OSU ECEN 4243 Computer Architecture, Spring 2024

Lab 2: Single Cycle CPU

Group: Brendan Bovenschen A20323473 and Landon Fox A20319340

March 23rd, 2024

Section 1: Introduction

In this lab, we implemented a single-cycle CPU in System Verilog using RISCV instructions. We were given add, addi, and, andi, beq, jal, lw, or, ori, slt, slti, sub, sw and tasked to create the rest of the instructions from the RISC-V RV32i. This lab's goal is to better understand the RISC-V architecture, how a single-cycle CPU works, and the stages within it. While single-cycle CPUs may sacrifice throughput for simplicity, they serve as an essential starting point for understanding CPU design principles.

Section 2: Baseline Design

The baseline design of a single-cycle CPU employing the RISC-V RV32i structure will complete each instruction within a single clock cycle. The CPU retrieves instructions from memory and deciphers them into control signals, orchestrating the sequence of stages: instruction fetch, decode, execute, memory access, and write back. Each instruction is executed within a single clock cycle, ensuring seamless operation.

The architecture is structured with separate modules responsible for different functionalities, which will execute each instruction within a single clock cycle. The testbench module serves as the entry point for simulation, coordinating the clock signal, reset, and data input/output. It initializes the memory with test data and checks the correctness of the output after simulation. The processor's core functionality is encapsulated within the riscvsingle module, with separate modules for control, data path, and instruction decoding. The controller module generates control signals based on the instruction's opcode and funct3/funct7 fields. The maindec module decodes the main instruction fields, determining control signals for various operations. The aludec module decodes ALU control signals based on the ALUOp and funct3 fields. The datapath module integrates the data path components, including PC, instruction memory, register file, ALU, and data memory.

Section 3: Detailed Design

The first step to expanding the functionality of our processor was to implement other operations within our ALU. This encompasses the remainder of our R-type instructions and non-loading I-types. In the ALU this is as simple as creating new operational chains that are multiplexed and controlled by our ALUControl signal. However, this wire must be expanded from 3 to 4 bits to handle the amount of operations needed through these instruction types. The control logic that produces our ALUControl will be determined by the funct3 and funct7 of our instruction and the value of ALUOp where each funct3 will represent an operation, and in the case of a funct3 representing two operations, the funct7 will determine which is used by the ALU.

To modify the branch logic of our system, we decided to create a set of flags in our ALU to represent the state of the module. These flags are zero, negative, overflow, and carry. Using only these flags, we are able to perform each branch operation in the RISC-V 32-bit instruction set. For each branch instruction, the ALU will perform a subtraction operation between the values of register A and B to compare the difference between the two. This value will control our set of flags that are sent back to our control unit. From here, we will set a BranchControl value that will be active when the branch condition is met based on the active flags and current instruction. When this value and our Branch control signal is active, the value of PCSrc must be changed to the sum of our immediate value (the label of the instruction to be jumped to) and the current PC. This is all controlled through our PCNext multiplexer.

The instruction jalr requires that our PCTarget outputs the sum of RD1 and an immediate value that will set PCNext, and the current PC value plus 4 is stored to our destination register. This is completed by adding a multiplexer to the first source of our PCTarget adder which will switch between our current PC value for all other operations, and the value at RD1 for jalr. To save logic, we can use the value of ALUSrc to control this, since its logic does not interfere with the rest of our system. To set our PCPlus4 value to be written to our source register, we will use the ResultSrc multiplexer which has this value as an input and it output can be modified by our control logic.

