Draft Proposed RISC-V Composable Custom Extensions Specification

Version 0.95.240403, 2024-04-03: Draft

Table of Contents

Preface

This document comprises draft proposed specifications for hardware-software and hardware-hardware interfaces, formats, and metadata, enabling independent, efficient, and robust composition of diverse composable custom instruction set extensions, composable extension units hardware, and composable extension libraries software.

It is a work in progress. We request your feedback.

At present this is not a work product of a RISC-V International SIG, Task Group, or subcommittee. Rather we share this work in the hope that it may motivate and inform a RISC-V International Task Group for a CX-ISA composable extensions extension, plus non-ISA (CX API, CX ABI, CXU logic interface) interoperation interface standards.

(Pending standardization, implementers might elect to implement the present specifications as their own *custom extension*.)

This work summarizes years of ongoing discussions and prototyping by (alphabetical order): Tim Ansell, Tim Callahan, Jan Gray, Karol Gugala, Olof Kingdren, Maciej Kurc, Guy Lemieux, Charles Papon, Zdenek Prikryl, Tim Vogt.

---

1. Introduction: a composable custom extension ecosystem

Tip blocks signify non-normative commentary. This Introduction is non-normative. Sections titled Example are non-normative.

Note blocks signify review comments: open issues, suggested improvements.

SoC designs employ application-specific hardware accelerators to improve performance and reduce energy use — particularly so with FPGA SoCs that offer both plasticity and abundant spatial parallelism. The RISC-V instruction set architecture (ISA) anticipates this and invites domain-specific custom instructions within the base ISA (Waterman & Asanović, 2019, p. 5).

There are many RISC-V processors with custom instruction extensions, and vendor tooling for creating them. But the software libraries that use these extensions and the cores that implement them are authored by different organizations, using different tools, and might not work together side-by-side in a new system. Different composable extensions may conflict in use of custom opcodes, custom CSR addresses, or their implementations may require different CPU cores, pipeline structures, logic extensions, models of computation, means of discovery, or error reporting regimes. Composition is difficult, impairing reuse of hardware and software, and fragmenting the RISC-V ecosystem: if you can't use *this* extension, you can't also use *that* extension.

The RISC-V Composable Custom Extensions Specification defines an ISA extension plus non-ISA interoperation interfaces (HW-HW and HW-SW) and metadata, enabling a new, entirely optional, backwards compatible, managed *subcategory* of custom extensions, that make it easy and routine to compose my composable extensions with yours, and others, without prior coordination, and without recompiling our libraries or our operating system.

This enables robust reuse of anyone's composable extensions and libraries, and provides a uniform programming model across all such extensions, together enabling a marketplace of reusable extensions, libraries, and hardware modules.

1.1. Open, agile, interoperable instruction set innovation

RISC-V International uses a community process to define a new optional standard extension to the RISC-V instruction set architecture. Candidate extensions must be of broad interest and general utility to justify the permanent allocation of precious RISC-V opcode space, CSR space, and more generally to add to the enduring, essential complexity of the RISC-V platform. New standard extensions typically require months or years to reach consensus and ratification.

In contrast, the extensions defined in this specification allow anyone, whether individual, organization, or consortium, to rapidly define, develop, and use:

- a *composable extension (CX):* the *interface contract* of a composable custom extension consisting of a set of *custom function (CF) instructions* and CX custom CSRs *(CX CSRs)* and their behavior;
- a *composable extension unit (CXU):* a composable hardware core that implements a composable extension;
- a *composable extension library* that issues custom functions of composable extensions;
- a processor that can use any CXU;
- tools to create or consume these elements; and

- to compose these arbitrarily into a system of hardware accelerated software libraries.

There need be *no central authority*, no lock in, no lock out, and no asking for permission. Composable extensions, their CXUs and libraries, may be open or proprietary, of broad or narrow interest. A new processor can use existing CXUs and CX libraries. A new composable extension, CXU, and library can be used by existing CPUs and systems. Many CXUs may implement a given composable extension, and many libraries may use a composable extension.

Such open composition requires routine, robust integration of separately authored, separately versioned elements into stable systems that *just work* so that if the various hardware and software elements correctly work separately, they correctly work together, and so that if a composed system works correctly today, it continues to work, even as extensions and implementations evolve across years and decades.

Composition also requires an unlimited number of independently developed composable extensions to coexist within a fixed ABI and ISA. This is achieved with *composable extension multiplexing*, described below.

1.2. Examples

Alice develops a multicore RISC-V-based FPGA SmartNIC application processor subsystem. The software stack includes processes that already use a cryptography CX library that issues custom instructions, of a cryptography composable extension, that execute on a cryptography composable extension unit.

Profiling reveals a compute bottleneck in file block data compression. Fortunately, the compression library can use a hardware-accelerated compression composable extension, if present in the system. Alice obtains a compression CXU package that implements the extension, adds it to the MPSoC system manifest, configures its parameter settings, then re-composes and rebuilds the FPGA design. The cryptography CXU, compression CXU, CXU interconnect, and CPU cores all use the same *CXU Logic Interface*, so this incurs no RTL coding. The *system CXU map* (a new part of the device tree) is updated to map from the compression *composable extension ID (CX_ID)* (a 128-bit GUID) to the compression unit *CXU_ID*.

The compression library calls the CX Runtime to discover if compression acceleration is available. The runtime consults the CXU map for that CX_ID, finding the compression CXU_ID. Next the library uses the CX Runtime to *select* the compression extension, and its CXU, prior to issuing compression instructions to this CXU. Later the cryptography library uses the same CX Runtime API to discover and select the cryptography extension prior to issuing cryptography instructions to the cryptography CXU.

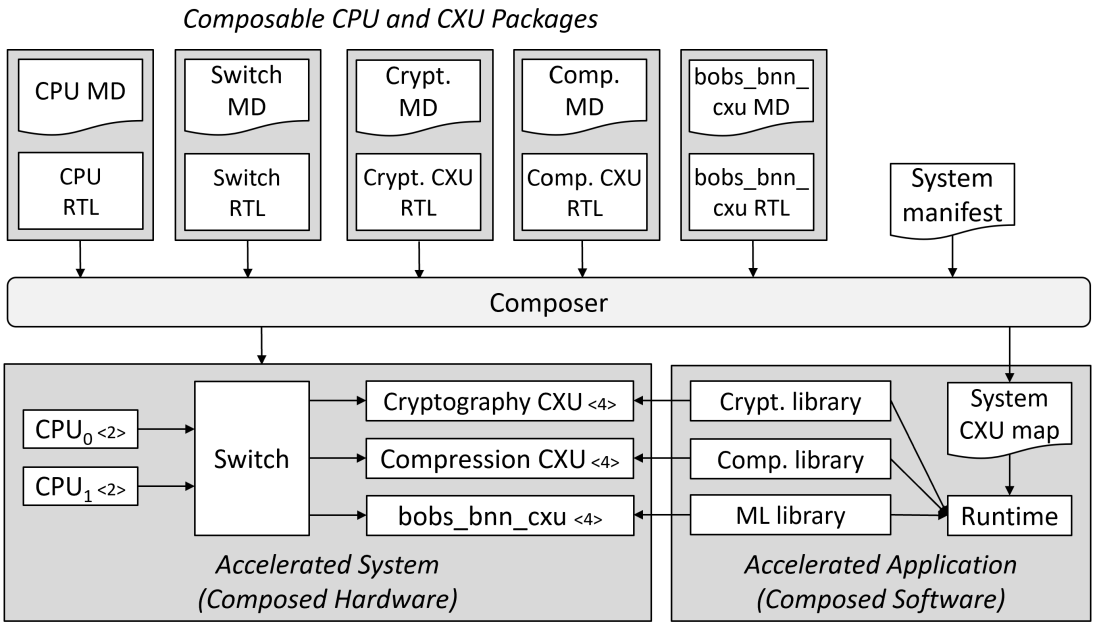


Figure 1. Bob's system, composed from CPU and CXU packages and composable extension libraries

Later, Bob takes Alice's system design, replaces the CPU cores with different (but also CXU-compatible) cores, and adds an ML inference library. For further acceleration, Bob defines a new binary neural network inference composable extension, `IBNN`, identified with a new CX_ID he mints. Bob's new BNN custom instructions reuse the standard custom instruction encodings, which is fine because they're scoped to `IBNN`. Bob develops `bobs_bnn_cxu` core, and CXU metadata that describes it. He adds that package to the system manifest and rebuilds the system, updating the CXU map. Bob's system now runs highly accelerated with cryptography, compression, and inference custom function instructions issuing from the various CPU cores and executing in the various CXUs.

Figure 1 illustrates this. A *Composer* EDA tool assembles and configures the reusable, composable CPU and CXU RTL packages into a complete system, per the system manifest, and generates a devicetree (or similar) that determines the system CXU map. Each extension library uses the CX Runtime to select its respective composable extension, and its CXU, prior to issuing custom function instructions of that extension to that CXU.

1.3. Scope: reliable composition via strict isolation

To ensure that composition of composable extensions and their CXUs does not subtly change the behavior of any extension, each must operate in isolation. Therefore, each custom function (CF) instruction is of limited scope: exclusively computing an ALU-like integer function of up to two operands (integer register(s) and/or immediate value), with read/write access to the extension's private state (if any), writing the result to a destination register.

A CF may access the CX's custom CSRs (CX CSRs).

A CF may not access other resources, such as floating-point registers or vector registers, pending definition of suitable custom instruction formats.

A CF may not access *isolation-problematic* shared resources such as memory, standard CSRs, the program counter, the instruction stream, exceptions, or interrupts, pending a means to ensure correct composition by design. (Except that, as with RISC-V floating point extensions, the error model accumulates CX custom operation errors in a shared CX status standard CSR.)

The isolated state of a composable extension can include private registers and private memories.

1.3.1. Stateless and stateful composable extensions

A composable extension may be stateless or stateful. For a stateless extension, each CF is a pure function of its operands, whereas a stateful extension has one or more isolated state contexts, and each CF may access, and as a side effect, update, the hart's *current* state context of the extension (only).

Isolated state means that latency notwithstanding, 1) the behavior of the extension only depends upon the series of CF requests issued on that extension, and of CX CSR accesses to that extension, and never upon on any other operation of the system; and 2) besides updating extension state, the CX status CSR, and a destination register, issuing a CF has no effect upon any other architected state or behavior of the system. Issuing a CF instruction or accessing a CX CSR may update the current state context of the composable extension but has no effect upon another state context of that extension, nor that of any other extension.

A CXU implementing a stateful composable extension is typically provisioned with one state context per hart, but other configurations, including one context per request, activity, fiber, task, or thread, or a small pool of shared contexts, or several harts sharing one context, or one singleton context, are also possible. Similarly, each CXU in a system may be configured with a different number of its state contexts.

All stateful composable extensions implement CX state context CSRs for uniform (extension-agnostic) CX state context save/restore/management.

CX CSRs provide access to control and status of a stateful composable extension. A stateful CX may also have other isolated state (i.e., as a side effect of the cumulative history of stateful CF instructions issued) that is not explicitly accessible, but which nevertheless determines the behavior the CX's CF instructions and custom CSRs.

1.4. Standard CX ISA extensions and interoperation interfaces

To facilitate an open ecosystem of composable extensions, CXUs, libraries, and tools, the specification defines new CX ISA extensions and varioius interoperation interfaces and formats:

- *CX-ISA*, the *Composable Extensions' ISA Extensions,*
- *CX-API*, the *Composable Extensions' Application Programming Interface,*
- *CX-ABI*, the *Composable Extensions' Application Binary Interface,*
- *CXU-LI*, the *CXU Logic Interface*,
- *CXU-MD*, build-time *CXU Metadata.*

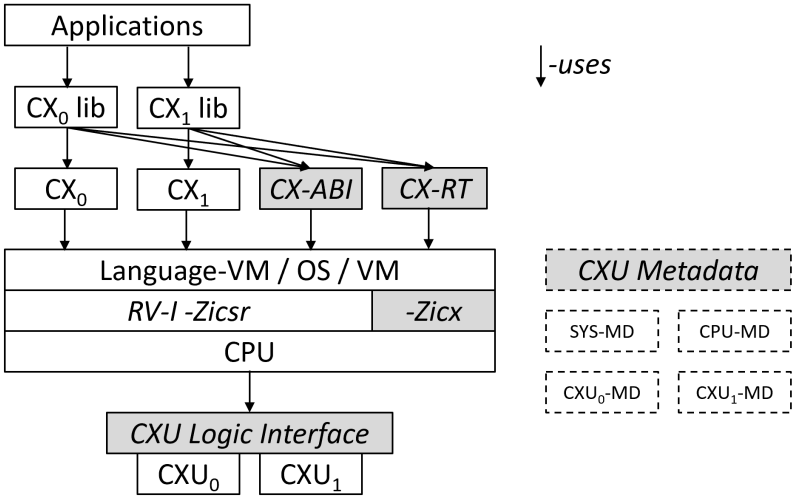


Figure 2. Hardware-software extensions stack. New standard extensions and formats are shaded.

The hardware-software extensions stack (Figure 2) shows how these extensions and formats work together to compose user-defined composable extensions CX_0 and CX_1, their libraries, and their CXUs into a system.

1.4.1. CXU Logic Interface (CXU-LI)

The CXU-LI defines the hardware-to-hardware logic extension between a *CXU requester* (e.g., a CPU) and a *CXU responder* (e.g., a CXU). When a custom function instruction issues, the CPU sends a *CXU request*, providing the request's *CXU identifier* (*CXU_ID*), the *function identifier* (*FUNC_ID*), *_state index* (*STATE_ID*), if any, and request data (operands). The CXU performs the function then sends a *CXU response* providing response data and error status.

In a system with multiple CPUs and/or CXUs, switch and adapter CXUs accept and route requests to CXUs and accept and route responses back to CPUs. The CXU-LI supports CPUs and CXUs of various *feature levels* of capability and complexity, including combinational CXUs, fixed-latency CXUs, and variable latency CXUs with flow control.

1.4.2. CX-ISA: composable extensions' ISA extension

The CX-ISA "composable extensions" extension adds four new standard CSRs that provide access-controlled composable extension multiplexing and error signaling. These CSRs modify the behavior of `custom-[012]` instructions (Waterman & Asanović, 2019, p. 143) and custom address CSRs, to compose, conflict-free, with other composable extensions and with any built-in custom extensions. The four new CXU CSRs are:

- `mcx_selector`: selects the hart's current CXU_ID and STATE_ID, for composable extension multiplexing;
- `cx_status`: accumulates CXU errors;
- `scx_table`, `cx_index`: efficient access control to CXUs and CXU state.

The supervisor mode `scx_table` CSR is probably insufficient for processors with hypervisor privilege levels. This will require additional spec design work and additional CX-ISA CSRs.

1.4.3. Composable extension multiplexing

Composable extension multiplexing provides inexhaustible collision-free custom instruction opcodes and custom CSR addresses, for diverse composable extensions, without resort to any *central assigned opcodes authority*, and thereby facilitates direct reuse of CX library binaries.

A custom-extension-aware library, prior to issuing a CF instruction, must first CSR-write a *system and hart specific* CX selector value to `mcx_selector`, routing subsequently issued CF instructions on this hart to its CXU and to a specific state context. Like the -V vector extension's `vsetvl` instructions, a CSR-write to `mcx_selector` is a prefix that modifies the behavior of CF instructions that follow. With each CF instruction issued, the CPU sends a CXU request to the hart's current CXU and its current state. This request is routed by standard switch and adapter CXUs to the hart's *current* CXU, which performs the custom function using the hart's current state context. Its response is routed back to the CPU which writes the destination register and updates `cxu_status`.

