Project Description:

This project is a simulator for the risc-v assembly language with an addition of some of the Compressed instructions. The input to the Simulator is a binary file that has the machine code that should be translated using the RISC-V assembly program. Each 32 bit represents an instruction word of one risc-v instruction, while the compressed instructions are represented as 16 bits. There are 32 registers each is specified by the risc-v official documentation. The simulator prints out each instruction that is being executed. Also, at the end of the program it prints out the modified and non-modified registers. The simulator terminates when it reaches the Ecall instruction to exit the code.

Execution and validation:

The program is called from the terminal and is given a filename that has the binary code that should be implemented and translated. If the file is available the program checks if it is open; in contrast, if there is no such file provided to the program, it will output "cannot access input file". The execution of such a file requires a c/c++ compiler installed in the terminal, which we would recommend using "gcc",GNU Compiler Collection.

Using gcc compiler:

Go to the located cpp directory using "cd". Then enter "g++ rv32i_sim.cpp –o rv32i_sim" to compile the program. Then “./rv32i_sim <enter file pathname>” to run the program.

Functions and implementation:
The code is divided into the main and 4 functions:

Main():

First the instruction is taken as 32 bits and is then validated if the opcode is not equal to 3 then it is a compressed instruction and the instruction is divided into 2 parts and are called by the Compression Function; otherwise, the instruction is sent to the normal Instructions function to start working on it. For each instruction that is retrieved from the main, the PC is incremented by 4. A flag condition is being used with the functions to know when the last instruction is being used as the Ecall with argument 10 in a0. After all the instructions have been executed and disassembled by the simulator, the program prints all the registers with their new values (dumps the registers).

---

emitError(string s):

This is a simple function used to print out the name of the file that cannot be opened by the program and then exits.

printPrefix(unsigned int instA, unsigned int instW):

It takes two argument the instruction word and the program counter and prints them both with a tab spaced to show to the user which instruction it is and what is its current counter in the PC.

instDecExec(unsigned int instWord):

This function receives the 32-bit instruction that has been passed on by the main as an unsigned integer. Before translating the instruction, every part of the format is initialized by manipulating with the word using the bool expressions & , | , and ^ with the right << and left >> shifting to deduce the values of the return destination (rd), register 1 (rs1), register 2 (rs2), function 3, function 7, and finally the opcode. Also, all the possible immediates are filled as specified by the risc-v official documentation. These immediates are J_imm, B_imm, I_imm, S_imm, and U_imm. To avoid any confusion, we specify regs[0] as 0 to be hard wired zero from the beginning till the final instruction. If the opcode is equal to ox33 then this is an R instruction and then it checks which R instruction it is by going through a switch case for the Func3 and if it encountered two that are the same use Funt7. Then comes the I instructions if the opcode is 0x13 but instead of using rs2 it uses the immediate to calculate the instruction needed. If the opcode is 0x3 then these are the Load instructions, the rest of the I-instructions, which access the memory to load the data into the specified register. In contrast, the store instructions take place if the opcode is 0x23. The

Branching instructions are used when the opcode is 0x63 and it compares two registers and based on the result the PC is added with the B_imm and deduced from it -4 to go through all the instructions and not skip the last one, since when the -4 was not introduced it went into an infinite loop. JAL

has an opcode of 0x6F and Links only when the rd is not equal to zero and updates the PC with the offset (J_imm). JALR is the same but with a different opcode of 0x67 and the exception and always returning a value. Ecall has the opcode 0x73 with regs[17] as its switch case choosing one of the five possibilities: 1 print int, 4 print string, 5 get int, 8 get string, 10 exit code. Finally, the U instructions LUI with opcode 0x37 and AUIPC with opcode 0x2F. If there is an unknown instruction then "Unknown Instruction" is printed in the simulator. This function always returns True with the exception of False when it reaches the Ecall 10 to exit or end.

instDecCompExec(unsigned int instWord):

This function follows the same logic as the previous one with minor exceptions. Some of which are there are only 3 opcodes available for the compressed instructions, and the funct6 is the most 6 significant bits to simplify our disassembling process. There is no rs2 in the compressed instruction, yet rd and rs1 are still used.