Assignment 5 — Simulatoropgaven

Schmidt, Victor Alexander, rqc908
Ibsen, Helga Rykov, mcv462
Barchager, Thomas Haulik, jxg170

Sunday 16:00, December 18th

1 Rapport

1.1 Implementation

In the assignment at hand, we were asked to make a RISC-V simulator, including app. 40 32-bit base instructions and the additional set of the 32-bit-long instructions for a number of multiplication operations. To achieve that, we created five helper-functions along with the main `simulate` function.

For this assignment, we have submitted two qualitatively different implementations. The first one — which is described in detail below — takes its point of departure in the classification of the instruction set into the decimal representation of the `opcode` field. The instructions are then structured according to the type: R-format, S-format, I-format and M-format.

The second implementation (cf. `simulate.c`) is thought of as splitting 32-bit-long instructions into seven fields, which is done by slicing them out with the help of the logical (for unsigned numbers) and arithmetic (for signed numbers) shift operations, as well as such logical operators as `AND` and `OR`. As far as the `simulate.c` implementation of the RISC-V simulator is quite straightforward, it does not seem necessary to provide any further comment here.

1.2 Simulate2.c implementation

int getField(): This function fetches a field from the end bit to start bit from a given instruction. It is used to fetch fields like opcode, rd, func3, rs1, rs2, func7 from the PC.

void setSignals(): This function helps to indicate which type of operation needs to be executed. It will set different signals depending on the given opcode: e.g opcode LUI sets the signal for RegWrite and ALUSrc.

int get_imm(): This function fetches the immediate from the instruction based on its opcode. It uses the getField() to fetch and is used with every operation that has the immediate field.

int ALU_control(): This function decodes the instruction based on the ALUOp, func3 and func7 fields. It is used to determine more precise what operation need to be executed.

int ALU_Action(): This function executes the found operation from ALU_control() and returns the result.

long int simulate(): This function is our main function. It is called by main.c with a given start address. So, after getting the start address, it fetches the instruction from the memory and begins its infinite loop. One loop counts as one instruction and every time we are done with a loop, we will add 4+ to our PC to get the next instruction unless the opcode is JAL or JALR, in which case the PC will be updated according to the instruction.

1.3 Simulator Tests (Documentation)

To document the correct implementation of both programs of the `simulator`, we have conducted four tests, one for each of the four .dis files located in the test folder. The `simulate.c` implementation of `simulator` have successfully passed all the four tests. The `simulate2.c` runs `echo` and `hello` without errors, while it fails when running `erat` and `fib`. The logs for simulate2.c can be found in the test folder, under the folder "Logs_simulate2".

echo.dis: We were successfully able to run this test — both implementations return as expected! When running the program, we were asked to type something and the program would echo it back in the terminal. This indicates that our simulator is able to read and write to the console and ensures a successful implementation of `ecall`.

erat.dis: While the `simulate.c` succeeds in running `erat.dis`, the `simulate2.c` fails doing that. A simple reason behind that is lack of time for finalising the code.

fib.dis: The `simulate.c`-implementation successfully implements `fib.dis` and returns as expected. Unfortunately, we ran out of time and could not finish the implementation of `simulate2.c` to make it return as expected with `fibs.dis`.

hello.dis: We were successfully able to run this test without any error, as the program returns "Hello form RISC_V". That goes for both implementations of our `simulator`.

1.4 How to run our code

To run our first implementation simulate.c, type "make sim" in the terminal and afterwards "./sim ./../tests/echo.dis -l log.txt".

To run our second implementation simulate2.c, type "make sim2" in the terminal and afterwards "./sim2 ./../tests/echo.dis -l log.txt".

Make sure to do a "Make clean".