ones. Understanding how each instruction worked in the control unit, and the crucial path it took through the data pata made implementing these instructions as simple as inputting them and expanding where needed. Once implemented all that was left to do was test them and confirm the correct working order.

Section 3: Detailed Design

Our processor design starts inside our “riscv_single.sv” file and it starts by creating our testbench module design(which will be instantiated later) which uses 2, 32-bit variables(WriteData and DataAdr) and 3 single-bit variables(clk, reset, and MemWrite). Inside of this module, it instantiates a top module(explained later) called “dut” and passes it: clk, reset, WriteData, DataAdr, and MemWrite in that order. It then loads data into the imem and dmem modules(explained later) from memfiles and beings iteriting through the commands in imem based on clockcycles. We then define our riscvsingle module design by passing it single bit variables: clk and reset as inputs, 32-bit variables Instr and ReadData as inputs, 32-bit variables PC, ALUResult, and WriteData as ouptus, and MemWrite as a single bit output.

Inside this module we instantiated our controller and our datapath modules passing them Instr[6:0] as the opcode, Instr[14:12] as funct 3, Instr[30] as Zero, and branchflags, ResultSrc, MemWrite, PCSrc, ALUSrc, RegWrite, Jump, LoadSRC, ImmSrc, and ALUControl as themselves(for controller), and WriteData as WriteData1, and clk, reset, ResultSrc, PCSrc, ALUSrc, RegWrite, ImmSrc, LoadSRC, ALUControl, Zero, branchflags, PC, Instr, ALUResult, and ReadData all as themselves(for datapath) connecting all of the shared outputs to the shared inputs. We repeated this process for every module we declared following the pattern given to us on page 407 of the textbook(see below) with a few minor changes.

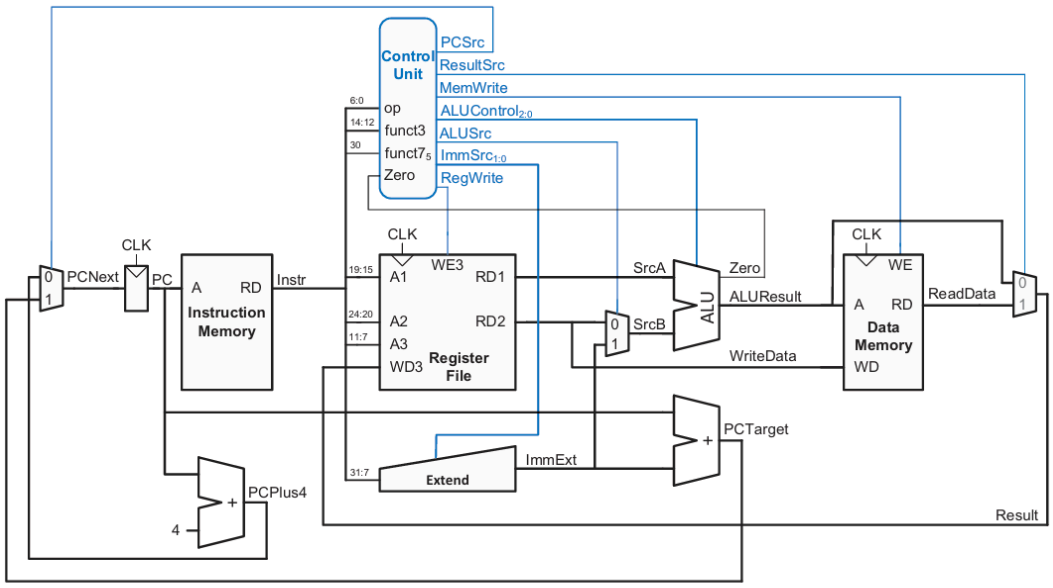

Figure 7.12 Complete single-cycle processor

Using this schematic as a guideline, we began connecting our modules together but in creating and testing our instructions, we noticed and corrected a few small design flaws. We began by increasing the size of the "controls" variable inside of our main decoder in order to account for us increasing ImmSrc, ResultSrc, and ALUop(which were also increased in order to account for their increased number of inputs accordingly, ALUop to handle more command types, Resultsrc to include PC+4, and ImmSrc to handle U-type instrcutions), and consequently increased the number of cases for the controls variable's selector variable named "op"(to account for additional instructions, including assigning the proper value to "control" for each instruction). We also added a second 1-bit-selector mux for our SrcA input into our alu taking SrcA and PC so we can execute AUIPC instructions as well as a 1-to-1 32-bit module with a 3-bit selector case statement based mux named "loadmux" for outputting different formats for our load instructions(based on funct3 passed from controller).

Section 4: Testing Strategy

Our testing strategy consisted primarily of converting and hard-coding the desired instruction values (in hex) into the "ourtest.memfile" to fill imem with our desired data, repeating the process with the dmem and "ourdata.memfile", and reconfiguring our do file to show the desired waveforms(and removing the schematic command because it kept crashing). After this we analyzed the waveform for each instruction and inspected any listed errors in the transcript or incorrect executions in the waveform before attempting to implement our designed into Vivado and ultimately onto the FPGAs.

Section 5: Evaluation

One major roadblock we ran into during this lab was the actual implementation of our code into the FPGA. We have included the error messages below. After talking with our TAs and hearing the professor's lecture over the subject, we were told this was normal and we were good to proceed with the remaining incompleted commands of the lab.