Danmarks Tekniske Universitet

02155 - Computer Architecture and Engineering Fall 2018

Assignment 3

RISC-V instruction set simulator

Irene Marie Malmkjær Danvy, s163905
Frederik Ettrup Larsen, s163920

This report contains 4 pages

29. november 2018

Indhold

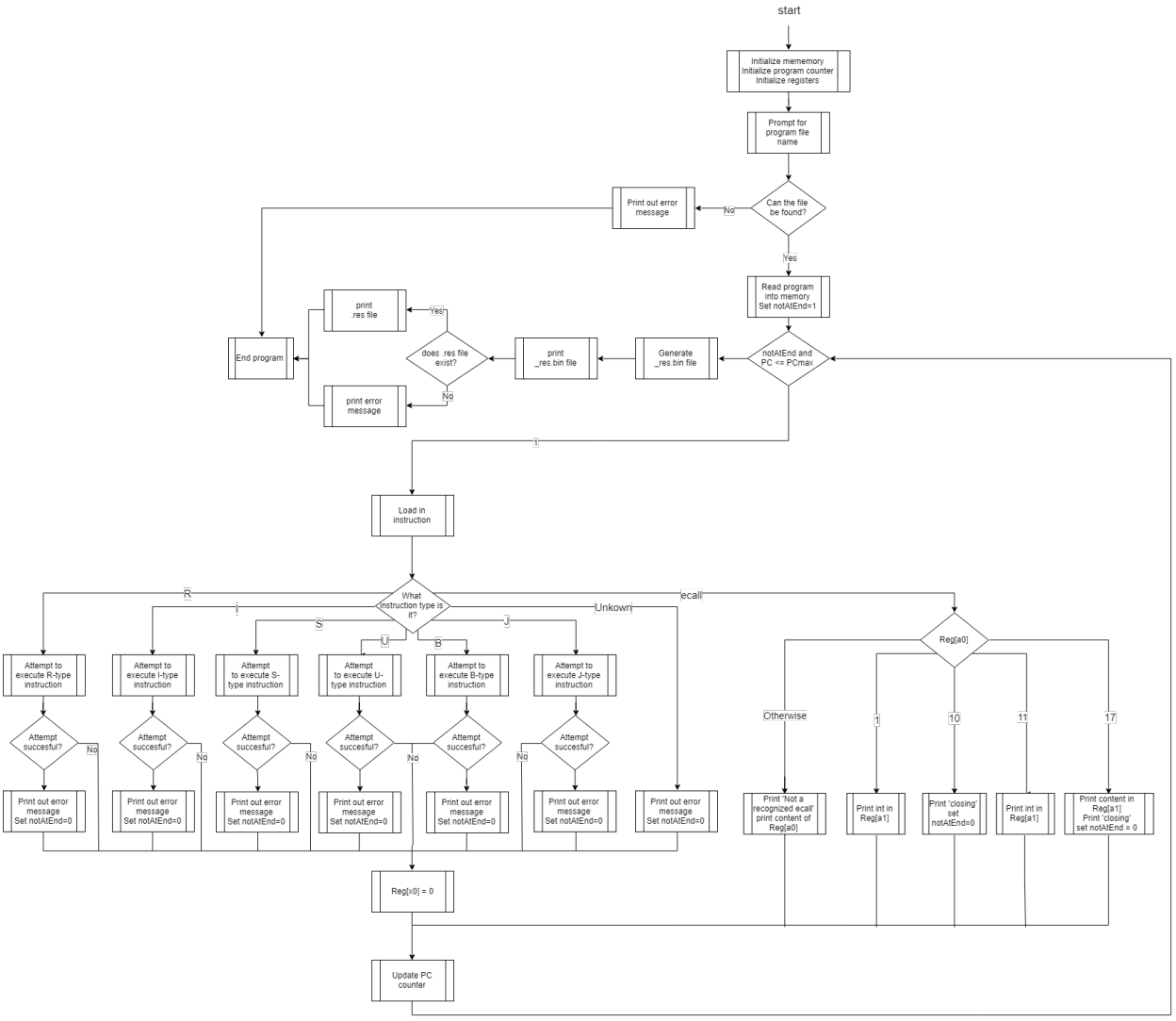
1 Introduction

This project is a RISC-V simulator written in C++. It can handle the minimal RISC-V instruction set, meaning RISC32I with the exception of the following instructions: `fence`, `fence.i`, `ebreak`, `csrrw`, `csrrs`, `csrrc`, `csrrwi`, `csrrsi` and `csrrci`,
To run the simulator, keep the binary file of the desired program in the same work space as the simulator, compile and run the simulator and type in filename of the binary file.