The `mcx_selector` CX selector value, a tuple (CXU_ID, STATE_ID), is system specific because different systems may be configured with different sets of CXUs, with different CXU_ID mappings, and is hart specific because different harts may use different isolated state contexts. Raw CX selector values are not typically compiled into software binaries.

In a system with multiple CX libraries that invoke CF instructions on different extensions, each library uses the CX Runtime to look up selectors for a CX_ID and update `mcx_selector`, routing CF instructions to its extension's CXU and state context. Over time, across library calls, `mcx_selector` is written again and again.

Reuse of custom instruction encodings across extensions will make debugging, esp. disassembly, more challenging.

The `mcx_selector` also incorporates a *custom operation exception enable* (`cxe`). When set, custom instructions and accesses to custom CSRs raise an illegal instruction exception. This enables software emulation of absent custom instructions, software emulation of absent composable extensions, and transparent virtualization of stateful composable extensions.

1.4.4. CX state context custom CSRs

The specification defines mandatory *CX state context custom CSRs* that stateful CXs implement to provide a uniform CX programming model:

- `cxs_error` (user R/W): CX state context extended error information;
- `scxs_status` (system R/W): CX state context status word;
- `scxs_index` (system R/W): CX state context blob index;
- `scxs_data` (system R/W): CX state context blob data at index.

These CX CSRs enable CX software to access stateful CX error status, and enable a CX-agnostic runtime or operating system to manage, save, and reload any CX state context.

1.4.5. CX-API (Application Programming Interface) and CX-ABI (Application Binary Interface)

Together the CX-API (the *CX Runtime* API) and CX-ABI provide the programming model used by composable extension libraries.

Both are necessary for correct discovery, operation, and composition of CX libraries. As described above (1.4.2) the current `mcx_selector` CSR selects the current composable extension/CXU and state context for the hart. However, a CX library should not directly create a CX selector value, nor directly access the CSR. Rather a CX library uses the CX Runtime to look up the CX selector value for its composable extension's CX_ID and to write it to `mcx_selector`, prior to issuing CF instructions. For example,

```
#include "cx.h"                          // CX Runtime: class use_cx { ... }
..
use_cx cx(CX_ID_IBitmanip);              // csrrw mcx_selector
uint32_t count = cf(pcnt_cf, data, 0);   // cx_reg cf_id, rd, rs1, rs2
```

The CX-ABI defines the calling convention for managing the `mcx_selector` CSR.

Its design follows these tenets and (competing) goals:

1. Support composition of CX libraries, including nested composition of CX libraries, alongside legacy custom extension libraries.
2. Support preexisting legacy custom extension libraries, even when they don't explicitly manage (disable) CX muxing.
3. Minimize the CX selection "trust surface" to that of the current function (or perhaps, current library).
4. Minimize the number of CX selector writes.

Therefore for maximum preexisting legacy custom extension library compatibility and maximum paranoia (least trust of other code), the CX-ABI keeps CX muxing off across function calls, only enabling CX muxing and selecting a CX and CX state context immediately prior to issuing that CX's custom instructions.

The CX-ABI defines these five rules, which must be implemented explicitly in code or automatically by CX-ABI aware compilers:

1. **[ABI-INIT]**: Initially, the selection is legacy mode.
2. **[ABI-ENTRY]**: On entry to a function, or following a function call, the selection is legacy mode.
3. **[ABI-SELECT-CX]**: Code **must** select a CX prior to issuing that CX's custom operations.
4. **[ABI-DESELECT-CX]**: Code that selects a CX **must** select legacy mode prior to calling a function, returning, or stack unwinding.
5. **[ABI-SELECT-LEGACY]**: Code **should** select legacy mode prior to issuing built-in custom operations.

This is discussed in more detail in the CX-ABI chapter.

1.5. System composition

1.5.1. Metadata and system manifest

To support automatic composition of CPUs and CXUs into working systems, this specification defines a standard CXU metadata format that details each core's properties, features, and configurable parameters, including CXU-LI feature level, data widths, response latency (or variable), and number of state contexts. Each CPU and CXU package, as well as the system manifest, include a metadata file.

1.5.2. Composer

A system composer (human or tool) gathers the system manifest metadata and the metadata of the manifest-specified CPUs and CXUs, then uses (manual or automatic) constraint satisfaction to find feasible, optimal parameter settings across these components. The composer may also configure or generate switch and adapter CXUs to automatically interconnect the CPU and the CXUs.

For example, a system composed from a CPU that supports two or three cycle fixed latency CXUs, a CXU_1 that supports response latency of one or more cycles, a CXU_2 that has a fixed response latency of three cycles, and CXU_3 which is combinational (zero cycles latency), overall has a valid configuration with three cycles of CXU latency, with the CPU coupled to a switch CXU, coupled to CXU_1 and CXU_2 and to a *fixed latency adapter CXU*, coupled to CXU_3.

1.5.3. Diversity of systems and operating systems

Composable extensions and CXUs are designed for use across a broad spectrum of RISC-V systems, from a simple RVI20U (+CX-ISA) microcontroller running bare metal fully trusted firmware, to a multicore RVA20S Linux profile, running secure multi-programmed, multithreaded user processes running various CX libraries, and with privileged hypervisors and operating systems securely managing access control to CXUs and CXU state.

1.6. Versioning

Interoperation specifications live for decades. Meanwhile "the only constant is change". This specification anticipates various axes of versioning.

- Specification versioning. This specification and its requirements will evolve. The extensions and formats it specifies will evolve. This includes the CXU Logic Interface, for example.
- CXU-LI versioning. The CXU hardware-hardware extension spec will evolve, with new signals, behaviors, constraints, metadata.
- Composable extension versioning. Any user-defined composable extension may evolve, changing or adding custom functions, changing behaviors, semantics.
- Component implementation versioning. Without changing the extensions it implements, the implementation of a component such as a CXU, CPU, or a CX library may change for a bug fix, a performance enhancement, or any other reason..

How are these anticipated and addressed?

CXU-LI versioning: A CXU module configuration parameter `CXU_LI_VERSION` indicates to the CXU the version of the CXU-LI signals and semantics in effect.

Versioning of the extension multiplexing mechanism: The `mcx_selector.version` field determines the current extension multiplexing version. It provides backwards compatibility with legacy custom instructions (i.e., multiplexing *off*) and forwards compatibility with future extension multiplexing schemes, anticipating future layouts and interpretations of other selector fields and future means of decoding `custom-[0123]` instructions into CXU requests.

Composable extension versioning: A composable extension is immutable. To change or add any custom functions or their behaviors, a new composable extension must be minted. (Consider the many AVX vector extensions variants have been introduced over many years.) With Microsoft COM software components, an extension `IFoo` might evolve to become `IFoo2`. The original `IFoo` remains and `IFoo` clients are unaffected. But every component implements `IUnknown::QueryInterface()`, to determine if the component implements a given extension. A component might implement both extensions, giving its client a choice.

Similarly a CXU might implement two composable extensions, e.g. `IPosit`, and `IPosit2`, an enhanced version of `IPosit` introduced later. In that case, the CXU will have two CXU IDs, `CXU_CXU_ID_MAX=2`, one for each extension it implements, each present in the CXU Map, from `CX_ID_IPosit` to the first CXU ID and `CX_ID_IPosit2` to the second. Thus each CX software library present can access the extension, functions, and behavior it depends upon, even if only one CXU module implements both behaviors.

Note how composable extension multiplexing facilitates extension versioning: a new version of an extension (i.e., a new extension) may be introduced at no cost to any existing or future extension.

Implementation versioning: This does not change the extension to a component (e.g.. for a CXU, its CXU-LI and the composable extension it implements). At system composition time it may be necessary to specify implementation version requirements, perhaps in metadata, but this should not be visible to, computed upon, nor depended upon, the HW-HW-SW interfaces.

TODO: Add examples of Alice and Bob's travails with their composed SoC designs, over time.

All version numbering uses semantic versioning semver.org.

1.7. Pushing the envelope

The hardware-hardware and hardware-software extensions proposed in this draft specification are a foundational step, necessary but insufficient to fully achieve the modular, automatically interoperable extension ecosystem we envision.

A complete solution probably entails much new work, for example in runtime libraries, language support, tools (binary tools, debuggers, profilers, instrumentation), emulators, resource managers including operating systems and hypervisors, and tests and test infrastructure including formal systems to specify and validate composable extensions and their CXU implementations.

Whether or not the specific abstractions and interoperation extensions proposed herein are adopted, we believe this specification motivates composable extension composition, and illustrates *one approach* for such composition scenarios using RISC-V, in sufficient detail to understand how the moving pieces achieve a workable composition system, and to spotlight some of the issues that arise.

1.8. Future directions, TODOs

The present specification focuses on composition at the hardware-software extension, and below. Future work includes:

- Expand the scope of composable extensions to include access to non-integer registers, CSRs, and memory, while preserving composition.
- Expand the CXU Logic Interface to support greater computation flexibility and speculative execution.
- Design and implement an automatic system composition tool.

1.9. Acknowledgements

Composable Extensions are inspired by the Interface system of the Microsoft Component Object Model (COM), a ubiquitous architecture for robust arms-length composition of independently authored, independently versioned software components, at scale, over decades (Microsoft, 2020).

(End of non-normative Introduction section.)

2. Composable extensions: the hardware-software interface

The Composable Extension abstraction bridges software and hardware, enabling diverse software libraries which target the same extension and diverse hardware CXU cores which implement the same extension. Then *composable extension multiplexing* enables composition of systems of separately authored and versioned components.

2.1. Definitions

A **custom function (CF)** is a function from two integer operands to an integer result and response status. May be stateless or stateful.

A **custom function identifier (CF_ID)** is an integer, in the scope of a composable extension, identifying a custom function. A **valid CF_ID** is a value that identifies a CF instruction implemented by a configured extension.

A **stateless custom function** is a CF that is a pure function of its operands (only). Never reads nor writes any other architected state. Given the same operand values, always produces the same result and response status.

A **stateful custom function** is a CF that is a function of its operands and its composable extension state context (only). May read and write the context but never reads or writes other architected state. Equivalently: a CF that is a function of its operands and of any prior CF invocations upon its composable extension (only).

A **composable extension (CX, extension)** is a fixed named set of custom functions and custom CSRs. May be stateless or stateful. *Fixed:* immutable, i.e., any versioning of the CFs or custom CSRs or the behavior of an extension necessarily defines a new extension. *Named:* has a composable extension identifier.

A **composable extension identifier (CX_ID)** is a 128-bit globally unique ID (*GUID*) [see RFC-4122], unique in history, identifying a composable extension.

A **stateless composable extension** is a fixed named set of set of stateless custom functions. A stateless CX has no read-write custom CSRs but may have read-only custom CSRs.

A **stateful composable extension** is a fixed named set of custom functions, at least one of which is a stateful custom function, plus a set of custom CSRs, plus a composable extension state context.

A **composable extension state context (state context, state, context)** is an isolated collection of state associated with a stateful composable extension. Isolated: stateful custom functions of the extension may read and write the state context, but no other element or operation of the system may read or write the state context.

A **CX CSR** is a custom CSR of a stateful composable extension.

A **CX operation** is a custom opcode instruction or a custom address CSR access.

A **function ID (FUNC_ID)** is an integer, in the scope of a composable extension, that identifies a custom operation. At present it conveys either a CF_ID of a custom function instruction or a CSR access of a custom CSR access instruction.

A **configured composable extension (configured extension)** is an extension that is configured (included) within a system and is implemented by a CXU of the system (a **configured CXU**). Within a system, a configured extension has some configured number of state contexts.

A **configured extension subset** is a configured extension in which one or more custom functions of the extension

are not implemented. The CF_IDs of unimplemented custom functions are invalid.

A **composable extension state context identifier (STATE_ID)** is an integer index, in the scope of a configured extension, in the range [0, no. of state contexts-1] identifying one of an extension's contexts in the system. A stateless extension has zero state contexts and uses STATE_ID=0 whenever a STATE_ID is required. A **valid STATE_ID** is a value that identifies a state context of a configured extension.

A **custom function instruction (CF instruction)** is a RISC-V custom instruction that executes a custom function using a composable extension unit, sourcing the integer operands from the register file and/or from an immediate field of the instruction, writing the integer result to the register file, and updating the CX status CSR with the response status.

A **composable extension unit (CXU)** is a core that implements one or more composable extensions. A **stateful CXU** implements at least one stateful composable extension.

A **composable extension unit identifier (CXU_GUID)** is a 128-bit globally unique ID (*GUID*) [see RFC-4122], unique in history, identifying a specific CXU implementation.

A **CXU_ID** is an integer, in the scope of a system, that identifies a configured extension implemented by a CXU. When one CXU implements multiple configured extensions, different CXU_IDs identify the configured extensions. A **valid CXU_ID** is a CXU_ID value that identifies a configured extension.

A **composable extension selector (CX selector, selector)** is a 32-bit value written to `mcx_selector` CSR to select the hart's current extension multiplexing version, (e.g., *off*, *version-1*, ...), whether *trap on custom operation* is *enabled*, and to specify the hart's current configured extension / CXU and current state context.

A **CX selector table** is a 4 KB aligned, 4 KB sized table of 1024 CX selectors. When CX access control (§2.9) is supported, each hart has a `scx_table` CSR to address its CX selector table.

A **selector index** is an integer that identifies an entry in a CX selector table (§2.9).

2.2. New CX control / status registers

A CX-ISA compatible CPU shall implement the `mcx_selector` and `cx_status` CSRs for extension multiplexing and custom function instruction execution.

When CX access control (§2.9) is supported, a CX-ISA compatible CPU shall implement the `scx_table` and `cx_index` CSRs.

All CX-ISA CSR fields marked *reserved* are WPRI, write preserve, read ignored, and all other fields are WARL, write any/read legal values. (An invalid CXU_ID or STATE_ID value is still *legal*).

All CX-ISA CSRs are initialized to zero on reset.

2.2.1. `mcx_selector` CSR 0xBC0: select active CXU and state context

The `mcx_selector` CSR implements composable extension multiplexing. It is assigned various CX selectors over time. This enables or disables CX multiplexing and selects the hart's current CXU and state context (within that CXU). It may only be read or written in machine level.

In a privileged architecture system, user level read access to `mcx_selector` values could reveal goings-on in other software threads and thus facilitate side channel attacks.

In a privileged architecture with M/S/U levels, for example, what CSRs are required and what access permissions should they have?

