ECEN4243- Lab 1

Braxtyn Ackley A20204606 Jarrett Crump A20252222

Section 1: Introduction-

In this lab we were asked to write a program in C that acted as a limited simulator for specified RISC-V instructions. For each instruction we had to implement call functions in the sim.c file, and the corresponding instruction behavior in the isa.h file. The purpose of this lab is to both give a run-through on C compiling, and to get us familiar with the RISC-V ISA since we have to implement instructions from it and compile it through C.

Our simulator should be able to take provided input files written in hexadecimal and process them as a RISC-V program. For this lab we were provided a shell file set, some premade testing files, and a basic skeleton for the sim.c and isa.h files.

Section 2: Baseline Design-

For this lab we were asked to implement calls and behavior for 38 basic RISC-V instructions. These instructions are split up into the 6 basic process types, R, I, S, B, U, and J. Each of the different process types have a different data layout and can hold different amounts of different variables.

Within the sim.c file we implemented the call functions for each instruction, using the opcode, funct3, and occasionally funct7 parameters to identify which instruction is being called in the called process type. Within each call function we also put an output that says what the function called is to make it easier to test. Within the isa.h file we created the corresponding behavior for the calls in the sim.c.

Section 3: Detailed Design-

The list of instructions we were supposed to implement is:

- R-type- add (ADD), subtract (SUB), shift left logical (SLL), set less than (SLT), unsigned set less than (SLTU), xor (XOR), shift right logical (SRL), shift right arithmetic (SRA), or (OR), and and (AND).
- Immediate (I) type- load byte (LB), load half (LH), load word (LW), unsigned load byte (LBU), unsigned load half (LHU), add immediate (ADDI), shift left logical immediate (SLLI), set less than immediate (SLTI), set less than immediate (SLTIU), xor immediate (XORI), shift right logical immediate (SRLI), unsigned shift right arithmetic immediate (SRAI), or immediate (ORI), and immediate (ANDI), and jump and link register (JALR).
- Store (S) type- store byte (BS), store half (SH), and store word (SW).
- Branch (B) type- branch if equal (BEQ), branch if not equal (BNE), branch if less than (BLT), branch if greater than or equal to (BGE), unsigned branch if less than (BLTU), and unsigned branch if greater than or equal to (BGEU)
- Upper (U) type- load upper immediate (LUI), and add upper immediate to PC (AUIPC).
- Jump (J) type- jump and link (JAL).
- There's also transfer control to OS (ECALL), which is I type, but acts different so its implemented in a different area

For the sim.c file each instruction was pretty similar since they just had to call to the isa.h file, but for each process category we had to construct a unique instruction decoder to translate the provided hexcode to binary to be read and used properly. Each process category has a specific layout of the data contained, so the decoder had to be tweaked for each. After converting the instructions to binary we could read the opcode, funct3, and funct7 when applicable, to identify which instruction is being called and where the relevant data is located within the instructions. Certain instructions also interact with the binary such as the shift instructions, so we had to convert it within the instructions as well to implement those.

For the isa.h file we implemented the functionality for each instruction individually. Some instructions such as add or subtract were implemented mathematically, some like the less than or greater than functions were logical in implementation, and some like the branch and load/store instructions were all based in computation.

Section 4: Testing Strategy-

To test our program we ran the provided memfiles and some that we made ourselves with sim.exe through cygwin and compared the memory value of the result using rdump, with what the final value of the PC was expected to be. In the provided memfiles there was a corresponding objdump file that gave the final expected PC values that we used as reference.

Section 5: Evaluation-

Ultimately, we weren't successful in implementing all of the instructions of the ISA. We found that our current implementation has issues with at least the JALR instruction, as well as the store and shift instructions. All other R, I, B, U, and J, instructions besides the ones mentioned above appear to function as they should. This is certainly something that could be ironed out with further testing, however throughout the process of troubleshooting, as we would fix something, we would uncover additional underlying issues.