Modular SAIL: dream or reality?

Petr Kourzanov[1]*and Anmol Anmol[1]
[1]imec DSRD CSA, *Kapeldreef 75, 3001 Leuven, Belgium*

Abstract

In order to truly benefit from RISC-V ISA modularity, the community has to address the issue of compositionality, going beyond modules at the specification level covering larger subsets of the RISC-V development flow including emulation, simulation and verification. In this paper we introduce modular SAIL*, an experiment to inject compositionality into the* SAIL-RISCV golden model*. We show that it is, in principle, not difficult to adapt the SAIL-RISCV flow (and ideally the* SAIL compiler *itself) to support modules at the emulator level. We back our findings by a comparative study of the resulting pluggable emulator's performance using both static and dynamic binding, which both exhibit same functional behaviour as the original monolithic emulator (aka RISC-V [***ISS***]).*

Introduction and related work

The benefits of open ISAs (e.g., RISC-V, [**OpenPOWER**]) and free CPUs (e.g. [**OpenSPARC**]) are well known [1]. RISC-V, on paper, has an additional bonus of offering a *modular* ISA: the implementers, tool providers as well as their users can freely agree on the set of *extensions* that need to be cast to HW and supported by SW. With the advent of co-processor interfaces such as OVI [2] and XIF [3], an *ecosystem* of extension IP vendors becomes imaginable. From standardization perspective, initiatives like CX [4] do promote this path, by pushing the interface to the ISA level.

Many of these alternatives come with different trade-offs. This can be for example in the level of SW/tooling support they require. Or the necessity of low-level assembly programming with intrinsics vs. the need to adjust the microarchitecture of the core's pipeline. Nevertheless they do share one crucial requirement - modularity of *spec*, *design*, *testing* and *verification*.

In order to truly enable such an ecosystem, the RISC-V community would address the *compositionality* not only of the RISC-V specification itself, but also of all ensuing artefacts: (1) emulators, (2) simulators, (3) test-benches, (4) compiler extensions, (5) the OS and (6) runtime libraries and middleware. While the ISA itself is open, it is unlikely that many players in the ecosystem will opt for open-sourcing their actual extension IP designs. With the rapid increase of the number of extensions for various kinds of accelerators for RVV, ML/AI and Crypto (and well as the number of their HW tapeouts) the modularity and compositionality of RISC-V extensions is expected to become more and more critical to the success of RISC-V community as a whole. In this paper we focus on the first and the most crucial artefact: the SAIL-generated `riscv_sim_RV64` emulator [5]. As SAIL-RISCV [6] is selected for the role of the *golden model*, it provides the ultimate source of truth with which further artefacts are measured. This includes, for example, the Spike simulator [7], [**RVFI**] tools, [**gcc**] & [**LLVM**] compiler toolchains.

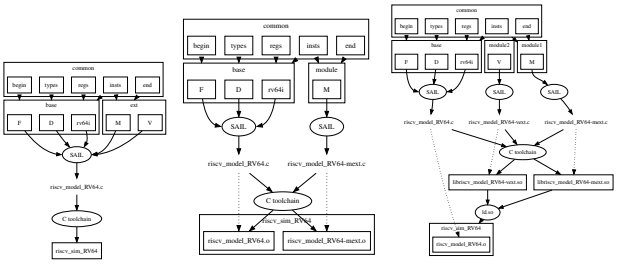
*Corresponding author: `peter.kourzanov@imec.be`

By modularizing the `riscv_sim_RV64` we aim for:

1. minimizing cross-module dependencies
2. binary plug-ability of extension *modules*
3. dynamic binding of the RISC-V ISA extensions
4. wide coverage of important [**HPC**] extensions: **M**, **V**, **P** (draft), **B** and **Crypto** (**K** in this paper)
5. minimal changes to the compiler and golden model

CBI [8] provides a fine-grained support for introducing individual instructions to a range of their cores using a proprietary Studio Fusion tool. New FUs introduced are isolated from the base core pipeline, however, the behaviour needs to be described to the toolchain at a high-level which then generates a full RTL for the complete core (as well as the required SW tools). OVI [2] from [**BSC**] and Semidynamics offers a custom signaling interface that allows a base core to offload some specific instructions to an IP coming from a separate entity. XIF [3] from [**OpenHW**] follows a more compositional approach whereby a base core decoder filters instruction stream and forwards unknown opcodes to a co-processor which then interfaces with the LSU and the RF. To our knowledge, in all three approaches the verification needed before and after RTL synthesis is still performed on the whole design.