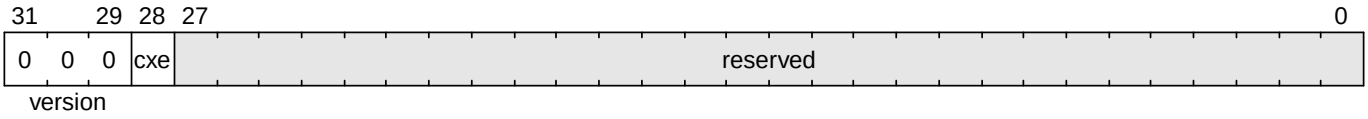


Figure 3. `mcx_selector` CSR OxBCO (version 0: legacy custom instructions))

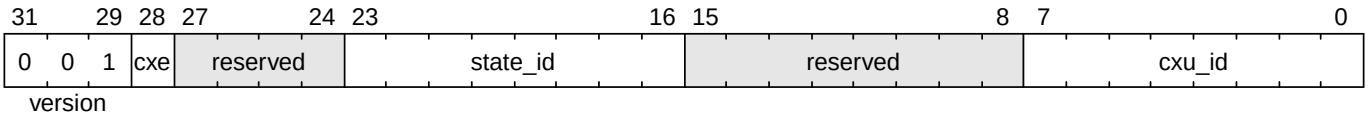


Figure 4. `mcx_selector` CSR OxBCO (version 1: extension multiplexing)

The `mcx_selector` CSR has the following fields:

`.version`: extension multiplexing version

- When `version=0`, disable composable extension multiplexing. When `cxe=0`, `custom-[0123]` instructions execute the CPU's built-in custom instructions and custom CSR addresses select the CPU's built-in custom CSRs. When `cxe=1`, `custom-[0123]` instructions and custom CSR accesses raise an illegal-instruction exception.
- When `version=1`, enable *version-1* composable extension multiplexing. The `cxu_id` and `state_id` fields select the current CXU and state context. When `cxe=0`, `custom-[012]` instructions issue CXU requests, and custom CSR accesses access CX CSRs, of the CXU and state context identified by `cxu_id` and `state_id`. When `cxe=1`, `custom-[012]` instructions and custom CSR accesses raise an illegal instruction exception.
- `version` values 2-7 are reserved.

`.cxe`: custom operation exception enable

- When (`version=0` or `version=1`) and `cxe=1`, a custom operation raises an illegal-instruction exception.

`.cxu_id`: select the hart's current CXU

- A valid `cxu_id` identifies a configured CXU.
- When enabled, when `cxu_id` does not identify a configured CXU, executing a custom operation instruction causes an invalid CXU_ID error. The `cx_status.CX` error bit is set and the instruction's destination register, if any, is zeroed.

`.state_id`: select the hart's current CXU's current state context

- A valid `state_id` identifies a state context of a CXU.
- When enabled, when `cxu_id` is valid, but `state_id` does not identify a state context of the current CXU, executing a custom operation instruction causes an invalid STATE_ID error. The `cx_status.IS` error bit is set and the custom operation instruction's destination register, if any, is zeroed.

No error occurs when `mcx_selector` is CSR-written with an invalid CX selector, i.e., when `.cxu_id` or `.state_id` are invalid. Rather, subsequently executing a custom operation instruction may cause a CXU_ID or STATE_ID error.

The hardware that detects these two errors might not be implemented by an extensible processor but rather in the CXU interconnect (bad `.cxu_id`) or in a selected CXU (bad `.state_id`).

The `version` field provides backwards compatibility with legacy custom extensions, and forwards compatibility with future CX systems. In future a new CX multiplexing version may be added, with a new layout and interpretation of selector fields and new means of decoding custom instruction fields into CXU requests. With seven non-zero values, it accomodates an additional extension multiplexing scheme every three years for twenty years.

The `cxe` field enables 1) software emulation of any built-in (legacy) custom instruction or custom CSR; 2) software emulation of any composable extension custom instruction or custom CSR; 3) transparent virtualization of CX state contexts; and 4) a representation of invalid selector sentinel value(s) to detect use of erroneous selector indices.

An illegal-instruction trap handler can emulate any absent built-in custom instruction or any custom instruction of a composable extension, then return to the following instruction.

Using CX access control (§2.9) CSRs, an OS can transparently virtualize many logical CX state contexts on fewer (or just one!) physical CX state contexts. When multiple CX libraries each try to open the same (e.g., singleton) CX state context, the OS can give each a unique CX selector index value, with all-but-one of their corresponding CX selector table entries set `cxe=1` to trap on first custom operation. Once such a selector index is used to select the thread's current CX, a custom operation incurs an illegal-instruction exception. The illegal-instruction trap handler determines which virtual CX state context currently has the physical CX state context, saves that CX state context, sets `cxe=1` on its selector table entry, restores the thread's current CX's state context, clears `cxe=0` for its selector table entry, rewrites `cx_index` with `cx_index` for the side-effect of updating `mcx_selector` with this selector table entry value with `cxe=0`, and returns from exception, reissuing the custom operation, which does not trap.

The selector's `cxe` field is subordinate to the `version` field so that future revisions of this specification may incorporate new trap behaviors and trap control bits.

The selector `0x3FFFFFFF` = `'{version:1, cxe:1, …, state_id:0xFF, …, cxu_id:0xFF}` is the canonical *invalid selector*.

Typically an OS will fill unused `scx_table[]` entries with this invalid selector to trap first custom operation use of an invalid selector index.

2.2.2. `cx_status` CSR 0x801: CX status

The `cx_status` CSR accumulates CX error flags, which include CX multiplexing errors as well as stateless and stateful CX custom operation errors. It may be written and read in all privilege levels.

Typical application software will write a CX selector to `mcx_selector` (perhaps indirectly via `cx_index`), write 0 to `cx_status`, execute some custom operation instructions, and read `cx_status` to determine if there were any errors.

Updates to `cx_status` are precise, *as if* each custom operation instruction issues and completes prior to the next, even if under the hood custom operations instructions are pipelined or complete out-of-order.

Since `cx_status` *behaves like* `fcsr`*, it should have similar high performance implementation considerations.*

For example, `cx_status` *bits only accumulate (i.e., are only set, never cleared, as a side-effect of custom operation instructions that go wrong). This may simplify a* `cx_status` *implementation if/when such instructions may complete out-of-order (e.g., when a first custom operation instruction is much longer latency than a second such instruction).*

Also, it is not until software reads `cx_status` *that previously issued custom instructions must complete, and even then, an out-of-order processor may value-speculate on* `cx_status` *to execute ahead of completion of previously issued custom operation instructions.*

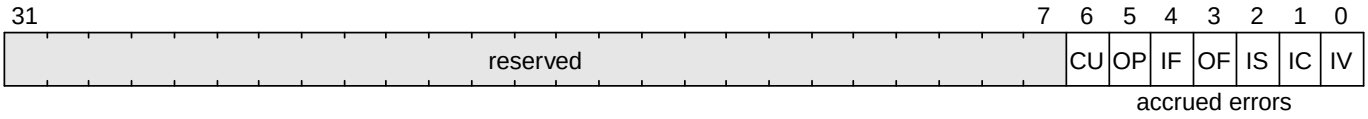


Figure 5. `cx_status` *CSR 0x801*

The `cx_status` CSR has the following fields:

`.IV`: invalid CX version error

- Set by a CSR-write to `mcx_selector`, or by a custom operation instruction, when `mcx_selector.version` is invalid. (For example, when new software writes a new selector type that old hardware does not implement.)

Arguably issuing a custom operation instruction with an invalid selector `version` *should raise an illegal-instruction exception. This can only arise when a fatally broken CX runtime or operation system issues new version selectors for old version hardware. Raising an illegal-instruction exception here would be consistent with V extension's* `vtype.vill` *behavior.*

`.IC`: invalid CXU_ID error

- Set by a custom operation instruction when `mcx_selector.cxu_id` is invalid.

`.IS`: invalid STATE_ID error

- Set by a custom operation instruction when `mcx_selector.cxu_id` is valid but `mcx_selector`.state_id is invalid.

`.OF`: state context is off error

- Set by a custom operation instruction when `mcx_selector.cxu_id` and `mcx_selector.state_id` are valid but the selected state context is in the *off* state.

`.IF`: invalid function ID error

- Set by a custom function instruction when `mcx_selector.cxu_id` and `mcx_selector.state_id` are valid but the instruction's CF_ID is invalid.
- Also set by a custom address CSR access instruction when `mcx_selector.cxu_id` and `mcx_selector.state_id` are valid but the custom CSR address is invalid.

`.OP`: CXU operation error

- Set by a custom operation instruction when `mcx_selector.cxu_id`, `mcx_selector.state_id`, and its

CF_ID/CSR address are valid but there is an error in the requested operation or its operands, in lieu of custom error state.

`.CU`: custom CXU operation error

- Set by a custom operation instruction of a stateful extension when `mcx_selector.cxu_id`, `mcx_selector.state_id`, and its CF_ID/CSR address are valid but there is an error in the requested operation or its operands, with custom error state available via the `cxs_error` CX CSR. The CX may also define additional CX instructions and CX CSRs that retrieve extended error information.

Should writing mcx_selector automatically zero `cx_status`? This shortens the code path to use an extension by one instruction but it precludes the use case of clearing errors, issuing a series of custom function instructions across multiple extensions, **then** *checking for errors.*

For simplicity we do not adopt this option.

How to best anticipate future changes to `cx_status`? One option: fields and behavior determined by hart's current CX version (`mcx_selector.version`). This becomes unwieldy when multiplexing between extensions switches different versions. One option: add a `cx_status.version` field, selecting an interpretation of `cx_status` CSR fields. Both options may lead to unnecessarily complicated error handling in software. Best option: only add new fields to it. Here simplest seems best.

2.2.3. `scx_table` CSR 0xBC1: CX selector table base

When CX access control (§2.9) is supported, the `MXLEN`-bit-wide `scx_table` CSR specifies the base address of the hart's CX selector table. The CSR may be read and written in machine level.

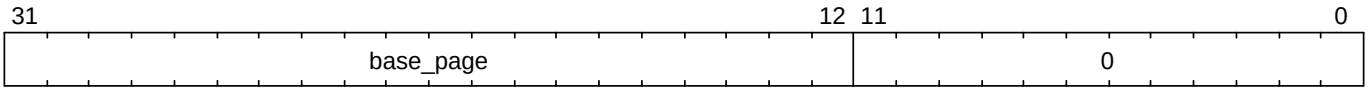


Figure 6. scx_table CSR 0xBC1 (when MXLEN=32)

CSR-writes to `scx_table` zero the twelve least significant bits of the table address, so a CX selector table address must be 4 KB aligned.

2.2.4. `cx_index` CSR 0x800: CX selector index

When CX access control (§2.9) is supported, the `cx_index` CSR selects an entry from the hart's CX selector table entry to write to the `mcx_selector` CSR. The CSR may be read and written in all privilege levels.

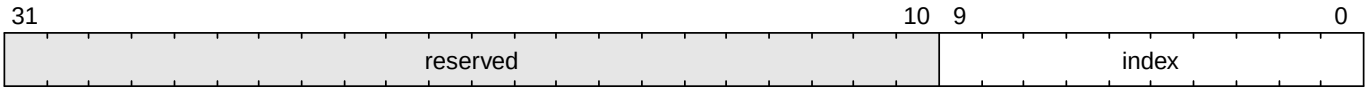


Figure 7. cx_index CSR 0x800

The 10-bit zero-extended index field specifies which entry in the hart's CX selector table (at the hart's `scx_table`) to use as the hart's current CX selector.

In response to CSR-write of `cx_index`, load the 32-bit CX selector at address (`scx_table + cx_index.index*4`) and CSR-write the CX selector to `mcx_selector`, performing the load and the CSR-write at the next higher privilege level, as if it were a `lw` instruction (and with a `lw` instruction's memory ordering rules) (§2.9).

Perhaps "at the next higher privilege level" should be "at machine mode privilege level".

2.2.5. Implicit CX-ISA CSR fences

There is an implicit fence between any CX-ISA CSR access and any series of custom operation instructions. All CX-ISA CSR accesses happen before any custom operation instructions which follow, and all custom operation instructions happen before any CX-ISA CSR accesses that follow.

For example, after issuing a long latency CF instruction, a CSR read of `cx_status` must await the CF instruction's CXU response.

2.3. Custom function instruction encodings

When `mcx_selector.version=1`, software issues CF instructions to the current state context of the current extension (i.e., of the current configured CXU) using R-type, I-type, and flex-type custom function instruction encodings.

For each instruction encoding, the CF instruction specifies the CF_ID, and source operand values, which may be two source registers, or one source register and one immediate value. R-type and I-type instructions always write a destination register whereas flex-type instructions never do so.

2.3.1. Custom-0 R-type encoding

Assembly instruction: `cx_reg cf_id,rd,rs1,rs2`

An R-type CF instruction issues a CXU request for a zero-extended 10-bit CF_ID `cf_id` with two source register operands identified by `rs1` and `rs2`. The CXU response data is written to destination register `rd`.

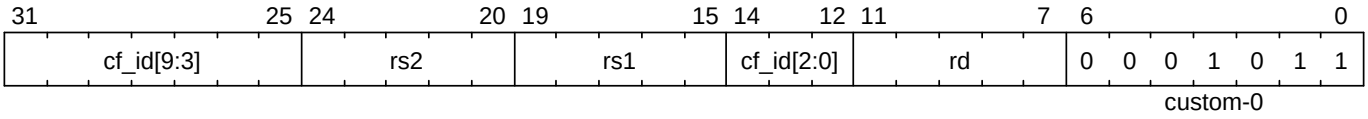


Figure 8. CX R-type instruction encoding

2.3.2. Custom-1 I-type encoding

Assembly instruction: `cx_imm cf_id,rd,rs1,imm`

An I-type CF instruction issues a CXU request for a zero-extended 3-bit CF_ID `cf_id` with one source register operand identified by `rs1` and a sign-extended 12-bit immediate value `imm`. The CXU response is written to destination register `rd`.

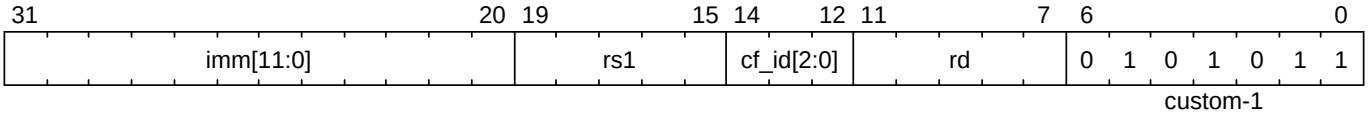


Figure 9. CX I-type instruction encoding

This encoding uniformly follows existing I-type instructions such as `addi` and `csrrw`, providing an immediate operand custom function instruction encoding at zero additional datapath cost.

2.3.3. Custom-2 flex-type encoding

Assembly instruction: `cx_flex cf_id,rs1,rs2`
Assembly instruction: `cx_flex25 custom`

A flex-type CF instruction issues a CXU request for a zero-extended 10-bit CF_ID `cf_id` with two source register operands identified by `rs1` and `rs2`. There is no destination register and CXU response *data* (but not a possible *error status*) is discarded. The instruction is executed purely for its effect upon the selected state context of the selected CXU.

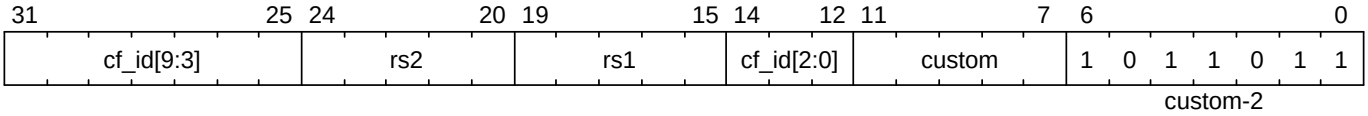


Figure 10. CX flex-type instruction encoding

Alternatively, equivalently, the `cx_flex25` form of instruction issues an arbitrary 25-bit custom instruction.