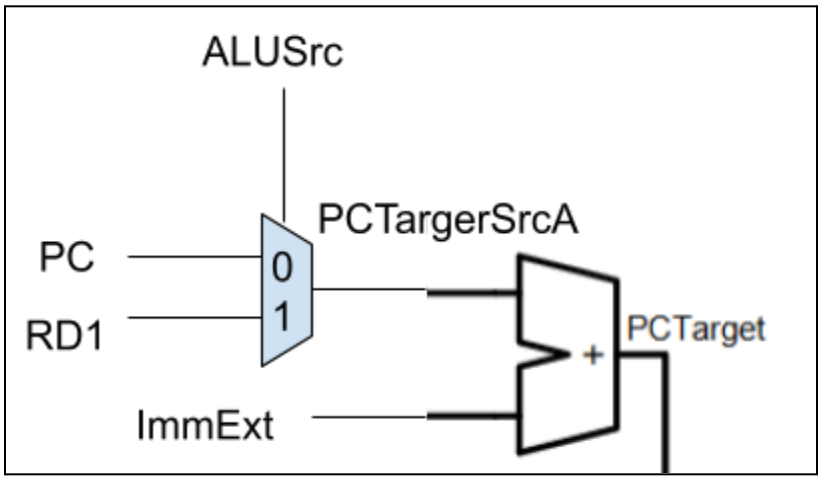

Figure 1: Logic for PCNext for jalr

The U-type instructions firsts require us to expand the functionality of our immediate extension module that will take a 20-bit input and output a 32-bit value that contains its source value in the upper 20-bits. For lui, we can just pass this value through our alu added with a value of 0 to conserve path logic. This will then be written to our source register. For auipc, we must take the value of our program counter and add it to the extended immediate to be written to our source register. We can pipe our PC value into the ALU with a multiplexer for SrcA just like what is done with SrcB. This will be set by our control logic when the auipc instruction is present, so we will call this control bit AddPC.

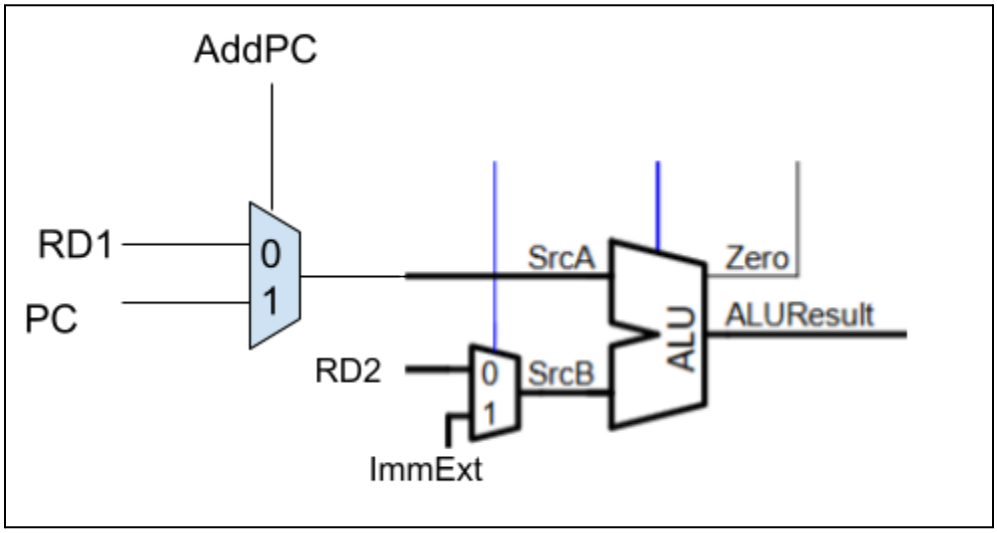

Figure 2: Logic for SrcA Relating to auipc

Implementing byte and halfword loads and store requires us to add subword write and read modules as well as change the implementation of our data memory. To properly write a ceratin range of bytes in our memory, we must create a byte mask. This is controlled by the funct3 of our instruction and the 2 least significant bits of our effective address. In this 4 bit mask, each bit of this mask represents a byte in the word to be written to. The funct3 controls the number of bits in the mask (1 for byte, 2 for halfword, 4 for word) and the byte address shifts these accordingly. For a store instruction, the write data will be copied a number of times depending on the length of the information to be stored. These copies will be operated with the bytemask to write this variable length of bits to the memory address without modifying the parts of data that are outside the range of the write. For load instructions, we can take the byte, halfword, or word at the position of our byte address, set it to the least significant bits of a 32-bit data wire, and extend it according to instructions type. This can be written to the destination register to complete the load instruction.

Section 4: Testing Strategy

Utilizing the test programs that were provided in lab one, we were able to determine that each instruction was correctly operating based on the completion of these files. When implementing each instruction, they could be tested using these memfiles, and when an instruction was added that changed some fundamental aspect of the processors, the other instructions were tested once again using these. Since the instruction memory in the processor was limited to 256 bytes or 64 words of memory, this was increase when simulating, but not when implementing on the fpga.