2 Design and Implementation

C++ was chosen as the implementation language for this simulator because it has two desirable traits: (1) it contains C, a language excellent for hardware near coding and bit-manipulation. (2) It contains libraries making reading and writing from and to files relatively simple.
An overview of the simulator's code structure can be seen in figure 1.
The memory was implemented as a very large (2^{20}) array of `uint8_t` elementes, each representing one byte. For this reason, register contents will often have to be broken down into up to 4 bytes, when storing, and combined for loading.
The stack pointer was, by convention, placed in `x2` and initialized with the array index of the last memory element.
The PC counter is initialized to the array index of the first memory element, namely zero. All programs are written into the start of the memory, with the first instruction at `Memory[0]`.
To read out an instruction, the value at `Memory[pc]` is `or`'ed and `shift`'ed to combine it with the values at pc+1, +2 and +3, which are also `shift`'ed accordingly.
To represent instructions, a union was written. This union was between a `unit32_t` and a series of structs with different bit fields, one struct for every instruction type except `I`, for which there are 2 different structs, one using `immediate` and one using `shamt`. This allows us to easily access all of the values stored in the instruction, as well as effectively handle any instruction as any type. It also means we don't have to reassign the instruction to different types, as there is one gloabal instruction type.
The registers are stored internally as an array of 32 integers (words) of length 32. To handle this in C++, many instructions involve some amount of casting as either a signed or unsigned int. This caused something of a problem for signed immediates, as the twelfth bit of a signed immediate is msb, and therefore the sign bit, but is not recognized as such by C++. In order to circumvent this problem, a function `signExtend` was implemented, it takes an integer, and the position of the msb, and returns an integer which has been sign extended properly.
Once the program has been loaded into memory, each instruction must be decoded and executed. In an attempt to make the switch statements necessary for such an undertaking more manageable, the instructions are first past through the function `whatKindOfInstruction` identifying their type from their opcode, and in the case of a few instructions also their `funct3`. Based on that result the instruction is sent to one of many functions containing the switch statements for executing that type instruction.
In a addition to the RISC-V simulation, the program also has a number of practical functions involving data management, which will now be described:

- `readFileIntoMemory`, a function used for reading binary files, which takes the name of the file, and reads it into memory under the assumption it is a program to be run.



Figur 1: Flow chart of the code design. It initializes memory, registers (including the stack pointer) and program counter, attempts to load the given program into memory, attempts to execute it in the while loop and exits if an ecall with 10 or 17 in register a0 is encountered or if the PC counter passes the last instruction. If no error has been encournted the register content is written into a binary file and the program ends. If the program file cannot be found or one of the instructions it contains is unknown, the program prints out an appropriate error message and ends.

If the file does not exists in the directory, the function returns a flag, upon witch the program ends with an error.

- `writeRegisterIntoFile`, a function for generating a file with the contents of the registers. This function takes the name of the binary input file, adds "_res" to its name and generates a binary the file containing the current content of the register. It should therefore usually be called after the simulated program has finished running.
- `printResult` will print the contents of the res file generated by `writeRegisterIntoFile`, and should therefore always be called after that function.
- `printExpected` will print the expected results, if the ".res" file exists in the directory, otherwise it gives an error.
- One more function, `writeMemoryToFile` also exists, it is not used in our code example, but can easily be added to the program if one desires. It works like the equivalent register code, only storing the memory instead.

`printResult`, `printExpected` and `writeMemoryToFile` only exits for the convenience of the user, and are not essential.

3 Discussion

Our simulator has been designed to take as much advantage of the structure of instructions as possible, having a function for each type of instruction, and recognizing instruction types based on their opcode (and sometimes also `funct3` field).
We decided to make the memory into a byte array, the same way it would be represented in a real risc-v processor. This choice of course has made it more difficult to handle storing and loading words, as they are 32 bits long. This means that we the simulator must break up the word, or stitch it back together whenever `lw` or `sw` is called. The other obvious implementation would have been to make an array of words. This would of course present a problem when storing and loading bytes. You would have to `or` up to four bytes together in one word, which means keeping track of the byte offset in a manner that would not be in style with how the rest of our code was implemented. Therefore we want with the byte array.
We choose to initialize the stack pointer to be at the top of the array, as by convention and as done in the Venus simulator. Much of the test code we've been given initializes its own stack pointer, so this of course becomes superfluous. However, we decided, that it would do no harm to initialize it if it was overwritten, but much harm if the stack pointer for some reason was uninitialized by the code, and we had not initialized it either.
We have broken the code into two header files, and one main file. This was done, to make the easier to understand and gain overview, and to sort out the two types of functions in the code. Functions relating to handling instructions, such as recognizing types, sign extending and executing instruction are stored in the aptly named `riscvInstructionHandler`. Functions that handles files, such as reading from a file, writing to a file and printing from a file, are all stored in `fileInOut`.

4 User Guide

To use this simulator, first make sure the .bin file file for you program is in the same folder as the instruction set simulator, or iss, program. To compile, make sure that every file is in the same directory, go to that directory in the terminal, and type make and hit enter. Then it should compile into a file called iss.
If you prefer not using makefiles, simply compile the three files `iss.cpp`, `fileInOut.cpp` and `riscvInstructionHandler.cpp` in your preferred C++ compiler, we used gpp.
You will be asked to write the name of your program (without .bin). Iss will then run the program, and generate a binary file, with "_res" added to the name of the input file. This file will be printed in the terminal, along the ".res" file, if such a file exists in the folder. If one also wishes to see the entire memory, one can simply uncomment line 96 in `iss.cpp`, where the command `writeMemoryIntoFile` is called. This will generate a binary file with "_mem" added to the name of the input. This file contains the entire memory after executing the program.
If you don't wish to get the terminal output, simply enter the program `iss.cpp`, go to the bottom of the main function, and comment out `printResult(input);` and `printExpected(input);`.