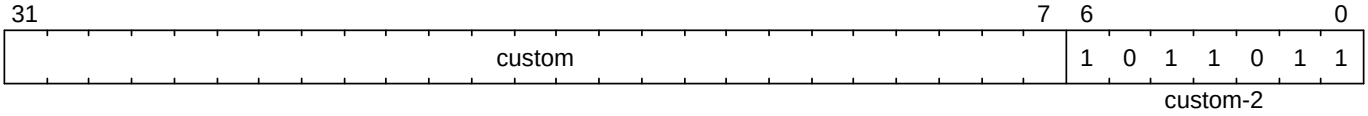


Figure 11. CX flex-type instruction alternate encoding

A flex-type CF instruction may be used with a CXU-L2 request's raw instruction field `req_insn` (3.4.6) to provide an arbitrary 32-7=25-bit custom request to a CXU. The absence of an (integer) destination register field is a feature that provides added, CPU-uninterpreted, custom instruction bits to a CXU.

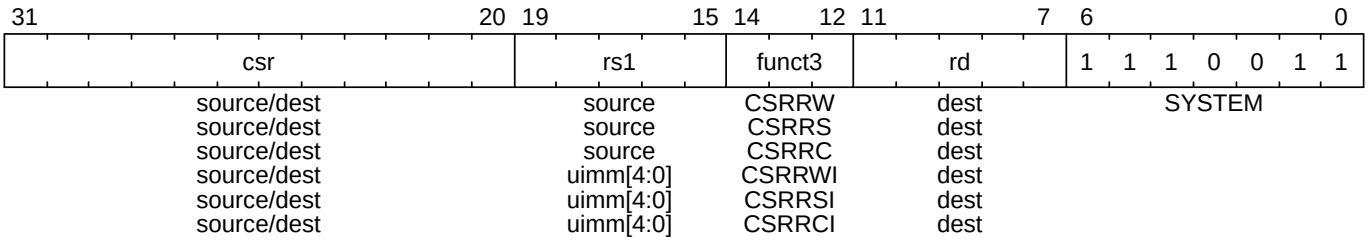
One disadvantage of this approach: when the selected CXU routinely discards the R[`rs1`] or R[`rs2`] operands, use of the flex-type custom function instruction can create a useless false dependency on the `rs1` and `rs2` registers, which may uselessly delay issue of the CF instruction in an out-of-order CPU core.

2.4. CX CSR accesses

When `mcx_selector.version=1`, CSR read/write instructions issue custom CSR accesses (i.e., *CX CSR* accesses) to the current state context of the current composable extension.

Per the Priv spec, an attempt to access a custom CSR without appropriate privilege level raises an illegal-instruction exception and an attempt to write a read-only custom CSR register raises an illegal-instruction exception.

Per the Zicsr spec, the only CSR access instructions that do not write to a CSR are `CSRRS` and `CSRRC` with `rs1=x0` and `CSRRSI` and `CSRRCI` with `uimm=0`. These are mapped to a CXU request pseudo-instruction `CSRR`, enabling a CXU to distinguish between a read-write access and a read-only access. There is no means to distinguish between a CX CSR read-write access and a write-only access: all CX CSR accesses are read accesses. There is no need and no means to distinguish between a CX CSR access using a source value from a source register and the equivalent access using a source value from the 5-bit `uimm` field.



In summary CX CSR access instructions are mapped into one of four CXU CSR access pseudo-instructions: `CSRR`, `CSRRW`, `CSRRS`, `CSRRR`.

2.5. Multiplexing custom instructions and custom CSR accesses across composable extensions

Figure 12 illustrates how custom function instruction and custom CSR accesses enjoy conflict-free composable extension composition via composable extension multiplexing. With multiplexing enabled (`mcx_selector.version=1`), when the CPU issues a custom operation instruction, it produces a CXU request from the fields of the instruction, two source operands from the register file and/or an immediate field of the instruction, and the `cxu_id` and `state_id` fields of `mcx_selector`. The CXU request may include the request ID cookie (defined by the CPU), the CXU_ID, STATE_ID, raw instruction, function (CF_ID or CSR access function), and operands. The CXU_ID identifies which CXU must process the request. The CXU includes state context(s) and a datapath. The STATE_ID selects the state context to use for this request. The CXU checks for errors in CXU_ID, STATE_ID, and function per 2.2.2, processes the request, possibly updating this state context, and produces a CXU response, which may include the same request ID cookie, a success/error status, and the response data. The CPU commits the custom operation instruction by updating `cx_status` (when response status is an error condition) and writing the response data to the destination register.

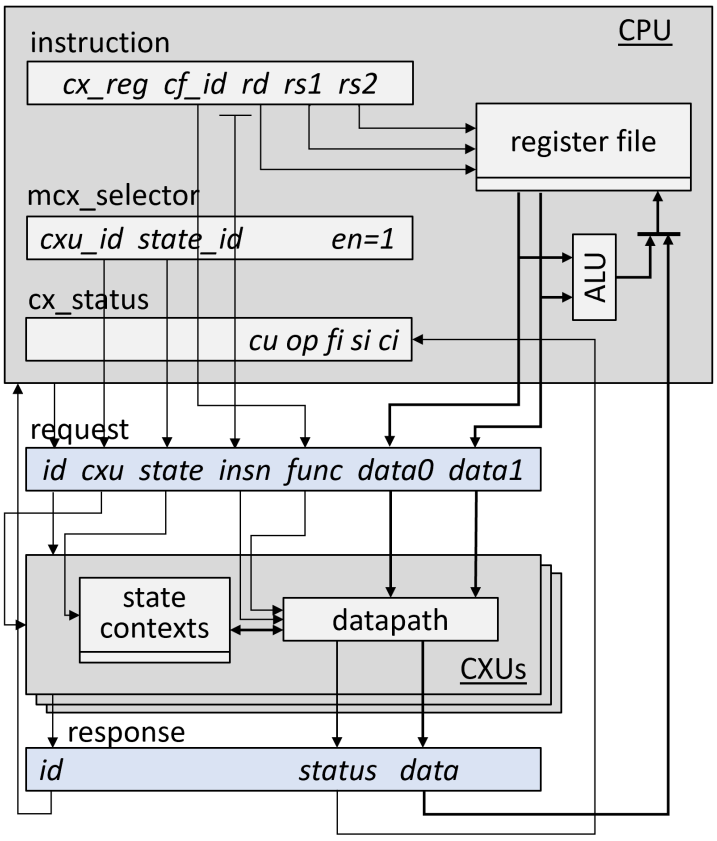


Figure 12. HW-SW interface: flow of information for execution of a custom operation instruction

Multiple custom operation instructions may be in flight at the same time, particularly in a system with pipelined CPUs or pipelined CXUs. A CPU may send a request ID and later receive the (same) ID back to correlate requests sent and responses received.

Table 1 defines the mapping from HW-SW interface entities, such as the `cf_id`, `rd`, `rs1`, `rs2`, `imm`, `csr`, `uimm` fields of a custom function instruction or a custom CSR access instruction and the `mcx_selector` and `cx_status` CSRs, to the CXU Logic Interface's request and response signals (§3.4).

Table 1. Mapping of HW-SW interface entities to CXU-LI signals

CXU-LI signal	← Source or → Destination
`req_id`	← CPU
`req_cxu`	← `mcx_selector.cxu_id`
`req_state`	← `mcx_selector.state_id`
`req_insn`	← `insn`
`req_func`	← `insn.cf_id` {`custom-[012]`} or `csr_func_id(insn)` {`csrr*`}
`req_data0`	← R[`insn.rs1`] {`custom-[012]` or `csrr[wsc]`} or `insn.uimm` {`csrr[wsc]i`}
`req_data1`	← R[`insn.rs2`] {`custom-[02]`} or `insn.imm` {`custom-1`} or `insn.csr` {`csrr*`}
`resp_id`	→ CPU
`resp_status`	→ `cx_status` bits
`resp_data`	→ R[`insn.rd`] {`custom-[01]` or `csrr*`}

A custom CSR access instruction (`CSRR`, `CSRRW`, `CSRRS`, `CSRRC`) maps to one of four `req_func` FUNC_IDs with msb set to one to distinguish them from custom function instructions' `CF_ID` function IDs. In general, `CXU_FUNC_ID_W = min(3, 1 + CF_ID_W)` bits.

```
enum { CSRR = 1<<CF_ID_W, CSRRW, CSRRS, CSRRC }; // msb set => CSR access
csr_read_only(insn) = (insn.funct3 == CSRR[SR][I]) && (insn.rs1 == 0);
csr_func_id(insn)   = CSRR + (csr_read_only(insn) ? 0 : insn.funct3[1:0]);
```

The signal that distinguishes custom function instruction from custom CSR access instruction CXU requests is conveyed as the MSB of CXU-LI's `req_func` FUNC_ID, rather than a separate one bit `req_csr_access` signal, to minimize the number of CXU-LI signal ports.

2.5.1. Precise exceptions

Custom function instruction execution preserves precise exception semantics. If an instruction preceding (in execution order) a custom operation instruction is an exception, the custom operation instruction does not execute, and has no effect upon architected state, including the `cx_status` CSR, and no effect on the current state context of the composable extension / CXU.

If an instruction following (in execution order) a custom operation instruction is an exception, the custom operation instruction executes, updating destination register, `cx_status`, and current state context, as appropriate.

A CPU may speculatively issue a custom operation instruction to a stateless CXU. Misspeculation recovery entails completing and discarding the CXU response. The custom operation instruction does not commit and there is no change to architectural state.

A CPU may not speculatively issue a custom operation instruction to a stateful CXU because the instruction may update the current state context and the CXU Logic Interface has no means to cancel a CXU request. In other words, a custom operation instruction of a stateful CXU, once issued, always commits.

Speculation is more than branch prediction. For example, in a pipelined CPU, instructions that follow a load or store instruction typically issue speculatively until the load or store is determined to not raise an access fault. Custom operation instructions of stateful CXUs must not issue in the wake of an instruction that may yet trap.

When a long latency custom operation instruction issues and a pipelined CPU continues issuing the following instructions in its wake, and one traps, the CPU nevertheless commits the custom operation instruction when the CXU eventually sends the response.

How can a CPU core determine dynamically whether a custom function instruction, or its composable extension, is stateless? (By definition custom CSR access instructions are always stateful.)

A software-defined approach could decorate the specification of a custom function to indicate whether it is stateful or stateless, and to encode this as an opcode bit in the `custom-[012]` instructions. Then a CPU may safely speculatively issue stateless CF instructions but non-speculatively issue stateful CF instructions.

A hardware-defined approach could add to the request and response streams defined in CXU-LI, a third stream, called the commit stream. This enables a CPU to speculatively issue any CF instruction and issue its CXU request, then later, when speculation is resolved, issue its commit token or cancel token. A stateful CXU, receiving and performing a CXU request, would defer from updating any CXU state until the request's corresponding commit token arrives.

2.6. CX State Context CX CSRs

Every stateful CX must implement four *CX State Context CX CSRs* that provide a uniform CX programming model:

- `cxs_error`: CX error;
- `scxs_status`: CX state context status;
- `scxs_index`: CX state context index;
- `scxs_data`: CX state context data at index.

The `cxs_` prefix indicates the CSR is a **CX s**tate context CX CSR.

These mandatory CX CSRs enable user-mode CX software to access a CX state context's error status, and enables a CX-agnostic supervisor-mode runtime or operating system to manage, initiailze, save, and reload any CX state context.

Consider abandoning the distinction between stateless and stateful CXs. Does it add significant benefit or clarity? A level 0 (combinational) CXU (sans `clk`), if provided, might just have `size=0` and/or `cxs_error=0`. That's fine.

2.6.1. CX error (cxs_error) CX CSR

The cxs_error CX CSR is a **WARL** UXLEN-bit user read-write CX CSR that conveys the error status of a CX state context. It may be updated in response to any CX operation and may also be read or written directly by software via a CSR read/write instruction. As usual §2.2.5 applies.

A CX may implement a cxs_error register with fewer than UXLEN bits (as few as zero bits). Unimplemented most-significant bits always read as 0.

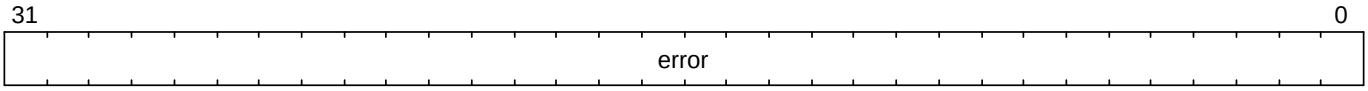


Figure 13. cxs_error: CX error register: user read/write CX CSR: 0x8FF (when UXLEN=32)

While cxs_error is inessential, it is mandatory to provide a **uniform** *way for CXs to convey extended error information arising from issuance of CX operations. This does not preclude a CX provding addtional CX CSRs or CX instructions to express additional aspects of a CX state context.*

Proposal: a cxs_error value of 0 indicates **no error** *(stateful CXs) or sometimes* **sorry, no error information** *(stateless CXs??).*

Each CX defines a behavior contract, indicating which CX operations set cxs_error and to which values. While there is at present no uniform specification for cxs_error values, we expect to discover and standardize recommended CX error categories and hence uniform cxs_error values.

It remains to be seen whether cxs_error should typically **accumulate** *errors or instead capture* **the last error** *(or success) condition.*

cxs_error is a reserved CX CSR: it may not be used by a CX for any other purpose. Therefore it may be safely used as a *probe* after a selector write, to check whether the selector addresses a valid CXU and state context:

```
csrw cx_index,x1        ; select some CXU and state context
csrw cx_status,x0       ; clear cx_status
csrr x0,cxs_error       ; probe, discarding error word
csrr x1,cx_status       ; retrieve cx_status
...                     ; cx_status.ci => invalid CXU_ID
...                     ; cx_status.si => invalid STATE_ID
```

2.6.2. CX state context status (scxs_status) CX CSR

The scxs_status CX CSR is a **WARL** SXLEN-bit supervisor read-write CX CSR that tracks and manages the state of a CX state context. It may be updated in response to any CX operation and may also be read or written directly by software via a CSR read/write instruction. As usual §2.2.5 applies.

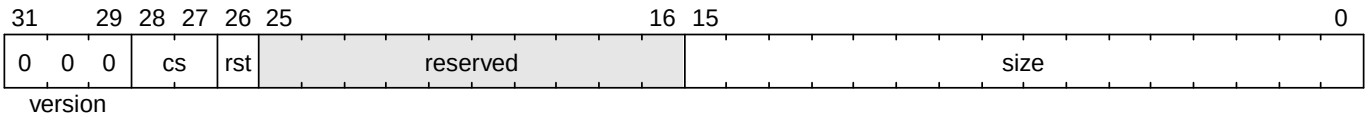


Figure 14. scxs_status: CX state context status register: supervisor read/write CX CSR:0x5FF

The `scxs_status` register has these fields:

`.cs`: state context status

- The state context status has four state values: { 0: `off`; 1: `initial`; 2: `clean`; 3: `dirty` }, corresponding to those of the `XS` field of the `mstatus` CSR, per the RISC-V Privileged ISA specification (Waterman et al., 2021, p. 26).
- On system reset, each state context of a stateful extension is in the `initial` state.
- A write `.cs=0` has the side effect of explicitly turning off the *current* state context. In this state, all CX custom instructions signal `CXU_ERROR_OFF` and set `cx_status.OF`.

