RISC-V ESL model

Luka Dejanovic (elektronskiluka@gmail.com)

Jan. 2018

1 Introduction

RISC-V is an open source instruction set architecture (ISA) developed after years of research at UC Berkeley. It is freely available for any purpose. Every RISC-V implementation contains a base integer ISA, which can be 32 bit (called RV32I) or 64 bit (RV64I). On top of that, various ISA extensions can be added to provide more functionality. UC Berkeley has published two specifications which can be found on https://riscv.org/specifications. These documents describe the architecture of all possible RISC-V instructions, as well as the motivation behind all ISA design decisions. The first specification describes the user level instructions. The second one describes the privileged architecture.

This document presents an Electronic System Level (ESL) model of the base RV32I integer ISA. The model is written in *SystemC*. The list of all RV32I instructions with their encoding patterns is given in appendix A. This model implements all instructions in this list that are relevant to user level use only (everything except FENCE, ECALL/BREAK, and CSR instructions). In total 37 instructions (all except bottom ten from figure 2) are implemented. Please be aware that this is a first attempt at a RISC-V ESL model, so there may be bugs.

2 How to use the model?

The ESL design files are contained in the ESL directory. To build the project *cmake* is needed. The project can be build the usual way, using the command line (*mkdir Build*, *cd Build*, *cmake ..*, *make*). The building process is easier when an IDE is used, for example CLion (https://www.jetbrains.com/clion).

The model expects an input file that contains instructions in hex format. The easiest way to create those files is to use a small RV32I *Python* assembler available on Github (https://github.com/metastableB/RISCV-RV32I-Assembler). The assembler is in the **RISCV-RV32I-Assembler** directory. Several example assembly files as well as their corresponding hex files are contained in the **Programs** directory:

- `fibonacci.s`: Calculates the n-th fibonacci number.
- `multiplication.s`: Performs multiplication of two integers (using repeated addition).
- `push_pop.s`: Tests pushing and poping to/from a stack.
- `recursive_sum.s`: Recursively finds the sum of firts n whole numbers.
- `store.s`: Performs various stores and loads.

A newly created user program can be assembled to hex using the small shell script `create_hex.sh` contained in the *Programs* directory (first, modify the script to your desired file names). The script uses the mentioned *Python* assembler.

The `sc_main` uses three files:

- Log file: Logs the instruction stream while it is executed (including memory and register file dumps). This file is only used when the project is build in **DEBUG** mode.
- Hex file: Contains instructions in hex. The memory module is initialized using this file.
- Assembly file: Appended at the beginning of the log file, for clarity.

The file paths are constructed in the first few lines of the main function (`sc_main.cpp` file). It is enough to modify the *program* string (this assumes .lg, .hex and .s file extensions for the log, hex and assembly file respectively). By default, the log files are written in the **Logfiles** directory and the hex and assembly files are read from the **Programs** directory. The main function sets the file paths, instantiates the modules needed for the RISC-V core and starts the *SystemC* simulation. The simulation starts when the `sc_start()` function is called. The execution ends when either `addi R0,R0,0` or an empty instruction is fetched (or some error occurs).

3 System architecture

All the submodules use blocking communication to interact with each other and exchange data. The model architecture that shows the submodules and their communication channels is presented in figure 1. The **ESL** directory contains the following design files:

- `ALU.h`: The arithmetic logic unit.
- `cpath.h`: Contains the control path logic. The control is the most important part of the design. The control is executed using *sections* (similar to a state machine). The execution of each instruction goes through the sections. The sections are:
  - *fetchAndDecode* (fetches next instruction and acquires the decoded instruction)
  - *setControl* (sets up control signals for: register file read, ALU execution, branch and jump logic, memory operation (stores and loads) and register file write back)
  - *readRegisterFile* (reads register file data)
  - *executeALU* (executes ALU operation)
  - *calculateNextPC* (resolves branches and calculates next program counter (PC) value)
  - *memoryOperation* (performs store and load instructions)
  - *writeBack* (performs register file write back)
- `CPU_interfaces.h`: Contains the data structures that are used for blocking communication between submodules.
- `Decoder.h`: Decodes the fetched instructions.
- `Defines.h`: Contains several macro definitions (including the `MEM_DEPTH` macro that determines the memory size).
- `Memory.h`: Contains byte addressable, little endian memory (the RISC-V specification mandates little endian memory organization for the base ISA). The memory is initialized from the hex file.
- `regs.h`: The register file.
- `sc_main.cpp`: Contains the main function: sets the file paths, instantiates all the submodules and communication channels and then starts the *SystemC* simulation.
- `Utilities.h`: Auxiliary file, used in `Decoder.h`.

