**Rust RISC-V ISA Simulator with Qt GUI**

**Darshan H Sonecha**

**Introduction**

This document represents design and implementation of RISC-V ISA simulator, particularly geared to suite towards SHAKTI. The implementation is to be in Rust. The goal of this simulator is all to allow for **execution of program**, **operation** **systems** and **simulator test code**. The implementation of simulator should support **accuracy**, **speed, reproducibility** and **options** yielding flexibility – for example, one of the option family could be for performance of the program being executed.

The simulator is to allow for development and test of simple programs to compilers to operating systems. There are various kinds of simulation in the real-world - architectural level **simulation**, **direct execution**, **threaded code**, and **instruction set simulators 1,[[1]](#footnote-1)**. ISS are also referred to as “complete system instruction set simulator,” as well as " **1**.

The first version of the implementation would be for RV32I only, additional extensions and expansions to follow later.

We will discuss various implementation details including optimization details that improve in the performance of the simulator.

**Block Level Architect**

**Software Processes as they are mapped to**

**Realization - Data Structures, Modules, Processes**

**Program Flow - Execution Model**

**Tests**

**Conclusion**

**Scratch Ideas in Work –**

1. Block diagram for ISA execution and data flow
2. Precision consideration
3. Block level to microarchitecture logic as implemented in software – mapping.
4. Tree for OPCODE walk through – Could be part of Block Leve data flow – initial block, probably a fastest way to categorize the “next instruction”.
5. Microarchitecture logic – mapped to blocks – on reverse mapped to functions (as in Rust code functions).
6. Microarchitecture Logic + Block Level Execution Flow + …
7. No Operating System Calls simulated.

More to follow….

Paper Notes:

1. **Flexible Timing Simulation of RISC-V Processors with Sniper**

* The open instruction set allows designs to be tailored for next-generation processor goals.
* Sniper is next-generation parallel multicore simulator, which allows trading-off simulation speed for accuracy with a range of simulation options.
* This work presents an extended version of Sniper which enables support for instruction set architecture (ISA) flexibility and introduces support for RISC-V.
* The ISA is the interface between hardware and software and is a major portion of what makes up an architecture. Simulating the performance of the microarchitectural implementation of an ISA is crucial component for design space exploration of next-generation designs.
* Sniper proves a range of flexible simulation options to explore a variety of different homogeneous and heterogeneous multicore architectures, as well as Python based runtime environment that allows for analysis and simulator control.
* The Sniper instruction trace format (SIFT) files are collected and stored on disk (in the case of single-threaded applications) or generated on the fly and used for bi-directional communication between front-end and backend Sniper components.

The components of the Sniper simulator include the front-end, SIFT traces and back-end.

Frontend

* Component collects the applications’ dynamic instruction state that connects to a standalone Sniper timing instance. Typically, this is done with binary instrumentation tools such as Pin.
* ROI – Region of Interest in the application is to be simulated in detail, the code sections outside of the ROI could be simulated in functional cache warming mode (where the memory subsystem is warmed before ROI execution) or could be fast-forwarded without cache warming.
* Instruction instrument callbacks: Module that intercepts each executed instruction.
* (System Call instrumentation and thread instrumentation – OS Calls)
* Like SIFT do we need to have a trace file format of our own?

Scheduler and Backend

* This is the main component of the timing simulator. (**Question**: Do we need to worry about scheduler?).
* Each application thread in the original program will have a matching thread in the Sniper.
* **Question**: Is the Simulator we are to build support multi-threaded applications?

1. **ARMSim: An Instruction-Set Simulator for the ARM processor**

* ARMSim is a lightweight ISA (Instruction Setup Architecture) level simulator and a trace generator too.
* Simulator or Virtual machine technology is an integral part of many computing systems today.
* The deterministic behavior of simulators makes programs execution reproducible, and thus helps in locating problems.

Simulation Strategies:

**Architectural Level Simulation**: Logic designers build Architectural simulators to express and test new designs.

**Direct Execution**: Target machine binaries can be executed natively on the simulator host processor by encasing the program in an environment that makes it execute as though it were on the simulated system.

**Threaded Code**: This is the simulation technique where each op-code in the target machine instruction set is mapped to the address of some (lower level) code in the simulator system, to perform the appropriate operation.