In this state, what do CX CSR accesses do?

Is state preserved, disturbed, reset, or left undefined, by setting a state context to the `off` state, then to another state?

`.rst`: state context reset control/status

- A write `.rst=1` resets the *current* state context to its initial (power up) state. This may be instantaneous, or it may take many cycles.
- While the state context reset is in progress, `scxs_status.rst == 1`. In this state, as with the `off` state, all CX custom instructions signal `CXU_ERROR_OFF` and set `cx_status.OF`.

In this state, what do CX CSR accesses do?

- When a CXU implements mulitple CX state contexts, and state context reset requires many clock cycles, it is possible for software to rapidly select and reset multiple CX state contexts, even before the first reset completes.
- When a CX custom instruction or CX CSR access modifies any aspect of the current state context, its state context status automatically changes to `dirty`.

`.size`: state context size

- This WARL field specifies the *current* size (number of XLEN-sized words) of the current state context.
- Reads return the current size of the current state context.
- The value read need not equal the last value written.
- The value read must equal the last value read, unless there has been a CX operation since.
- Writes return the previous size and `cs` status of the current state context.
- Different CXU implementations of the same composable extension may have different state context sizes.
- Different state contexts of the same CXU may have different state context sizes.
- At different times, the same state context of the same CXU may have different state context sizes.

For most stateful CXUs, the size of a state context is fixed. For some stateful CXUs, the size of a state context may depend upon the sequence of CF instructions performed. For example, a stateful vector math CXU may provide CF instructions to allocate per-state context vector storage from a common, private shared pool, and may allow different state contexts to represent different sized vectors.

2.6.3. CX state context index (scxs_index) and CX state context data (scxs_data) CX CSRs

Together scxs_status.size, scxs_index, and scxs_data provide a CX-agnostic way for software to save a CX state context to a state context save record *blob*, and later to reload the context from the blob data.

Software should not interpret the blob data. In different systems, the different CXUs that implement a CX may use different blob formats and sizes.

The scxs_index CX CSR is a **WARL** SXLEN-bit supervisor read-write CX CSR that specifies the index and optional index auto-increment of access(es) of CX state context data.

A CX may implement its scxs_index CX CSR with fewer than 16 bits (as few as zero bits). Unimplemented most-significant bits always read as 0.

Its .index field specifies the index, within the current CX state context's context save data, that is accessed by a CSR read or write of the scxs_data CX CSR.

Its .incr field specifies that each time a CSR read or write of scxs_data commits, scxs_index.index is incremented by one. Thus software may read, e.g., the first three words of the current CX state context using:

```
li a0,0x80000000
csrw scxs_index,a0  // .index = 0
csrr a1,scxs_data   // .index = 1
csrr a2,scxs_data   // .index = 2
csrr a3,scxs_data   // .index = 3
```

Whenever the .index field is written with a value greater or equal to scxs_status.state, it is zeroed: scxs_index.index = 0.

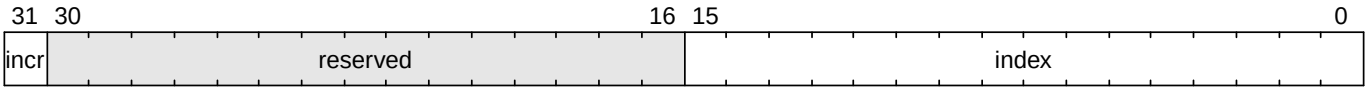


Figure 15. scxs_index: CX state context index register: supervisor read/write CX CSR:0x5FE

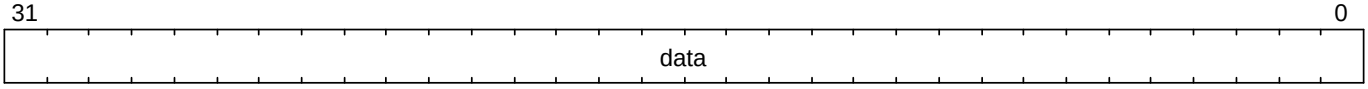


Figure 16. scxs_data: CX state context data register: supervisor read/write CX CSR:0x5FD (when SXLEN=32)

TODO: define reserved ranges of CX CSRs

2.7. CX/CXU identity CX CSRs

This entire CX/CXU info CX CSRs section is provisional and non-normative.

Two sets of optional machine-mode read-only CX CSRs allow software to interrogate the identity of the selected CX (CX_GUID) or its CXU (CXU_GUID).

These are machine mode only because they are only useful to system software and should not be used by user mode CX software. To select a CX, CX software has already discovered it via its CX_GUID. CX software must not depend on specific CXU implementations or versions — all implement a specific CX ISA contract, all should behave identically.

2.7.1. 32-bit CX_GUID (`mcx_guid[0123]`) CX CSRs

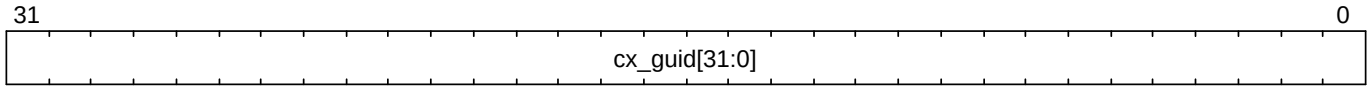


Figure 17. `mcx_guid0` CX_GUID-0 register: machine read-only CX CSR:0x8FF (when UXLEN=32)

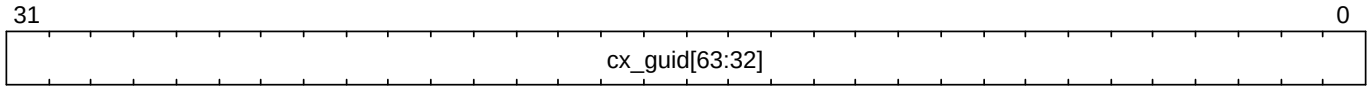


Figure 18. `mcx_guid1` CX_GUID-1 register: machine read-only CX CSR:0x8FF (when UXLEN=32)

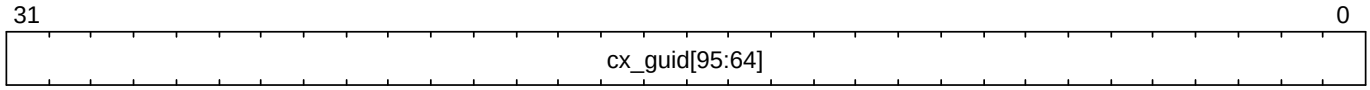


Figure 19. `mcx_guid2` CX_GUID-2 register: machine read-only CX CSR:0x8FF (when UXLEN=32)

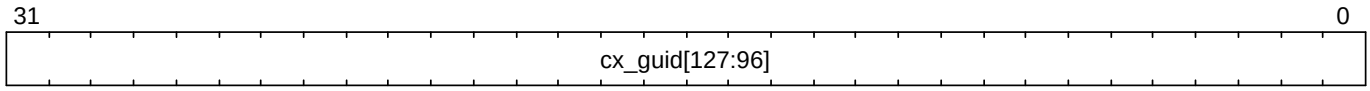


Figure 20. `mcx_guid3` CX_GUID-3 register: machine read-only CX CSR:0x8FF (when UXLEN=32)

2.7.2. 64-bit CX_GUID (`mcx_guid[01]`) CX CSRs

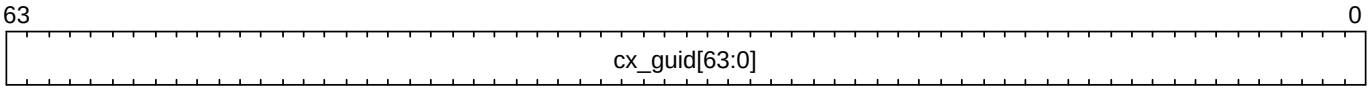


Figure 21. `mcx_guid0` CX_GUID-0 register: machine read-only CX CSR:0x8FF (when UXLEN=64)

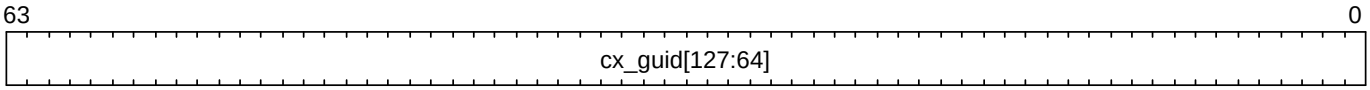


Figure 22. `mcx_guid1` CX_GUID-1 register: machine read-only CX CSR:0x8FF (when UXLEN=64)

2.7.3. 32-bit CXU_GUID (`mcxu_guid[0123]`) CX CSRs

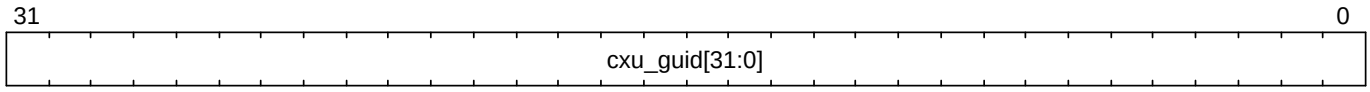


Figure 23. `mcxu_guid0` CXU_GUID-0 register: machine read-only CX CSR:0x8FF (when UXLEN=32)

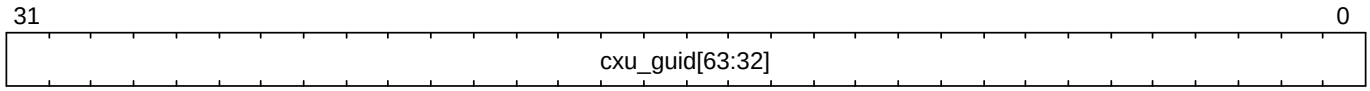


Figure 24. `mcxu_guid1` CXU_GUID-1 register: machine read-only CX CSR:0x8FF (when UXLEN=32)

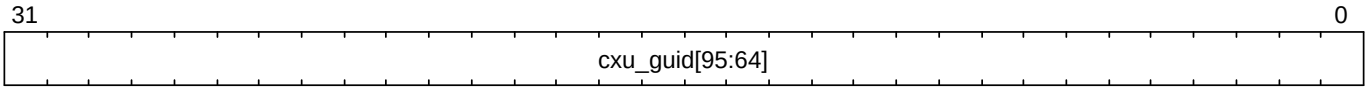


Figure 25. `mcxu_guid2` CXU_GUID-2 register: machine read-only CX CSR:0x8FF (when UXLEN=32)

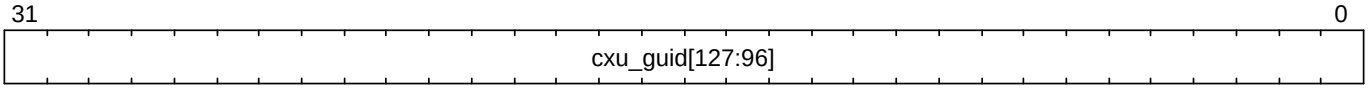


Figure 26. `mcxu_guid3` CXU_GUID-3 register: machine read-only CX CSR:0x8FF (when UXLEN=32)

2.7.4. 64-bit CX_GUID (`mcxu_guid[01]`) CX CSRs

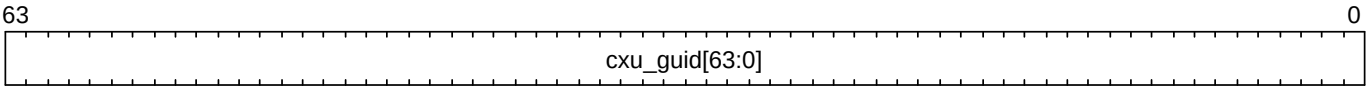


Figure 27. `mcxu_guid0` CXU_GUID-0 register: machine read-only CX CSR:0x8FF (when UXLEN=64)

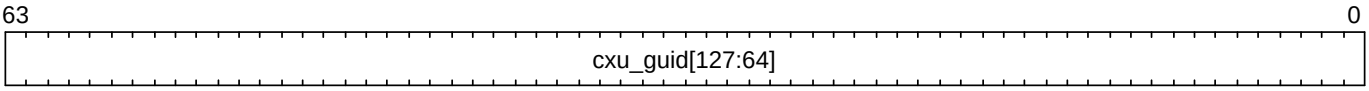


Figure 28. `mcxu_guid1` CXU_GUID-1 register: machine read-only CX CSR:0x8FF (when UXLEN=64)

2.8. Resource management and context switching

A software resource manager (e.g., thread pool, language runtime, language virtual machine, operating system, hypervisor) multiplexes software loci of execution (e.g., request, worker, actor, activity, task, fiber, continuation, thread, process), upon one or more hardware threads (*harts*).

The RISC-V per-hart state includes the program counter and integer register file, and optionally, floating point and vector register files, and various CSRs. CX-ISA extends per-hart state with the CX-ISA CSRs (§2.2) and the subset of the various configured state contexts of the stateful configured composable extensions allocated to that hart.

A CXU implementing a stateful composable extension is usually configured with one state context per hart in the entire system, but other configurations, including one context per locus, or a small pool of cooperatively or preemptively managed contexts, or several harts sharing one context, or one singleton context, are possible. Similarly, each CXU in a system may be configured with a different number of its state contexts.

The resource manager maintains the mapping of loci to harts, and the mapping of harts to CX state contexts. The resource manager consults a *System CXU Map* specifying the mapping CXU_IDs of the configured composable extensions of the system, and for each CX/CXU, the no. of state contexts it is configured with. A stateless CXU has zero contexts.

Over time, the resource manager must reset, save, and restore hart state, including its extension state contexts, to initialize a hart or to perform a context switch.

To reset hart state, for each CX state context of the hart, execute

```
cx_index = indices[i];
scxs_status = '{ cs:1, rst:1, size:0 };
while (scxs_status.rst)
    ;
```

This resets that state context to its initial state. It is also necessary to reset `cx_status`.

```
cx_status = 0;
```

To save hart state, first save `cx_status`, then for each CX state context of the hart, execute

```
save.cx_status = cx_status;
...
cx_index = indices[i];
status[i] = scxs_status;
```

to obtain `.size`, the size (in XLEN-bit words) of the state context blob for the selected state context. Allocate array `save[i][]` to store the serialized state context. For each word in `.size`, execute

```
scxs_index = '{ incr:0, 'index:0 };
for j in scxs.size:
  save[i][j] = scxs_data;
```

(When XLEN=32, use `sw`; when XLEN=64, use `sd`.)

To restore hart state, for each extension state context of the hart, first execute

```
lw a0, selectors[i]
csrw mcx_selector, a0
lw a0, status[i]
cx_write_status a0
```

to restore the state context status word. Then for each word in `status[i].size`, execute

```
lw/ld a0, save[i][j]
cx_write_state j,a0
```

to restore each word of the state context. Finally restore the saved `cx_status`.

```
lw a0,saved_cx_status
csrw cx_status,a0
```

When different CXUs implement the same composable extension, they may have different serializations, of different sizes.

Discuss preemption scenario where following context save, later restore, the locus moves to a different STATE_ID of a CXU. `cx_index` *may (but should not) change. However, resource manager must change* `mcx_selector`.

`cf_read_state` *and* `cf_write_state` *are random access. It is possible this induces unnecessary CXU hardware area. Perhaps specify a stream-out/stream-in extension instead.*

Discuss impact of mixed sized state contexts blobs upon system code and upon CXU design. Can a state context blob ever be too big to reload?

