Computer Architecture Lab 1: Instruction Level RISC-V Simulator

Jax Jeffries A20255596
Sivan Auerbach A20303421

Section 1: Introduction:

The purpose of this lab was to familiarize ourselves with the RISC-V architecture and the implementation of the majority of instructions. In this lab, we were given some sample functions, the shell of the ISA, the majority of the sim file, and an abundance of testing files. We implemented several R-type, I-Type, J-Type, S-Type, U-Type, and B-Type instructions to gain a better understanding of the inner workings of RISC V. Overall, this lab allowed us to gain an in-depth understanding of each instruction, instruction type, and overall architecture of the simulator. By implementing each instruction of the ISA ourselves, and more importantly analyzing and debugging the implementation, we got hands-on experience and understanding of the way the processors work and RISC-V instructions are implemented.

Section 2: Baseline Design

The baseline design of this lab consists of three main files: sim.h which holds the declaration of the structs CURRENT_STATE and NEXT_STATE, each holding information about register values and PC value. The second file is sim.c, which is in charge of reading lines from the input file one instruction at a time, decoding the instruction by breaking it apart bit-by-bit, into its components. The function decode_and_execute() reads the opcode and branches execution according to the category of instruction, and the corresponding process analyzes the func3 and func7 values, and calls the correct isa.c function of the instruction (e.g. ADD, SUB, SW, etc.). We followed the examples of implemented instructions to implement the rest.

Section 3: Design

In some parts of our design, we decided to differ from the baseline design given to us. The original design of sim.c used a series of if/else statements to identify the different instructions. We modified this into switch case structure for simplicity and elegance. In addition, we did not send Func3 and Func7 values into the isa.c functions, and dealt with them in the sim.c code. One

of the main changes we made was to the sign extension macro, which failed in some tests, so we changed it so that the input for placement of the MSB was equivalent to the number of bits (e.g. SIGNEXT(imm, 12) for a 12-bit immediate. Throughout our code, we added a lot of print statements for clarity and debugging purposes.

Section 4: Testing Strategy

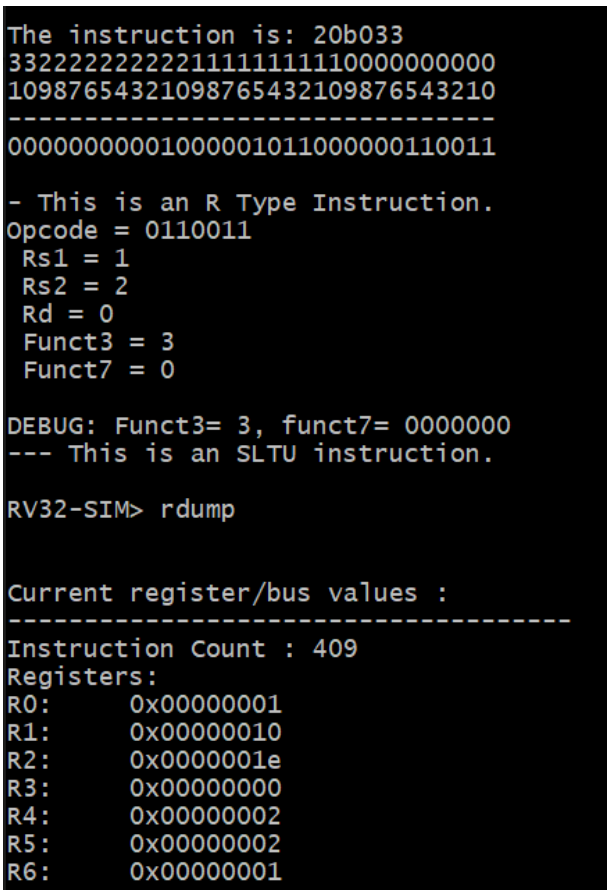
As in any lab, the testing of our instructions was a large and important part of our work. We used the provided tests to analyze and implement our instructions. The simulator would allow us to run a .memfile that contained the assembly instructions to implement the desired function and compare it to the result that should follow. We would start by running the simulator with the desired testing file and run each instruction one by one using the “run 1” command. Through following along and consulting the assembly and object dump files we were able to ensure that it would pass each test. Once the instruction passed a test we would begin to simulate more lines by using the “run” command with a higher number of lines and taking a deeper look at any failed tests. If the file failed a test, the BNE (branch if not equal) instruction would make the program jump to an ECALL and return only 0s. From there we would use the “rdump” command to look closely at the x29 and x30 registers (the two registers that contained the values being compared) and try to decipher where the issue occurred from the values there and the previous instructions. We were able to find which test number it failed by looking at the value stored in x11, as well as the PC value. We would know that all tests were passed when we received the software interrupt message at the bottom of the terminal along with the glorious “li a0, 10”instruction that signified the instruction passed all tests in the memfile. At any point in the execution, we could use the links and appendix provided on Canvas to decipher unclear instructions and trace the root of the problem.