While these tests were useful, they were not able to be implemented for the load and store instructions due to the large size of memory required to be simulated. Because of this, we created a set of smaller test programs that implemented these instructions and different situations in which these instructions would be used with known outputs similar to the test files used in lab 1. This allowed us to verify the functionality of the modules handling these instruction and determine that the instruction was operating correctly.

Section 5: Evaluation

Since the test files for lab one aren't helpful for determining if our loading and storing instructions work as intended, we created a test program that stores immediate values and loads them back to another set of registers which is check with the first set of registers to see if their values are the same. This was done with a variable size of data structures to test each of the instructions that handle these sizes. Like lab one, if these values were not equal, a branch to a fail section of the program was activated which is placed right after the pass section and an ecall to end the program. With this program implemented, we could test each instruction using the tests from lab one and our load and store program to test all our S-type and load instructions. Seeing that the pass case was reached on each of these instructions, we could determine that each instruction was working as intended.

To implement our processor on an fpga board, we must create a system to interface with the memory on the system. This is handled through two activation bits: PCReady and MemStrobe. When an instruction is called that must access our system memory (load and store instructions), we must prevent our processor from performing any other instructions until the memory has been accessed. This will be handled by the MemStrobe bit, which will tell the system memory unit that we intend to interface with it. When this protocol is completed, this is when our PCReady bit will be activated and cause our instructions to process like normal. Because of this, we can set our PCRegister enable signal to our PCReady bit which will halt instruction processing until this is enabled again. We were unable to fully implement a program on our fpga, but we were able to load our processor with no instructions using Vivado as seen below.

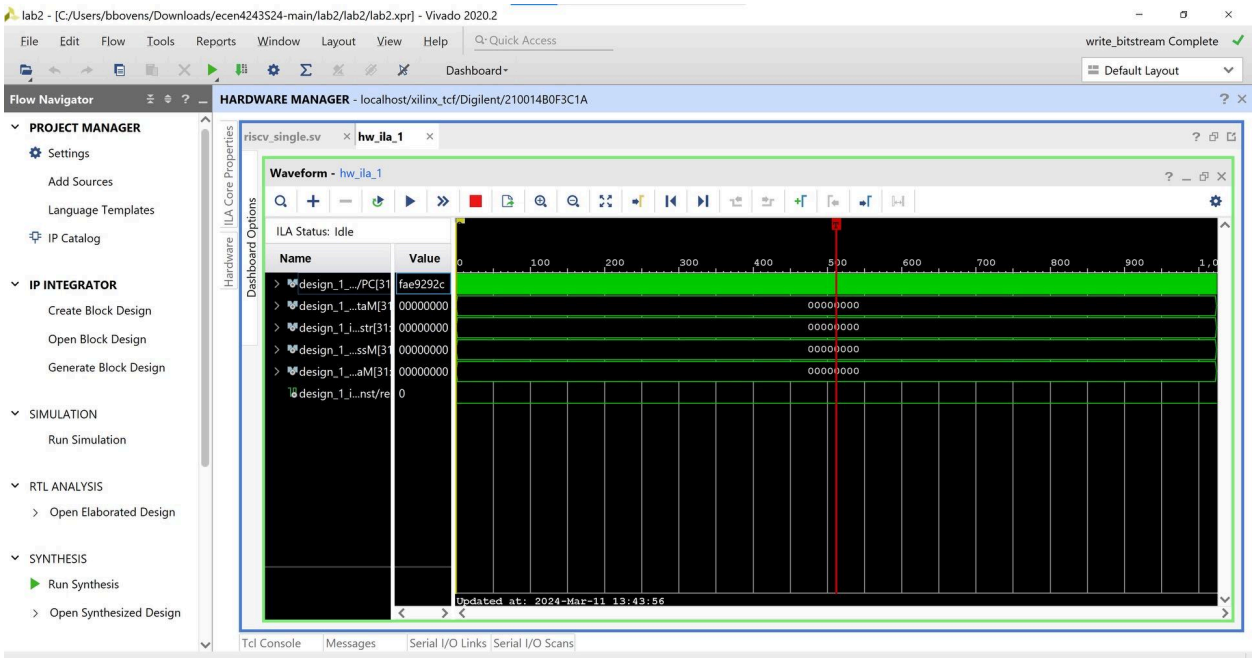

Figure 3: Attempted Vivado Implementation