Is it necessary or helpful for CXU metadata to declare fixed- or variable-sized extension state contexts?

2.9. CX access control

Fully trusted software, executing in machine level, has full access to every CXU and every state context. Software may write an arbitrary CX selector value to the `mcx_selector` CSR, addressing any CXU and any state context. This is sufficient to implement composable extension multiplexing but does not provide means to protect one hart's CXUs' state from another hart, nor to limit a hart's access to a given CXU.

When a CPU implements user level and machine level privileged architecture, an attempt to CSR-write `mcx_selector` from user level generates an illegal-instruction exception.

Machine level software may provide to user level software an `ECALL` function to change `mcx_selector`.

Alternatively, the machine level illegal-instruction exception handler can determine whether the new CX selector value is valid for the user level code executing on the hart, optionally perform the CSR-write on its behalf, and return from exception.

Whether `ECALL` or exception handler, a detour into system level is prohibitively slow: reconfiguring composable extension multiplexing should take, at most, a few clock cycles.

The optional CX access control CSRs `scx_table` and `cx_index` allow less privileged *user code* to rapidly multiplex composable extensions, but only among those extensions and state contexts that it is granted access by more privileged *system code*.

CX access control requires at least user level and machine level privileged architecture, and a memory access control system, i.e., either RISC-V PMP or RISC-V virtual memory access control.

For each hart, the system code provisions a *CX selector table*, 4 KB aligned, comprising 1024 32-bit CX selectors, which is read/write to system code and inaccessible from user code.

Initially the table is initialized with 0 in the 0th entry, and the invalid selector (`0x10000000`) in every other entry. Selector index 0 selects table entry 0, with value `0x0000000` = `'{version:0, cxe:0}`, which disables CX multiplexing, thereby selecting the CPU's built-in custom instructions and custom CSRs. The system code CSR-writes the table address to the hart's `scx_table` CSR. Then in response to a system call requesting access to a composable extension and one of its state contexts, system code determines whether the access is granted. If so, it determines the CX selector value for it, allocates an entry for that CX selector value in the CX selector table, and returns the index (the *selector index)* of that entry to user code.

This index is analogous to a Unix file descriptor — an opaque token to a resource granted by system code.

To select this CX/CXU and its state, user code CSR-writes its index to `cx_index`. In response, the CPU loads from memory (at more privileged level) the CX selector word at that index in the selector table and copies it (CSR-writes it) to `mcx_selector` — no OS detour required.

This mechanism also conceals the specific CXU_ID and STATE_ID information from user code, precluding some possible side channel attacks.

3. Composable Extension Unit Logic Interface

The CXU-LI defines a set of common hardware logic signaling extensions enabling straightforward, correct composition of CPUs and CXUs. In the CXU-LI, a CPU is a requester and a CXU is a responder. The CPU sends a CXU request and eventually receives a CXU response. For each request there is exactly one response.

3.1. Definitions

A **CXU request (request)** is a group of CXU-LI signals that may include request flow control, REQ_ID, CXU_ID, FUNC_ID, STATE_ID, the raw instruction, and integer operands, produced by a CXU requester, conveying request data to a CXU.

A **CXU response (response)** is a group of CXU-LI signals that may include response flow control, REQ_ID, response status, and integer result, produced by a CXU, conveying response data to a CXU requester.

A **request ID (REQ_ID)** is a tag (a *magic cookie*) that correlates a CXU request and its corresponding CXU response.

A **CXU response status (response status, status)** is a CXU-LI success/error code produced by a CXU in response to receiving a CXU request, indicating success or else an error in the request's CXU_ID, FUNC_ID, STATE_ID, operation, or a composable extension specific error.

A **CXU requester** (requester) is a core that sends CXU requests to CXU(s) and receives CXU response(s) from CXUs.

A **CPU** is a CXU requester that implements RISC-V RV-I-Zicsr + *(CX-ISA extension)* instruction set, issues CXU requests upon issuing custom function instructions and custom CSR access instructions (collectively, *custom operation* instructions), and writes a destination register and the CX status CSR in response to CXU responses.

A **composable extension unit (CXU, responder)** is a core that implements one or more composable extensions. It receives CXU requests and sends CXU responses to CXU requesters. A CXU that also issues CXU requests is an **intermediary CXU**; otherwise it is a **leaf CXU**.

A **Switch CXU (switch)** is an intermediary CXU. For each request received, the switch either sends a response itself (e.g., a CXU_ERROR_CXU response) or arbitrates and forwards the request to a subordinate CXU, and later forwards the corresponding response to the original requester.

A **CXU feature level adapter (adapter)** is an intermediary CXU that receives requests and sends responses at one CXU-LI feature level and adapts them for and forwards them to a subordinate CXU with a lesser feature level.

A **configured system (system)** is a computer system including one or more CPUs and zero or more CXUs that implement a set of configured composable extensions.

3.2. Example configured system

Figure 29 illustrates a configured system composed of two CPUs and five CXUs, plus two switches and a level adapter for CXU_3. Each CPU has two harts. CXUs 0-2 are stateful and CXUs 3-4 are stateless. Each stateful CXU has one state context per hart. CXU_1 has an additional state context per hart for isolated stateful requests from CXU_2.

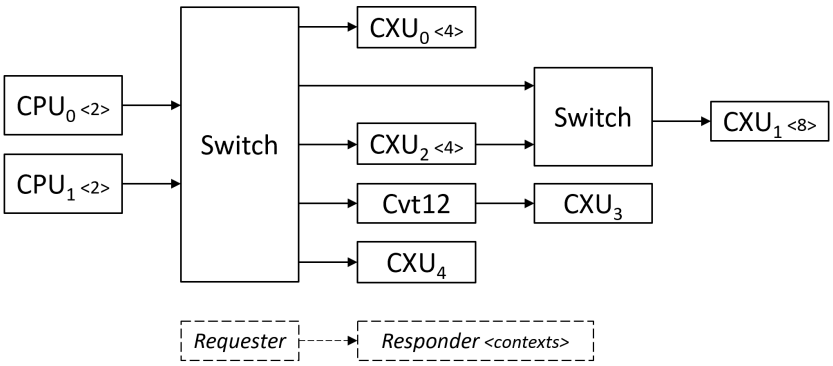


Figure 29. Configured system composed of two CPUs and five CXUs

In general, a CPU that issues one CXU request per cycle is directly coupled to one CXU, usually a switch CXU. A system of CXUs forms a directed acyclic graph.

3.3. CXU-LI feature levels

The CXU-LI is stratified into separate feature levels: -L0: combinational; -L1: fixed latency; -L2: variable latency; and -L3: reordering. Each feature level adds yet more CXU request and response signals, module ports, and behaviors to the feature level below it.

Stratification keeps simple use cases simple and frugal, and makes more complex use cases possible.

3.3.1. CXU-L0: combinational CXU

The CXU, which implements a stateless composable extension, computes a combinational function of the CXU request, sending a CXU response after some propagation delay. There is no flow control.

Example: combinational bitmanip unit with a population count custom function.

3.3.2. CXU-L1: fixed latency CXU

Each cycle, the CXU computes a function of the CXU request **and the specified state context, if any, updating the context**, sending a CXU response after a configured **fixed non-negative number of clock cycles**. With an initiation interval of II=1/cycle, there is no flow control of requests or responses.

Examples: stateless: a pipelined multiplier; stateful: a pipelined multiply-accumulate unit wherein the state is the current total.

Perhaps minimum II should also be configurable, e.g. CXU_INIT_INTERVAL=1+.

3.3.3. CXU-L2: variable latency CXU

The CXU computes a function of the CXU request and the specified state context, if any, updating the context, sending a CXU response, **in order, in a later clock cycle**. There is **request and response flow control** so the CXU can suspend receiving requests and the CPU can suspend receiving responses.

Example: a multiply-divide unit with a variable-latency multi-cycle divide, with early-out.

3.3.4. CXU-L3: reordering CXU

The CXU computes a function of the CXU request and the specified state context, if any, updating the context, and sending a CXU response in a later clock cycle. **Responses for requests with the same state context are sent in order, otherwise may be sent out of order**. There is request and response flow control.

CXU-L3 incorporates a request-response ID for the requester to correlate responses received to requests sent.

Example: a stateless, variable latency posit floating point unit, which, having received a pdiv request then a pmul request, responds out of order, sending the pmul response ahead of the pdiv response.

3.3.5. Feature levels summary

In summary, all CXU-LI feature levels have request and response function, data, and status. Level 0 is combinational. Level 1 adds clocking, fixed latency, and state contexts. Level 2 adds variable latency and request and response flow control and the raw instruction. Level 3 adds reordering. (Table 2.)

Table 2. CXU-LI feature levels summary

Level	CXU type	Req valid, func, data, resp data, status	Clock, reset, clock enable, state ID, resp valid	Req ready, resp ready, raw insn	Reordering, req ID
0	combinational	Y			
1	fixed latency	Y	Y		
2	variable latency	Y	Y	Y	
3	reordering	Y	Y	Y	Y

Compared to all possible subsets of features, CXU-LI levels are relatively simple and practical. Each level is a superset of lower levels, simplifying composition of dissimilar CXUs using common CXU feature level adapters.

3.4. CXU-LI signaling

CXU cores *of a particular feature level* implement a common set of request and response signals. Table 3 lists all CXU-LI signals of all feature levels in a canonical order: transaction signals (request/response valid, ready, REQ_ID), context (CXU_ID, STATE_ID), function (raw instruction, FUNC_ID), and data. The Level column indicates which levels introduce which signals. The Dir column indicates the signal direction from the perspective of a responder. The bit width of each bit vector is determined by a width parameter, configurable per CXU (§3.4.1).

Table 3. All CXU-LI signals, by feature level

Level	Dir	Port	Width Parameter	Description
1+	in	`clk`		clock
1+	in	`rst`		reset
1+	in	`clk_en`		clock enable
	in	`req_valid`		request valid
2+	out	`req_ready`		request ready
3	in	`req_id`	`CXU_REQ_ID_W`	request REQ_ID
	in	`req_cxu`	`CXU_CXU_ID_W`	request CXU_ID
1+	in	`req_state`	`CXU_STATE_ID_W`	request STATE_ID
	in	`req_func`	`CXU_FUNC_ID_W`	request FUNC_ID

Level	Dir	Port	Width Parameter	Description
2+	in	req_insn	CXU_INSN_W	request raw instruction
	in	req_data0	CXU_DATA_W	request operand data 0
	in	req_data1	CXU_DATA_W	request operand data 1 / CSR address
1+	out	resp_valid		response valid
2+	in	resp_ready		response ready
3	out	resp_id	CXU_REQ_ID_W	response ID
	out	resp_status	CXU_STATUS_W	response status
	out	resp_data	CXU_DATA_W	response data

All signals are positive-true logic.

3.4.1. CXU-LI configuration parameters

Table 4 presents CXU-LI bit vector width parameters and ranges of possible values.

Table 4. CXU-LI width configuration parameters

Level	Quantity	Width Parameter	Range	Default	Description
3	REQ_ID	CXU_REQ_ID_W	0-64	0	request/response ID width
	CXU_ID	CXU_CXU_ID_W	0-16	0	CXU_ID width
1+	STATE_ID	CXU_STATE_ID_W	0-16	0	STATE_ID width
	FUNC_ID	CXU_FUNC_ID_W	3-11	11	FUNC_ID width
2+	insn	CXU_INSN_W	0, 32	0	raw instruction width
	data	CXU_DATA_W	32, 64	32	request/response data width
	status	CXU_STATUS_W	3	3	response status width

Zero width bit vectors are problematic in some HDLs. Parameter signals declared 0-bits wide should nevertheless be declared [0:0], driven 1'b0 by sender, and ignored by receiver.

Table 5 presents other CXU configuration parameters.

Table 5. CXU-LI: other CXU configuration parameters

Level	Parameter	Range	Default	Description
	CXU_LI_VERSION	24'h010000	24'h010000	CXU-LI version; 24'h01_00_00 == 1.00.00
	CXU_N_CXUS	1+	1	number of CXUs at/below this CXU
1+	CXU_N_STATES	0+	0	number of composable extension state contexts
1	CXU_LATENCY	0+	1	latency (clock cycles) from a request to its response
1	CXU_RESET_LATENCY	0+	0	min. latency (clock cycles) from negation of reset to first request

CXU_LI_VERSION indicates the version of the CXU-LI signals and semantics in effect, using semantic versioning semver.org, encoded as 24'hxx_yy_00: (major=xx,minor=yy,patch=00). Since CXU_LI_VERSION is an extension specification and not an implementation, there is never a patch level. See also §1.6.

CXU_LI_VERSION *anticipates subsequent evolution of CXU-LI.*

CXU_N_CXUS is the number of logical CXUs at/below this CXU. For a leaf CXU this may be more than one when the

CXU implements multiple composable extensions (including multiple versions of one composable extension).

`CXU_N_STATES` is the number of composable extension state contexts for every stateful extension implemented by this CXU. It must be 0 if every composable extension implemented by the CXU is stateless. It must be 1+ if any composable extension implemented by the CXU is stateful. When a leaf CXU implements multiple stateful composable extensions, i.e. `CXU_N_CXUS>1`, each must be configured with the same number of state contexts.

`CXU_LATENCY` and `CXU_RESET_LATENCY` are specific to CXU-L1 fixed latency CXUs. See §3.3.2.

3.4.2. Clock, reset, clock enable

CXU-L0 is combinational. Other feature levels' signaling is (mostly) synchronous to rising edge (*posedge*) of `clk`.

When the reset input signal `rst` is asserted on posedge `clk`, it supersedes all other CXU-LI signaling. Any request processing in progress is abandoned, all internal state is reset, and `req_ready` and `resp_valid` output signals, if present, are negated. A CXU-L1 CXU (which does not have a `req_ready` output) must be ready to receive its first request after no more than its configured `CXU_RESET_LATENCY` clock cycles following negation of `rst`.

A clock enable input signal `clk_en` facilitates clock gating of a CXU. When `clk_en` is asserted on posedge `clk`, synchronous elements of the CXU (i.e., memories, registers, flip-flops) may change. When `clk_en` is negated on posedge `clk`, no changes may occur to synchronous elements of the CXU. CXU operation is suspended. Therefore, when negating `clk_en`, a CXU requester must disregard all CXU output signals, esp. `req_ready` and `resp_valid`.

In the twilight of Moore's Law, energy efficiency is a first order design concern, and it is a shame to burn power computing routinely discarded results.

All modern FPGAs enable simple clock gating via free `clk_en` inputs on all LUT-cluster D flip-flops.

If a requester never clock gates a CXU with `clk_en`, it should assert `clk_en` with a constant `'1`. FPGA and ASIC implementation tools typically optimize away such signals and their D flip-flop clock enables.

Perhaps provide another configuration parameter `CXU_USE_CLK_EN=0/1` to configurably-ignore `clk_en`. This could simplify conversion of preexisting RTL function units, sans `clk_en` gating, into new CXUs.

3.4.3. Request and response valid-ready flow control

CXU-L2 and -L3 provide CXU request and response channel synchronous valid-ready flow control. For each channel, the sender may assert data and a positive-true data `valid` signal indicating it is ready to send data. The receiver may assert a positive-true `ready` signal indicating it is ready to receive data. On posedge `clk`, if both `valid` and `ready` are asserted, data transfers from sender to receiver; otherwise, no transfer occurs during that clock cycle.

