

Team Members:

Ahmed Raslan 900211250

Bemen Girgis 900213066

MagdElIDIn AbdalRaaof 900211145

Implementation:

Our CLI implementation mainly consists of 2 classes, memory and riscv. The memory class is a generic class that uses a binary search tree for implementing address to value. It was later used to implement registers and normal program memory. riscv deals with simulating the program. The program starts by creating an object of riscv. An object of type riscv has two memory objects called mem and reg to deal with our 4GB of memory and our 32 registers, a program counter called pc, an unordered map containing the instructions and their appropriate hex code for each instruction, an unordered map that contains register names and their corresponding register number, and a final unordered map to store labels and their locations. Then using the read_program function, the program in the path provided is read.

We also need to discuss how objects of type memory are initialized. An object of type memory defines address data pairs as a data type (addressDataPair), and a vector containing pointers to the previous data type is also defined (addressDataPairs); finally, a section address data type is defined using vector of pair of const character pointers and size_t (sectionAddresses). An object of type memory contains a map to mark the block in memory, a pointer of type sectionAddresses, and a variable to define memory size. Additionally, there exist two path pointers for _writeFile and _initFile as well as a set to contain constant addresses. To parse the initialization file, we open it then ignore comments or empty lines and we check depending on the memory format

used how to define the address and the data. It creates a new pair before pushing it into the `parsedPairs` vector of `addressDataPairs`. Memory then takes these parsed pairs and inserts them into the block map referenced earlier.

The `read_program` function takes line by line from the program file and executes the `read_line` function on it. The file is closed, and the memory and registers are initialized before parsing the program using `parse_program`. Register initialization works by having the register initialization file and the register write files and creating the 32 registers accordingly by generating the `reg` memory object. A constant address is then set where we write to memory in a big-endian format. Afterward, memory is initialized in much the same way, except we set section addresses of text, data, and stack.

`Read_line` checks if a line is just whitespace or if it is a comment. It returns and exits the function if that is the case. Otherwise, it checks if the line contains a label by finding the `‘:’` character. If a label exists then its location is identified in the `labels` array and we read the rest of the line. If a label is not present, add the line to the `instructions` array.

`Parse_program` takes the instructions we gathered and writes them to memory after parsing each individual instruction. The individual parsing is done by splitting the instructions into a mnemonic and its operands then it parses the operands accordingly by setting them according to the nature of their instruction type.

Then we start executing the program line by line as long as the program is not over. We check each instruction we parsed and execute it depending on its type to match its execution in RISC-V. We first check the type, then check the exact instruction to determine how to approach the problem. Load instructions determine how many bytes to take from the memory and sign extends the outcome. It then writes the instruction's destination register to the register memory object and

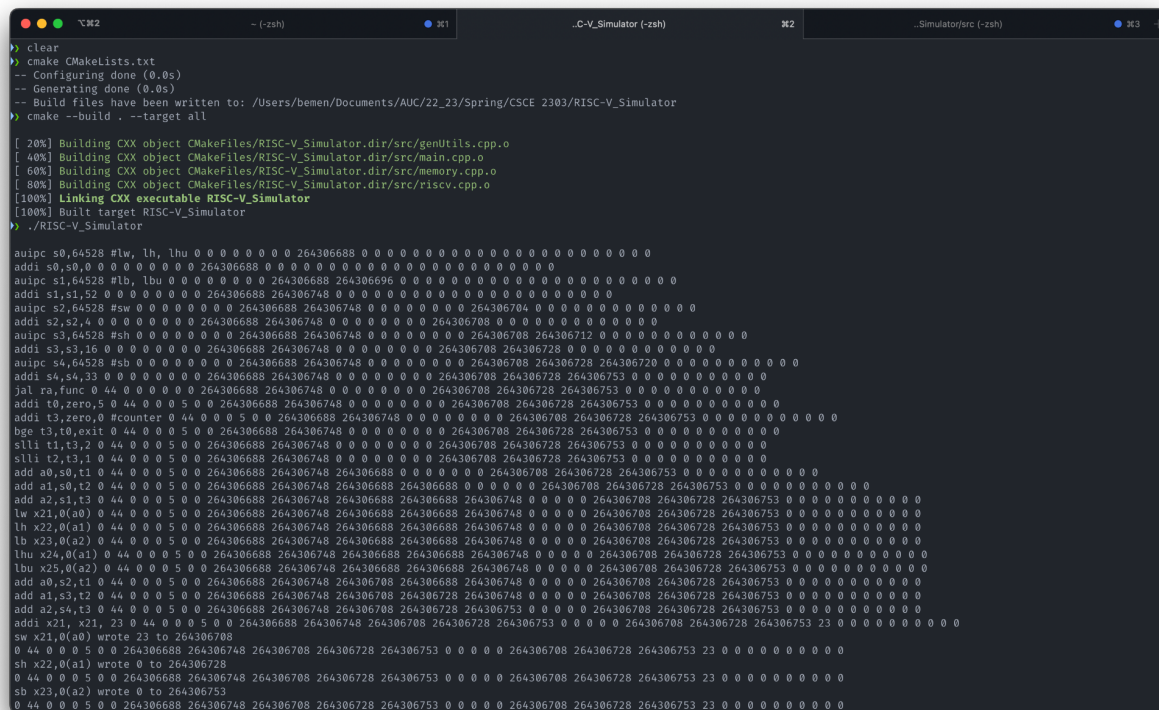
zero extends if needed for unsigned. Immediate operations are simply bitwise operations between the parameters directly. LUI and AUIPC are seen as utype and we modify the value of rd according to the write_value determined by the upper immediate of that instruction being bit shifted by 12 bits to the left. Branch instructions modify the value of the program counter according to the instruction and the label used while jtype just modify the pc and the register passed to it to store the return address. Rtype instruction operate like immediates, but they have a third register instead of an immediate. Stype decides how many bytes to write to memory depending on the instruction. Reading and writing to memory objects occurs by reading from the block map. If the pc is ever -1 or greater than the size of the instructions array, the program has finished running. After each execution, the values inside the registers and memory are printed alongside the pc value. Bonuses that have been implemented are the translation into machine code, as can be seen from how instructions are defined using binary and are translated from machine code for execution, a GUI implementation using Qt, and outputting of the values in all three formats (only in GUI, though). No known bugs have been detected.