**Instruction Set Simulators**: ISS execute target machine pragmas by simulating the effects of each instruction on a target machine, one instruction at a time.

Simulators are written to test concepts and processors design tradeoffs; flexibility is important and speed is not of primary importance.

A simulated system starting execution in a known state will always proceed along the same path. This is useful for experiments and debugging purposes.

Instead of decoding the operation fields each time an instruction is executed, the instruction is translated once into a form that is faster to execute. This idea has been used in a variety of simulators for a number of applications.

**Structure of ARMSim (Behavioral Model)**

* System Binaries
* Binary Data Representation
* Determinism
* Low Startup
* Extensible
* Statistics
* Various stages of model

1. **FAST, ACCURATE, and Validated Full-System Software Simulation of x86 Hardware**

* Validate the timing model against real hardware using a set of microbenchmarks.

In Progress.

Primary applications for simulators consist of computer architecture studies and performance tuning of compiled software and the compilation process itself.

Tools:

1. Microsoft Word
2. Emacs, IntelliJ IDE, GNU, Community 2018.3
3. GCC – RISC-V Cross Compiler, GNU
4. Bluespec System Verilog Simulation Model for accuracy of implementation, Bluespec Inc.
5. Additional tools used by Class-C Processor team – RISC-V Torture, CSMIT, AAPG etc. as applicable.

Data Structures:

RV32I\_Opcode\_Map {

u32 U\_lui; // imm[31:12] | rd | 0110111

u32 U\_auipc; // imm[31:12] | rd | 0010111

u32 J\_jal; // imm[20|10:1|11|19:12] | rd | 1101111

u32 I\_jalr; // imm[11:0] | rs1 | 000 | rd | 1100111

u32 B\_beq; // imm[12|10:5] | rs2 | rs1 | 000 | imm[4:1|11] | 1100011

u32 B\_bne; // imm[12|10:5] | rs2 | rs1 | 001 | imm[4:1|11] | 1100011

u32 B\_blt; // imm[12|10:5] | rs2 | rs1 | 100 | imm[4:1|11] | 1100011

u32 B\_bge; // imm[12|10:5] | rs2 | rs1 | 101 | imm[4:1|11] | 1100011

u32 B\_bltu; // imm[12|10:5] | rs2 | rs1 | 110 | imm[4:1|11] | 1100011

u32 B\_bgeu; // imm[12|10:5] | rs2 | rs1 | 111 | imm[4:1|11] | 1100011

u32 I\_1b; // imm[11:0] | rs1 | 000 | rd | 0000011

u32 I\_1h; // imm[11:0] | rs1 | 001 | rd | 0000011

u32 I\_1w; // imm[11:0] | rs1 | 010 | rd | 0000011

u32 I\_1lu; // imm[11:0] | rs1 | 100 | rd | 0000011

u32 I\_1hu; // imm[11:0] | rs1 | 101 | rd | 0000011

u32 S\_sb; // imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011

u32 S\_sh; // imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011

u32 S\_sw; // imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011

u32 I\_addi; // imm[11:0] | rs1 | 000 | rd | 0000011

u32 I\_slti; // imm[11:0] | rs1 | 010 | rd | 0000011

u32 I\_sltiu; // imm[11:0] | rs1 | 011 | rd | 0000011

u32 I\_xori; // imm[11:0] | rs1 | 100 | rd | 0000011

u32 I\_ori; // imm[11:0] | rs1 | 110 | rd | 0000011

u32 I\_andi; // imm[11:0] | rs1 | 111 | rd | 0000011

u32 I\_slli; // 0000000 | shamt | rs1 | 001 | rd | 0010011

u32 I\_srli; // 0000000 | shamt | rs1 | 101 | rd | 0010011

u32 I\_srai; // 0100000 | shamt | rs1 | 101 | rd | 0010011

u32 R\_add; // 0000000 | rs2 | rs1 | 000 | rd | 0110011

u32 R\_sub; // 0100000 | rs2 | rs1 | 000 | rd | 0110011

u32 R\_sll // 0000000 | rs2 | rs1 | 001 | rd | 0110011

u32 R\_slt // 0000000 | rs2 | rs1 | 010 | rd | 0110011

u32 R\_sltu; // 0000000 | rs2 | rs1 | 011 | rd | 0110011

u32 R\_xor; // 0000000 | rs2 | rs1 | 100 | rd | 0110011

u32 R\_srl; // 0000000 | rs2 | rs1 | 101 | rd | 0110011

u32 R\_sra; // 0100000 | rs2 | rs1 | 101 | rd | 0110011

u32 R\_or; // 0000000 | rs2 | rs1 | 110 | rd | 0110011

u32 R\_and; // 0000000 | rs2 | rs1 | 111 | rd | 0110011

u32 I\_fence; // 0000 | pred | succ | 00000 | 000 | 00000 | 0001111

u32 I\_fence.i; // 0000 | 0000 | 0000 | 00000 | 001 | 00000 | 0001111

u32 I\_ecall; // 000000000000 | 00000 | 000 | 00000 | 1110011

u32 I\_ebreak; // 000000000001 | 00000 | 000 | 00000 | 1110011

u32 I\_csrrw; // csr | rs1 | 001 | rd | 1110011

u32 I\_csrrs; // csr | rs1 | 010 | rd | 1110011

u32 I\_csrrc; // csr | rs1 | 011 | rd | 1110011

u32 I\_csrrwi; // csr | zimm | 101 | rd | 1110011

u32 I\_csrrsi; // csr | zimm | 110 | rd | 1110011

u32 I\_csrrci; // csr | zimm | 111 | rd | 1110011

};

0

31

|  |  |
| --- | --- |
| x0/zero | Hardwired zero |
| x1/ra | Return address |
| x2/sp | Stack pointer |
| x3/gp | Global pointer |
| x4/tp | Thread pointer |
| x5/t0 | Temporary |
| x6/t1 | Temporary |
| x7/t2 | Temporary |
| x8/s0/fp | Saved register, frame pointer |
| x9/s1 | Saved register |
| x10/a0 | Function argument, return value |
| x11/a1 | Function argument, return value |
| x12/a2 | Function argument |
| x13/a3 | Function argument |
| x14/a4 | Function argument |
| x15/a5 | Function argument |
| x16/a6 | Function argument |
| x17/a7 | Function argument |
| x18/s2 | Saved register |
| x19/s3 | Saved register |
| x20/s4 | Saved register |
| x21/s5 | Saved register |
| x22/s6 | Saved register |
| x23/s7 | Saved register |
| x24/s8 | Saved register |
| X25/s9 | Saved register |
| x26/s10 | Saved register |
| x27/s11 | Saved register |
| x28/t3 | Temporary |
| X29/t4 | Temporary |
| X30/t5 | Temporary |
| X31/t6 | Temporary |

32

31

0

|  |
| --- |
| PC |

32

References:

1. ARMSim: An Instruction-Set Simulator for the ARM processor, Alpa Shah, Columbia University
2. The RISC-V Reader – An Open Architecture Atlas, First Edition, 1.0.0, David Patterson, Andrew Waterman, November 7, 2017
3. Flexible Timing Simulation of RISC-V Processors with Sniper, Neethu Bal Mallya, Cecilia Gonzalez-Alvarez, Trevor E. Carlson, CARRV 2018, June 2018
4. Fast, Accurate, and Validated Full-System Software Simulation of x86 Hardware, Frederick Ryckbosch, Stijn Polfliet, Lieven Eeckhout, Ghent University, IEEE Computer Society, 2010.
5. ARMISS: An Instruction Set Simulator for the ARM Architecture, Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, Ge Yu, Institute of Computer Software and Theory, Northestern University
6. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS, Alasdair Armstrong, University of Cambridge, UK, et. al., January 2019.
7. SHAKTI: An Open-Source Processor Ecosystem, Neel Gala, G.S.Madhusudan, InCore Semiconductors Pvt. Ltd., Paul George, Anmore Sahoo, Arjun Menon, V. Kamakoti, Indian Institute of Technology, Madras, Advanced Computing & Communications, Processor Ecosystem, Volume 02 Issue 03 September 2018.
8. <https://riscv.org/software-tools/risc-v-gnu-compiler-toolchain/> (Tools)
9. The Rust Programming Language, Steve Klabnik and Carlo Nichols with contributions from the Rust Community, no scratch press, San Francisco, CA.
10. Mastering Qt 5, Packet Publishing, December 2016.
11. Spike -

1. Instruction set simulator [ISS] execute target machine program by simulating the effects of each instruction at a time. [↑](#footnote-ref-1)