Once a sender asserts data and asserts data `valid` on posedge `clk`, it must assert the same data and `valid` on each subsequent posedge `clk` until the receiver asserts `ready` and the transfer occurs.

A `valid` output must not depend (via combinational logic) upon a `ready` input. However, a `ready` output may depend upon a `valid` input.

With request and response flow control, a requester must not indefinitely negate `resp_ready` in response to a responder negating `req_ready`.

This precludes a potential cyclical wait deadlock in a composed system.

3.4.4. Response status / error checking

At any feature level, in response to receiving a CXU request, the CXU error-checks the request data, performs the request, and outputs the first (i.e., lowest numbered) `[2:0] resp_status` condition that applies:

Table 6. CXU response status values and conditions

Name	Value	Condition
`CXU_OK`	0	no errors occurred processing request
`CXU_ERROR_CXU`	1	`req_cxu` is not a CXU_ID implemented by CXU
`CXU_ERROR_STATE`	2	`req_state` is not a valid STATE_ID for `req_cxu`
`CXU_ERROR_OFF`	3	`req_state` is valid but this *serializable* state context is in the *off* state
`CXU_ERROR_FUNC`	4	`req_func` is not a valid CF_ID / `req_data1` is not a valid CSR address / write access to a read-only CSR
`CXU_ERROR_OP`	5	request operand(s) or state are a domain error for the custom function or custom CSR access
`CXU_ERROR_CUSTOM`	6	request causes a custom error (of a serializable composable extension)

When parameter `CPU_CXU_ID_W=0`, `req_cxu` is ignored: no `CXU_ERROR_CXU` errors.

When parameter `CPU_STATE_ID_W=0`, `req_state` is ignored: no `CXU_ERROR_STATE` errors.

`STATE_ID=0` is the only valid STATE_ID for the CXU of a stateless composable extension.

CXU state may change if and only if the response status is one of `CXU_OK`, `CXU_ERROR_OP`, or `CXU_ERROR_CUSTOM`.

When a response status is `CXU_ERROR_CUSTOM`, the CXU should update the specified state context's custom error status as a side effect of the request. Otherwise, a CX library may be surprised to observe that the custom error bit `cx_status.CU` is set without observing a corresponding error bit upon retrieving (via `cx_read_status`) its state context's error state.

In response to receiving `resp_status` of `CXU_ERROR_CXU`, `CXU_ERROR_STATE`, `CXU_ERROR_OFF`, or `CXU_ERROR_FUNC`, a CPU ignores `resp_data` and uses zero as the result of the CF instruction.

When a CF instruction writes a destination register, (i.e., `custom-[01]` but not `custom-2`), the result of the CF instruction is written to the register, irrespective of the CXU response status.

Can certain errors suppress destination register writes? No: data dependent writeback cancelation is irregular and unnecessarily complicates out of order CPUs.

Together these rules ensure { CXU, state, function } ID errors are well behaved at the hardware-software extension. By making the CPU responsible for zeroing such results, each CXU in a system's CXU DAG need not incur redundant logic and delay to respond `resp_data=0` on these three errors. For synchronously signaled CXU-LI levels, in an FPGA, with reset-able flip-flops, a registered `resp_data` input may be zeroed for negligible cost.

3.4.5. Function ID

The CXU request FUNC_ID `req_func` indicates the custom operation to perform: when the most-significant bit of `req_func` is 0, the remaining bits are the CF_ID of a custom opcode instruction; when the most-significant bit of `req_func` is 1, bits `req_func[1:0]` encode one of four CSR access requests:

Table 7. FUNC_ID CSR access requests

Value	Req.	Description	Behavior
`1:0`	`CSRR`	read-only access	`resp_data = CSRs[req_data1];`
`1:1`	`CSRRW`	read-write access	`resp_data = CSRs[req_data1]; CSRs[req_data1] = req_data0;`
`1:2`	`CSRRS`	read-set access	`resp_data = CSRs[req_data1]; CSRs[req_data1] |= req_data0;`
`1:3`	`CSRRC`	read-clear access	`resp_data = CSRs[req_data1]; CSRs[req_data1] &= ~req_data0;`

The minimum `CXU_FUNC_ID_W` is 3: 1-bit MSB + 2-bit CSR access request type; the maximum is 11: 1-bit MSB + maximum 10-bit `CF_ID_W`.

Perhaps the spec should require "When the most-significant bit of `req_func` is 1, bits `req_func[CXU_FUNC_ID_W-2:2]` must be 0."

3.4.6. Raw instruction

At CXU-LI feature level 2, or higher, CXU requests may be configured (`CXU_INSN_W=32`) to include the raw instruction word (`req_insn`) of the custom operation instruction that issued the CXU request, or all zeroes otherwise. A CXU may use the raw instruction data to help perform the custom operation, or it may ignore the raw instruction entirely.

The raw instruction complements the CF_ID-derived FUNC_ID `req_func` identifier. CF_ID is the preferred, future proof way to select a custom function. It is ISA neutral and abstracts the CPU away from CXU, and potentially reduces verification complexity.

However, access to the raw CF instruction word can enable additional use cases. As an example, consider a CXU with a private vector, matrix, or complex number register file. When this CXU receives a CXU request including its raw instruction word, it may opt to ignore either or both of the two integer request operands `req_data0` and `req_data1`, and instead partially decode the raw instruction word to recover `rs1` and `rs2` fields, even `rs3` if there are spare custom instruction bits, to determine which of its CXU register file entries to read. Similarly, the CXU can decode the raw instruction word to recover an `rd` field to determine which CXU-private register file entry to write back and whether to do so.

This feature is best used with the `custom-2` flex instruction format which has no `rd` destination register field, freeing those bits for arbitrary uses.

Does raw instruction access merits security threat modeling? Imagine adversarial CXUs, snoopily watching the dynamic instruction stream go by, even when `req_valid` is negated.

At present the custom instructions and CSR access instructions are 32b instructions. If this changes, CXU_INSN_W may have values other than 0 and 32.

Half-baked idea (not recommended): Imagine a dynamic facility by which any arbitrary instruction word, not just `custom-[012]` format instructions, may be a CF instruction, issued to a CXU. This might be a table of (mask,pattern) tuples, or a 32-bit `mcx_opcodes_mask` CSR bit vector of 5-bit major opcodes, identifying instructions to divert to the current CXU. Or perhaps, in the hardware domain, a CPU might first issue each instruction to the current CXU, and only execute the instruction in the CPU if the CXU delegates it back to the CPU.

3.4.7. Request-response ID

CXU-LI feature level 3 (reordering CXU) includes a request-response ID REQ_ID, a `REQ_ID_W` -bit signal used by requesters to correlate responses received with requests sent. With each request, the CXU receives the REQ_ID as `req_id`, and later, with each response, the CXU sends back the same REQ_ID as `resp_id`. For each request/response pair, the CXU must send the requester the identical request-response ID value that the requester previously sent to the CXU.

Operation and behavior of a CXU must not depend in any way upon any `req_id` value received, except to receive it and later to return it to the requester.

An out-of-order completion CPU may send a REQ_ID indicating the destination register of the request, and rely upon it when the response eventually returns.

3.5. CXU-L0 combinational CXU signaling

A combinational CXU, which implements a stateless composable extension, computes a combinational function of the CXU request, sending a CXU response after some propagation delay. There is no flow control.

3.5.1. CXU-L0 configuration parameters

Table 8. CXU-L0 configuration parameters

Parameter	Description
`CXU_LI_VERSION`	CXU-LI version number
`CXU_N_CXUS`	number of CXUs at/below this CXU

For `CXU_LI_VERSION` and `CXU_N_CXUS`, see §3.4.1.

3.5.2. CXU-L0 signals

Table 9. CXU-L0 signals

Dir	Port	Width Parameter	Description
in	`req_valid`		request valid
in	`req_cxu`	`CXU_CXU_ID_W`	request CXU_ID: selects the requested CXU
in	`req_func`	`CXU_FUNC_ID_W`	request FUNC_ID
in	`req_data0`	`CXU_DATA_W`	request operand data 0
in	`req_data1`	`CXU_DATA_W`	request operand data 1
out	`resp_status`	`CXU_STATUS_W`	response status
out	`resp_data`	`CXU_DATA_W`	response data

CXU-L0 signaling is asynchronous. CXU outputs are pure combinational functions of CXU inputs.

CXU-L0 has no `resp_valid` signal because it would just reflect `req_valid`.

3.5.3. CXU-L0 signaling protocol

Protocol:

1. Request transfer
   a. Requester asserts CXU request signals `req_*` and asserts `req_valid`.
   b. CXU asynchronously receives CXU request.
2. Response transfer
   a. CXU performs steps 1, 2, 4, and 6 of response status / error checking per §3.4.4, and asserts `resp_status`.
   b. CXU asserts `resp_data`, a combinational custom function of the operands.
   c. Requester asynchronously receives CXU response.

As a CXU-L0 CXU is combinational, its delay folds into to the path timing analysis of its requester.

3.5.4. CXU-L0 example

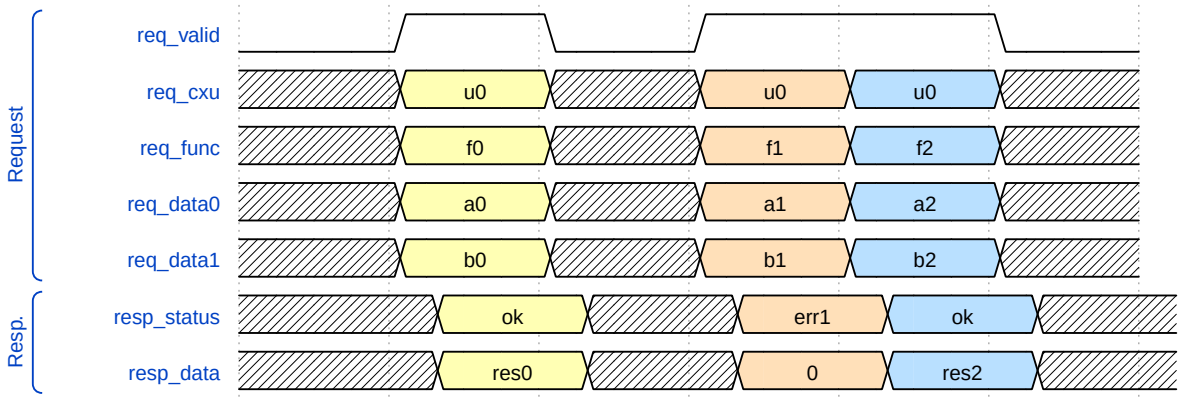


Figure 30. Example CXU-L0 signaling protocol waveform

Figure 30 is an example waveform for three CXU-L0 requests and responses, arising from executing custom function instructions `f0(a0,b0)`, `f1(a1,b1)`, and `f2(a2,b2)`. All three instructions issue to the same CXU `u0`. Function `f1` incurs an error.

3.6. CXU-L1 fixed latency CXU signaling

Each cycle, a fixed latency CXU computes a function of the CXU request **and the specified state context, if any, updating the context**, sending a CXU response after a configured **fixed non-negative number of clock cycles**. With an initiation interval of II=1/cycle, there is no flow control of requests or responses.

Lacking request flow control, if a CXU-L1 CXU is configured with multiple requesters, requesters must not send multiple simultaneous requests.

3.6.1. CXU-L1 configuration parameters

Table 10. CXU-L1 configuration parameters

Parameter	Description
CXU_LI_VERSION	CXU-LI version number
CXU_N_CXUS	number of CXUs at/below this CXU
CXU_N_STATES	number of composable extension state contexts
CXU_LATENCY	latency (clock cycles) from a request to its response
CXU_RESET_LATENCY	minimum latency (clock cycles) from negation of reset to first request

For CXU_LI_VERSION, CXU_N_CXUS, and CXU_N_STATES, see §3.4.1.

CXU_LATENCY, specific to CXU-L1, configures the CXU latency, which is the number of clock cycles from receiving a request to sending a response, of every custom function implemented by the CXU. CXU_LATENCY=0 configures the CXU to respond to the request in the same clock cycle.

A CFI-L1 CXU with CXU_LATENCY=0 resembles a CXU-L0 combinational CXU, except it may implement a stateful composable extension.

Example: an extended precision arithmetic CXU which implements add_save_carry *and* add_with_carry_save_carry *CF instructions. Like an ALU, this has zero cycle latency, but supports additional state context(s), each with a carry bit.*

CXU_RESET_LATENCY, specific to CXU-L1, configures the CXU reset latency, which is the minimum number of clock cycles from negation of rst to first assertion of req_valid. CXU_RESET_LATENCY=0 configures the CXU to be ready for a CXU request in the same cycle that rst is first negated.

3.6.2. CXU-L1 signals

Table 11. CXU-L1 signals

Dir	Port	Width Parameter	Description
in	clk		clock
in	rst		reset
in	clk_en		clock enable
in	req_valid		request valid
in	req_cxu	CXU_CXU_ID_W	request CXU_ID
in	req_state	CXU_STATE_ID_W	request STATE_ID
in	req_func	CXU_FUNC_ID_W	request FUNC_ID
in	req_data0	CXU_DATA_W	request operand data 0
in	req_data1	CXU_DATA_W	request operand data 1
out	resp_valid		response valid
out	resp_status	CXU_STATUS_W	response status
out	resp_data	CXU_DATA_W	response data

3.6.3. CXU-L1 signaling protocol

CXU-L1 is (mostly) synchronous to posedge clk when CXU_LATENCY>0. See §3.4.2.

Protocol:

1. Request transfer.

a. Requester asserts CXU request signals `req_*` and asserts `req_valid`.

b. `CXU_LATENCY=0`: CXU receives CXU request asynchronously.
`CXU_LATENCY>0`: CXU receives CXU request on posedge `clk`.

2. Custom function execution.

a. CXU performs response status / error checking per §3.4.4.

b. CXU performs a function of the operands and the selected state context.

c. CXU may update the selected state context, logically prior to any updates from subsequent requests.

3. Response transfer.

a. `CXU_LATENCY=0`:

i. CXU asserts CXU response signals `resp_valid`, `resp_status`, and `resp_data` asynchronously.

ii. Requester receives CXU response asynchronously.

b. `CXU_LATENCY>0`:

i. After (`CXU_LATENCY-1`) cycles, CXU asserts `resp_valid`, `resp_status`, and `resp_data`.

ii. Requester receives CXU response on posedge `clk`.