Running the program:

To run the CLI, open the project directory and type in the following commands on your terminal.

```
$ cmake CMakeLists.txt
$ cmake --build . --target all
$ ./RISC-V_Simulator
```

Or simply run the available executables (Do you trust me enough though?)



```
> clear
> cmake CMakeLists.txt
-- Configuring done (0.0s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/bemen/Documents/AUC/22_23/Spring/CSCE 2303/RISC-V_Simulator
> cmake --build . --target all

[ 20%] Building CXX object CMakeFiles/RISC-V_Simulator.dir/src/genUtils.cpp.o
[ 40%] Building CXX object CMakeFiles/RISC-V_Simulator.dir/src/main.cpp.o
[ 60%] Building CXX object CMakeFiles/RISC-V_Simulator.dir/src/memory.cpp.o
[ 80%] Building CXX object CMakeFiles/RISC-V_Simulator.dir/src/riscv.cpp.o
[100%] Linking CXX executable RISC-V_Simulator
[100%] Built target RISC-V_Simulator
> ./RISC-V_Simulator

auipc s0,64528 #lw, lh, lhu 0 0 0 0 0 0 0 0 264306688 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
addi s0,s0,0 0 0 0 0 0 0 0 0 264306688 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
auipc s1,64528 #lb, lbu 0 0 0 0 0 0 0 0 264306688 264306696 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
addi s1,s1,2 0 0 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
auipc s2,64528 #sw 0 0 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306704 0 0 0 0 0 0 0 0 0 0 0 0
addi s2,s2,4 0 0 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 0 0 0 0 0 0 0 0 0 0 0 0
auipc s3,64528 #sh 0 0 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 264306712 0 0 0 0 0 0 0 0 0 0
addi s3,s3,16 0 0 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 264306728 0 0 0 0 0 0 0 0 0 0
auipc s4,64528 #sb 0 0 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 264306728 264306720 0 0 0 0 0 0 0 0
addi s4,s4,33 0 0 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
jal ra,func 0 44 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
addi t0,zero,5 0 44 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
addi t3,zero,0 #counter 0 44 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
bge t3,t0,exit 0 44 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
slli t1,t3,2 0 44 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
slli t2,t3,1 0 44 0 0 0 0 0 0 264306688 264306748 0 0 0 0 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
add a0,s0,t1 0 44 0 0 0 0 0 0 264306688 264306748 264306688 0 0 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
add a1,s0,t2 0 44 0 0 0 0 0 0 264306688 264306748 264306688 264306688 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
add a2,s1,t3 0 44 0 0 0 0 0 0 264306688 264306748 264306688 264306688 264306748 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
lw x21,0(a0) 0 44 0 0 0 0 0 0 264306688 264306748 264306688 264306688 264306748 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
lh x22,0(a1) 0 44 0 0 0 0 0 0 264306688 264306748 264306688 264306688 264306748 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
lb x23,0(a2) 0 44 0 0 0 0 0 0 264306688 264306748 264306688 264306688 264306748 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
lhu x24,0(a1) 0 44 0 0 0 0 0 0 264306688 264306748 264306688 264306688 264306748 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
lbu x25,0(a2) 0 44 0 0 0 0 0 0 264306688 264306748 264306688 264306688 264306748 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
add a0,s2,t1 0 44 0 0 0 0 0 0 264306688 264306748 264306708 264306688 264306748 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
add a1,s3,t2 0 44 0 0 0 0 0 0 264306688 264306748 264306708 264306728 264306748 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
add a2,s4,t3 0 44 0 0 0 0 0 0 264306688 264306748 264306708 264306728 264306753 0 0 0 0 264306708 264306728 264306753 0 0 0 0 0 0 0 0
addi x21, x21, 23 0 44 0 0 0 0 0 0 264306688 264306748 264306708 264306728 264306753 0 0 0 0 264306708 264306728 264306753 23 0 0 0 0 0 0 0 0
sw x21,0(a0) wrote 23 to 264306708
0 44 0 0 0 0 0 0 264306688 264306748 264306708 264306728 264306753 0 0 0 0 264306708 264306728 264306753 23 0 0 0 0 0 0 0 0
sh x22,0(a1) wrote 0 to 264306728
0 44 0 0 0 0 0 0 264306688 264306748 264306708 264306728 264306753 0 0 0 0 264306708 264306728 264306753 23 0 0 0 0 0 0 0 0
sb x23,0(a2) wrote 0 to 264306753
0 44 0 0 0 0 0 0 264306688 264306748 264306708 264306728 264306753 0 0 0 0 264306708 264306728 264306753 23 0 0 0 0 0 0 0 0
```

The program runs the file program.txt and initializes memory and registers from files in the ./bin directory.