Section 5: Evaluation

In this lab, we implemented a series of instructions in the ISA file, as well as the proper handling for each instruction. After extensive testing, we were able to write implementations that passed all the .memfile tests. One thing that we came across during testing is that the tests implement the assembly pseudoinstruction "li", as *addi, x0, imm[11:0]* (or a combination of "lui" and "addi" if the immediate is larger than 12 bits). The register x0 is supposed to always hold the value zero, which is why when added with an immediate (using addi) the result should be equivalent to "load immediate/"li". However, in many of the .memfile tests, the tests perform operations on register x0 that change the value of x0 to be nonzero, which henceforth "break" all the li instructions. Luckily, this is usually in the last/second-to-last test of the memfile and therefore does not have a big effect on execution. However, this could be a serious reliability issue. One approach to solving this is prohibiting changes to x0, or checking it is zero/resetting it to zero before any "addi" operations.

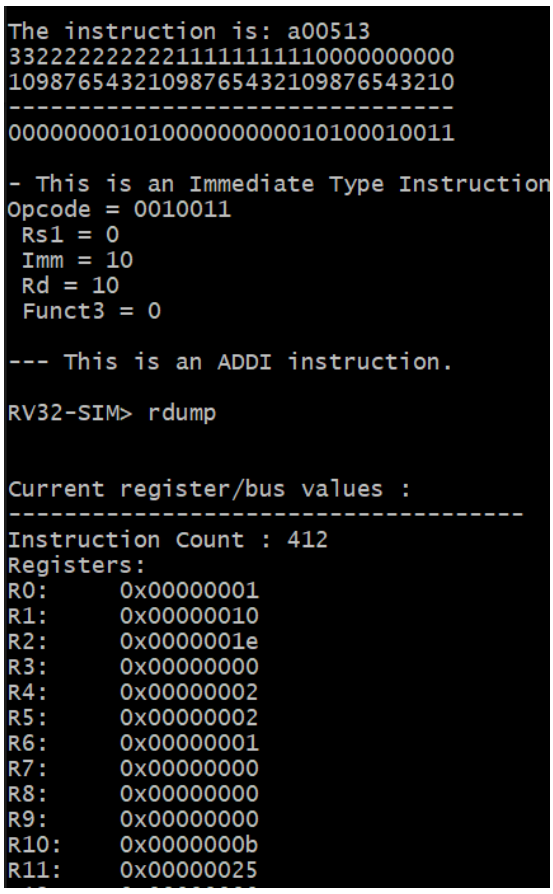
```
403   test37:
404     li a1, 37
405     li x1, 0x00000010
406     li x2, 0x0000001e
407     sltu x0, x1, x2
408     li x29, 0x00000000
409     bne x0, x29, fail
410
411   success:
412     li a0, 10
413     ecall
```

Test 37 of slu,memfile, value of



After executing “sltu x0, x1, x2”

the value of x0 is nonzero



After executing “li a0, 10” we see

a0=x10=(1)+10=0x0b, not 0x0a