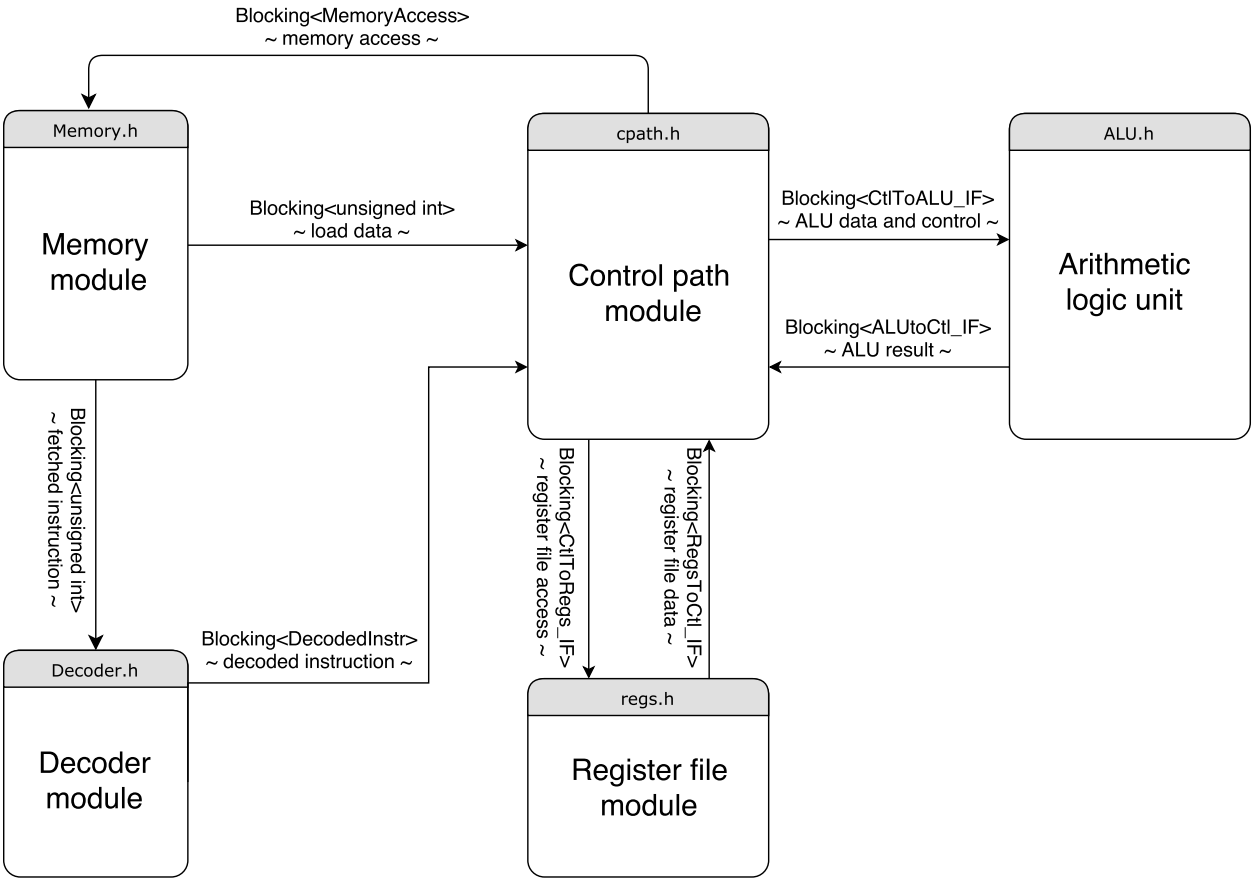

Figure 1: model

4 Testing

The instruction decoding is prone to errors because it involves a lot of bit manipulation. *Googletest* framework is used to individually verify the decoding of each implemented instruction. The tests are in `ISA_tests.h` file in the tests directory. To run them *googletest* needs to be downloaded and configured (https://github.com/google/googletest).

There are also tests for each example assembly file (using simple C++ assertions). The test class is in the `Program_tests.h` file in the **ESL** directory. These tests compare the core state (memory and register file contents) after the program execution with the expected values. It is good practice to extend these tests for each new assembly program. It is important to note that not all instructions and cases are covered with the initial set of tests. Therefore, the test suite needs to be extended to ensure bug-free behavior.

Appendix A RV32I

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 | pred | succ		00000	000	00000	0001111	FENCE
0000 | 0000 | 0000		00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

Figure 2: RV32I instructions set (with encodings).