Methodology



(a) *Original* **(b)** *Static* ***M*** **(c)** *Semi-Dyn* ***M&V***

Figure 1: *Proposed changes to the SAIL flow*

In our experiment, most significant changes were introduced in the `Makefile` of the model. As can be seen in Fig.1, instead of producing a single whole-program-processed emulator Fig.1a, we are segregating

large extensions from each other (and from most of the base core instructions and architectural state) in Fig.1b. Once this was found to be working, we enabled dynamic binding as in Fig.1c. For our aims, in this experiment, we are post-processing the generated C emulator(s) for each extension module automatically by a custom script, addressing the compositionality aspects with the help of the following transformations:

- prefixing and *externalizing* symbols of the module API: `zdecode`, `zprint_insn`, `zexecute`, `{CREATE,KILL}(zast)` and `model_{init,fini}`
- prefixing and relocation of data-structure declarations from the generated C file to a H file
- making extern the (shared) architectural state, as most registers are in the base for our experiment
- making *static* incidental definitions and helpers which might get duplicated by current approach
- removal of (de)initialization of the (shared) exception handling and run-time system mechanisms

Implementation and Results

In effect, each module obtains a well-defined API for performing the decode-print-execute loop and for committing the results to the RF. While each generated C emulator has all the features to run just the instructions provided by the given ISA descriptions, it obviously lacks the base instructions and hence can not be run in isolation from the base emulator - which provides the system context, load/store, exception and architectural state management such as CSRs.

To integrate these extensions back into the base emulator, we have applied a simple patch that adds required calls to respective functions in `zstep` and `model_init` of the base emulator.[1] In this paper, static or dynamic bindings are inserted verbatim in the source code as an example. A version with fully dynamic loading of extensions parameterized from command-line is published at our [9] github repository.

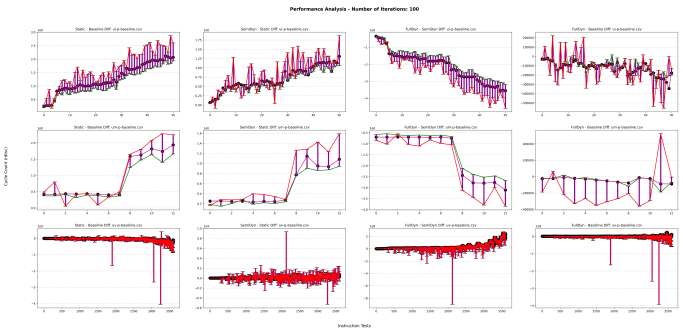
To see the effect of modularization and dynamic binding on emulator performance we have performed a series of runs for each of the base, **M**, and **V** testbenches on a dedicated `Xeon(R) w5-2455X` system running at 3.2Ghz.

The impact on binaries (`text+data+bss`) for the orig, base, static/dynamic is on Fig.2b. We can observe that changes (with **M**, **B**, **K**, **P** & **V**) do not lead to excessive size expansion and that [**LTO**] is effective.

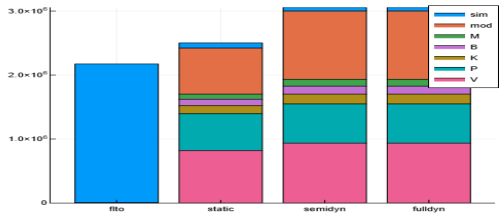
Conclusions and future work

In this paper we have shown that SAIL modularization is a promising path for improving compositionality of RISC-V designs and their ISAs. Addressing this is of paramount importance as systems grow in complexity and variation, and our work makes first steps in this

[1] latest model requires a patch to `zextensionEnabled`, too



(a) *Performance of adapted SAIL flow*



(b) *Impact of changes on binaries*

Figure 2: *Performance study results*

direction for RISC-V ISA emulators. Vendors can now exchange dynamic libraries instead of patches to the golden model, and keep internals concealed (when necessary) while relying on their partners to deliver well-tested, verified and validated extension modules.

We intend to improve upon the current version of modular SAIL by extending the loader to perform *ondemand* loading of extensions, in addition to provisions for *certifying* used dynamic libraries at load time. Extending this work to cover more extensions, primarily **F** & **D** as modules and solving the problem of modular *inheritance* of e.g., floating-point and/or vector features in other extensions such as **Vector-Crypto** would be an interesting followup. Another promising direction is to apply AI & ML techniques to automatically instantiate required modules provided a user-friendly description, trained on the published RISC-V specs. We hope that our findings will be helpful in adapting tools such as SAIL compiler to produce more modular and compositional extension modules.

References

[1] A. Waterman. "The RISC-V instruction set". In: *HotChips*. DOI: `https://doi.org/10.1109/HOTCHIPS.2013.7478332`.

[2] *Open Vector Iface*. URL: `https://shorturl.at/ulFa9`.

[3] *CORE-V-XIF*. URL: `https://shorturl.at/egnAV`.

[4] *CX Proposal*. URL: `https://shorturl.at/ufIGF`.

[5] *SAIL*. URL: `https://github.com/rems-project/sail`.

[6] *Model*. URL: `https://github.com/riscv/sail-riscv`.

[7] *Spike RISC-V ISS*. URL: `https://shorturl.at/OOBlf`.

[8] *CBI from Codasip*. URL: `https://shorturl.at/X61Kj`.

[9] *Repo*. URL: `https://github.com/kourzanov/sail-riscv`.