Composable Custom Extensions and Custom Function Units for RISC-V

Jan Gray (Gray Research) , Tim Vogt (Lattice Semiconductor), Tim Callahan (Google), Charles Papon (SpinalHDL),
Guy Lemieux (University of British Columbia), Maciej Kurc (Antmicro), Karol Gugala (Antmicro)

This poster introduces a draft specification for composable custom instruction extensions in RISC-V. The RISC-V custom instruction encoding space is unmanaged, leading to potential conflicts when combining different accelerators and their libraries into one system. This specification defines interop interfaces including a physical logic interface and CSRs that manage the composition of multiple, independently developed custom instruction extensions. Contributions include custom interface multiplexing and stateful but isolated state for multiple harts sharing multiple custom function units (CFUs).

Today, custom extensions don't interoperate

SoCs may use app-specific hardware accelerators to improve performance and energy – particularly so with FPGA SoCs that offer plasticity and abundant spatial parallelism. The RISC-V ISA explicitly supports domain-specific custom extensions.

There are many RISC-V processors with custom instruction extensions, and now some vendor tooling. But the accelerated libraries that use these extensions and the cores that implement them are authored by different organizations, using different tools, and may not work together. Different custom extensions may conflict in use of opcodes, or their implementations may require different CPU cores, pipeline structures, logic interfaces, models of computation, means of discovery, context switching, or error reporting. Composition is difficult, impairing reuse of hardware and software, and fragmenting the RISC-V ecosystem.

Unleashing innovation in interoperable custom extensions

RISC-V International uses a community process to define a new *standard extension* to the RISC-V ISA. New extensions must be of broad interest and utility to merit allocation of precious RISC-V opcode space, CSR space, and generally to add to the enduring complexity of the platform. New extensions typically require years to reach consensus and ratification. Each coexists with all other extensions. Might any new *custom extension* also safely coexist (compose) with all extensions? Might there be a rich ecosystem of plug-and-play custom extensions? *Yes!*

Our proposed interop interfaces allow *any party* to rapidly define, develop, and use:

- *a custom interface (CI):* a custom extension consisting of a set of *custom function (CF) instructions*,
- a *custom function unit (CFU):* a composable hardware core that implements a custom interface,
- an *accelerated CI library* that issues custom instructions,
- a processor that can mix and match any CFUs (plural), and
- tools to create and compose these elements into systems.

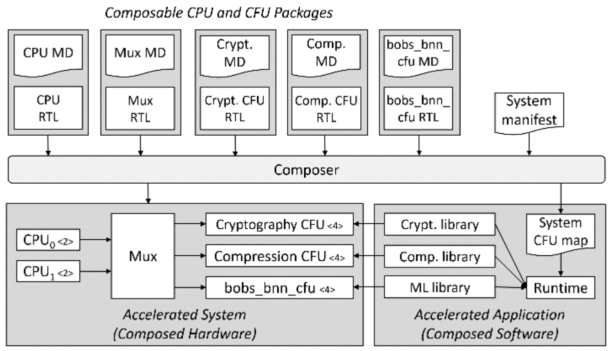

Figure 1: Composing interfaces, CFU cores, and libraries into systems

Custom interfaces, their CFUs and libraries, may be open or proprietary, even of narrow interest. Anyone can mint a new one. A new CPU core can use existing CFUs and CI libraries. A new interface, CFU, or library can be used by existing CPUs and systems. Many CFUs may implement a given custom interface, and many libraries may issue instructions of a custom interface.

Such composition requires routine integration of separately authored, separately versioned elements into stable systems that *just work* together, now, and over time, as elements evolve.

To ensure composition does not change the behavior of any interface, interfaces' state contexts are isolated: a CF instruction only accesses its source operands and its current state context.

Custom interface multiplexing

Interface multiplexing provides an inexhaustible, collision-free opcode space for custom instructions without any *central opcode authority*. Every new interface can use any or all of the `custom-0/-1` opcode space. Each accelerated CI library, prior to issuing any custom instructions, calls a runtime to obtain that interface's *(CFU,state) selector value* and write it to a new **`mcfu_selector`** CSR. This selects the hart's *current* interface (and CFU core) and its current interface state context. Like the vector extension's **`vsetvl`** instruction, an **`mcfu_selector`** write configures the behavior of custom instructions that follow.

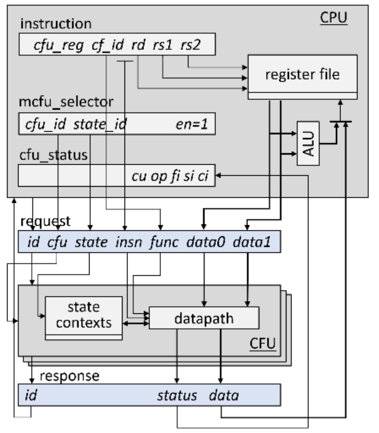

Figure 2: SW-HW interface: CI multiplexing ⇒ CFU logic interface

Custom function unit logic interface (CFU-LI)

A CPU executes a CF instruction by sending a *CFU request* to a CFU, carrying context IDs and operands. The CFU processes the request, may update its state, and sends a *CFU response*, which updates a destination register and the **`cfu_status`** CSR.

The CFU-LI defines standard signaling and metadata for combinational, fixed-latency, and variable-latency CFUs, so that CPU and CFU packages may be automatically composed.

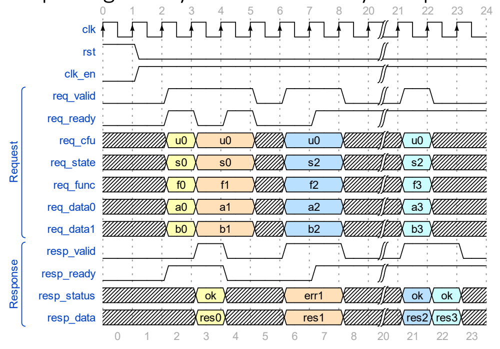

Figure 3: Example variable-latency CFU-L2 transactions

References

[1] *Draft Proposed RISC-V Composable Custom Extensions Specification*, https://github.com/grayresearch/CFU.