3.6.4. CXU-L1 example

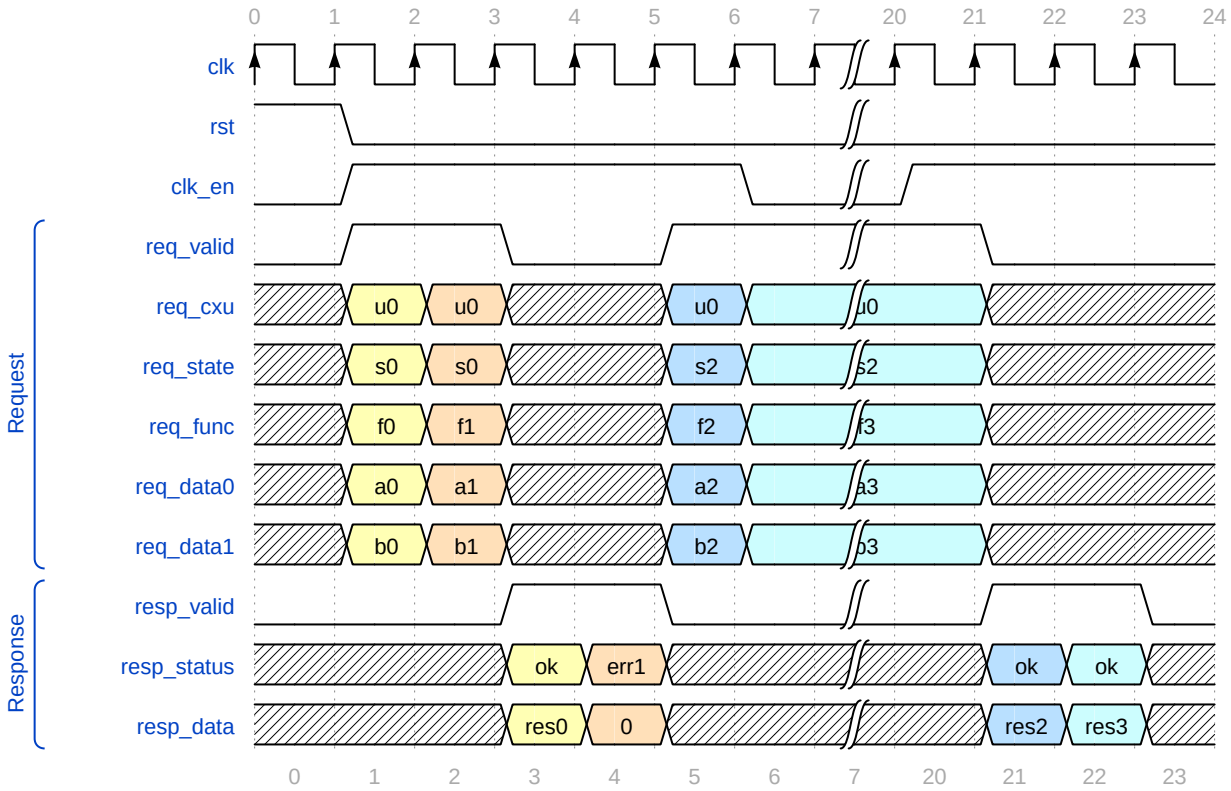


Figure 31. Example CXU-L1 signaling protocol waveform (`CXU_LATENCY=2`, `CXU_RESET_LATENCY=0`)

Figure 31 is an example waveform for four CXU-L1 CXU requests and responses, arising from executing four custom function instructions `f0`-`f3`. Since `CXU_RESET_LATENCY=0`, the CXU is ready for request `f0` in cycle 1, the same cycle `rst` is negated. With `CXU_LATENCY=2`, each response occurs 2 (enabled) clock cycles after each request is received. Each instruction issues a CXU request to the same CXU `u0`. Instructions `f0` and `f1` use state context s0; `f2` and `f3` use state context `s2`. Request `f1` results in an error response. With `clk_en` negated in cycles 6-19, the CXU is frozen

until cycle 20, when it finally receives the `f3` request. The `f2` response, otherwise due in cycle 7, is also delayed, until cycle 21.

3.7. CXU-L2 variable latency CXU signaling

A variable latency CXU computes a function of a CXU request and the specified state context, if any, updating the context, sending a CXU response, in order, in a later clock cycle. There is **request and response flow control** so the CXU can suspend receiving requests and the requester can suspend receiving responses.

When the requester is a CPU, use of CXU-L2 allows the CPU to delay receipt of a CXU response. This affords the CPU pipeline greater flexibility to dynamically prioritize other units' accesses to register file write port(s). Conversely, CXU-L2 can complicate design of a CXU, which may have to respond to negated `resp_ready` by buffering the response in an output FIFO or by applying back pressure through its processing pipeline, or negate `req_ready` to delay receipt of new requests.

3.7.1. CXU-L2 configuration parameters

Table 12. CXU-L2 configuration parameters

Parameter	Description
`CXU_LI_VERSION`	CXU-LI version number
`CXU_N_CXUS`	number of CXUs at/below this CXU
`CXU_N_STATES`	number of composable extension state contexts

For `CXU_LI_VERSION`, `CXU_N_CXUS`, and `CXU_N_STATES`, see §3.4.1.

3.7.2. CXU-L2 signals

Table 13. CXU-L2 signals

Dir	Port	Width Parameter	Description
in	`clk`		clock
in	`rst`		reset
in	`clk_en`		clock enable
in	`req_valid`		request valid
out	`req_ready`		request ready
in	`req_cxu`	`CXU_CXU_ID_W`	request CXU_ID
in	`req_state`	`CXU_STATE_ID_W`	request STATE_ID
in	`req_func`	`CXU_FUNC_ID_W`	request FUNC_ID
in	`req_insn`	`CXU_INSN_W`	request raw instruction
in	`req_data0`	`CXU_DATA_W`	request operand data 0
in	`req_data1`	`CXU_DATA_W`	request operand data 1
out	`resp_valid`		response valid
in	`resp_ready`		response ready
out	`resp_status`	`CXU_STATUS_W`	response status
out	`resp_data`	`CXU_DATA_W`	response data

3.7.3. CXU-L2 signaling protocol

CXU-L2 is synchronous to posedge `clk`. See §3.4.2. CXU-L2 includes the request's raw instruction. See §3.4.6.

Protocol:

1. Request transfer.
   a. Requester asserts CXU request signals `req_*` and asserts `req_valid`.
   b. Responder may assert `req_ready`.
   c. CXU receives CXU request on posedge `clk` when `req_valid` and `req_ready` are both asserted, per §3.4.3.
2. Custom function execution.
   a. CXU performs response status / error checking per §3.4.4.
   b. CXU performs a function of the operands and the selected state context.
   c. CXU may update the selected state context, logically prior to any updates from subsequent requests.
3. Response transfer.
   a. Prior to issuing responses from subsequent requests (i.e., in order of requests) CXU asserts `resp_status` and `resp_data` and asserts `resp_valid`.
   b. Requester may assert `resp_ready`.
   c. Requester receives CXU response on posedge `clk` when `resp_valid` and `resp_ready` are both asserted, per §3.4.3.

3.7.4. CXU-L2 example

Figure 32 is an example waveform for four CXU-L2 CXU requests and responses, arising from executing four CF instructions `f0`-`f3`. (Assume `CXU_INSN_W=0`, no `req_insn`.) Each instruction issues a CXU request to the same CXU `u0`. Instructions `f0` and `f1` use state context `s0`; `f2` and `f3` use state context `s2`.

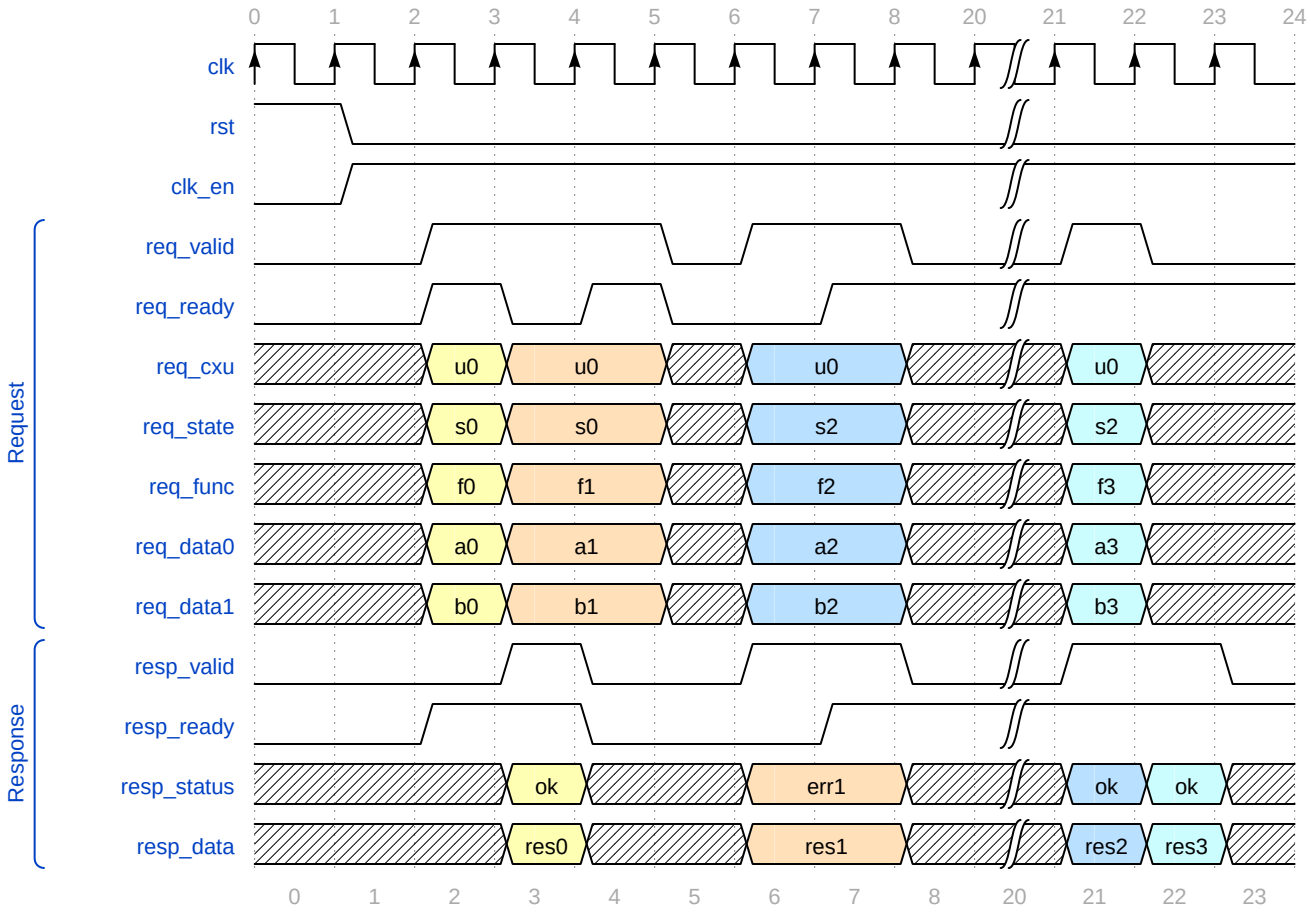


Figure 32. Example CXU-L2 signaling protocol waveform

The CXU receives request `f0` in cycle 2 and responds in cycle 3.

Requester asserts request `f1` in cycle 3, but it is not received by the CXU until it asserts `req_ready` in cycle 4. The CXU sends the `f1` response in cycle 6, an error response, a latency of 2 cycles. Requester asserts `resp_ready` and receives the response in cycle 7.

Requester asserts request `f2` in cycle 6, but it is not received by the CXU until it asserts `req_ready` in cycle 7. The CXU responds to `f2` in cycle 21, a latency of 14 cycles.

Requester asserts request `f3` in cycle 21, and the CXU responds in cycle 22.

3.8. CXU-L3 reordering CXU signaling

A reordering CXU computes a function of the CXU request and the specified state context, if any, updating the context, and sending a CXU response in a later clock cycle. **Responses for requests with the same context are sent in order, otherwise may be sent out of order**. There is request and response flow control.

CXU-L3 incorporates a request-response ID for the requester to correlate responses received to requests sent.

This CXU-LI feature level is motivated by past experience building floating point CXUs. Different functions, e.g., comparison, conversion, multiplication, addition, division, and square root, exhibit a wide range of latencies. Some functions, e.g. addition and multiplication, may be pipelined and afford an initiation interval II=1/cycle, while others, e.g. division and square root, may be variable latency and perform one request at a time.

Particularly when a composable extension is stateless and when the requester (e.g., an in-order-issue/out-of-order completion CPU) tolerates out of order responses, response reordering can improve performance and simplify CXU logic by reducing average CXU latency, enabling greater CXU parallelism, and reducing request blocking and response queueing.

When a composable extension is stateful, response reordering cannot occur for any sequence of requests with the same state context, to ensure identical response data and program behavior over time and over different CXU implementations of the same composable extension.

3.8.1. CXU-L3 configuration parameters

Table 14. CXU-L3 configuration parameters

Parameter	Description
CXU_LI_VERSION	CXU-LI version number
CXU_N_CXUS	number of CXUs at/below this CXU
CXU_N_STATES	number of composable extension state contexts

For CXU_LI_VERSION, CXU_N_CXUS, and CXU_N_STATES, see §3.4.1.

3.8.2. CXU-L3 signals

Table 15. CXU-L3 signals

Dir	Port	Width Parameter	Description
in	clk		clock
in	rst		reset
in	clk_en		clock enable
in	req_valid		request valid
out	req_ready		request ready
in	req_id	CXU_REQ_ID_W	request REQ_ID
in	req_cxu	CXU_CXU_ID_W	request CXU_ID
in	req_state	CXU_STATE_ID_W	request STATE_ID
in	req_func	CXU_FUNC_ID_W	request FUNC_ID
in	req_insn	CXU_INSN_W	request raw instruction
in	req_data0	CXU_DATA_W	request operand data 0
in	req_data1	CXU_DATA_W	request operand data 1
out	resp_valid		response valid
in	resp_ready		response ready
out	resp_id	CXU_REQ_ID_W	response ID
out	resp_status	CXU_STATUS_W	response status
out	resp_data	CXU_DATA_W	response data

3.8.3. CXU-L3 signaling protocol

CXU-L3 is synchronous to posedge `clk`. See §3.4.2. CXU-L3 includes a request-response ID. See §3.4.7. CXU-L3 includes the request's raw instruction. See §3.4.6.

Protocol:

1. Request transfer.
   a. Requester asserts CXU request signals `req_*` (including new CXU-L3 signal `req_id`) and asserts `req_valid`.
   b. Responder may assert `req_ready`.
   c. CXU receives CXU request on posedge `clk` when `req_valid` and `req_ready` are both asserted, per §3.4.3
2. Custom function execution.
   a. CXU performs response status / error checking per §3.4.4.
   b. CXU performs a function of the operands and the selected state context.
   c. CXU may update the selected state context, logically prior to any updates *to the same state context* from subsequent requests.
3. Response transfer.
   a. Prior to issuing responses from subsequent requests *to the same state context* (i.e., in order of requests to the same state context) CXU asserts `resp_id`, `resp_status`, `resp_data` and asserts `resp_valid`.
   b. Requester may assert `resp_ready`.
   c. Requester receives CXU response on posedge `clk` when `resp_valid` and `resp_ready` are both asserted, per §3.4.3.

3.8.4. CXU-L3 example

Figure 33 is an example waveform for four CXU-L3 CXU requests, illustrating two different valid out-of-order response sequences, arising from executing four CF instructions `f0`-`f3`. (Assume `CXU_INSN_W=0`, no `req_insn`.) Each instruction issues a CXU request to the same CXU `u0`, but with various state contexts `s0`, `s1`, `s0` (again), and `s3`. This constrains the CXU to respond to request `f0` with state `s0`, before responding to subsequent request `f2` for state `s0`.

Note that each CXU request is tagged with a `req_id`, a value that is returned by the CXU with the corresponding `resp_id`, and used by the requester to correlate responses to requests and recover the reordering as necessary.

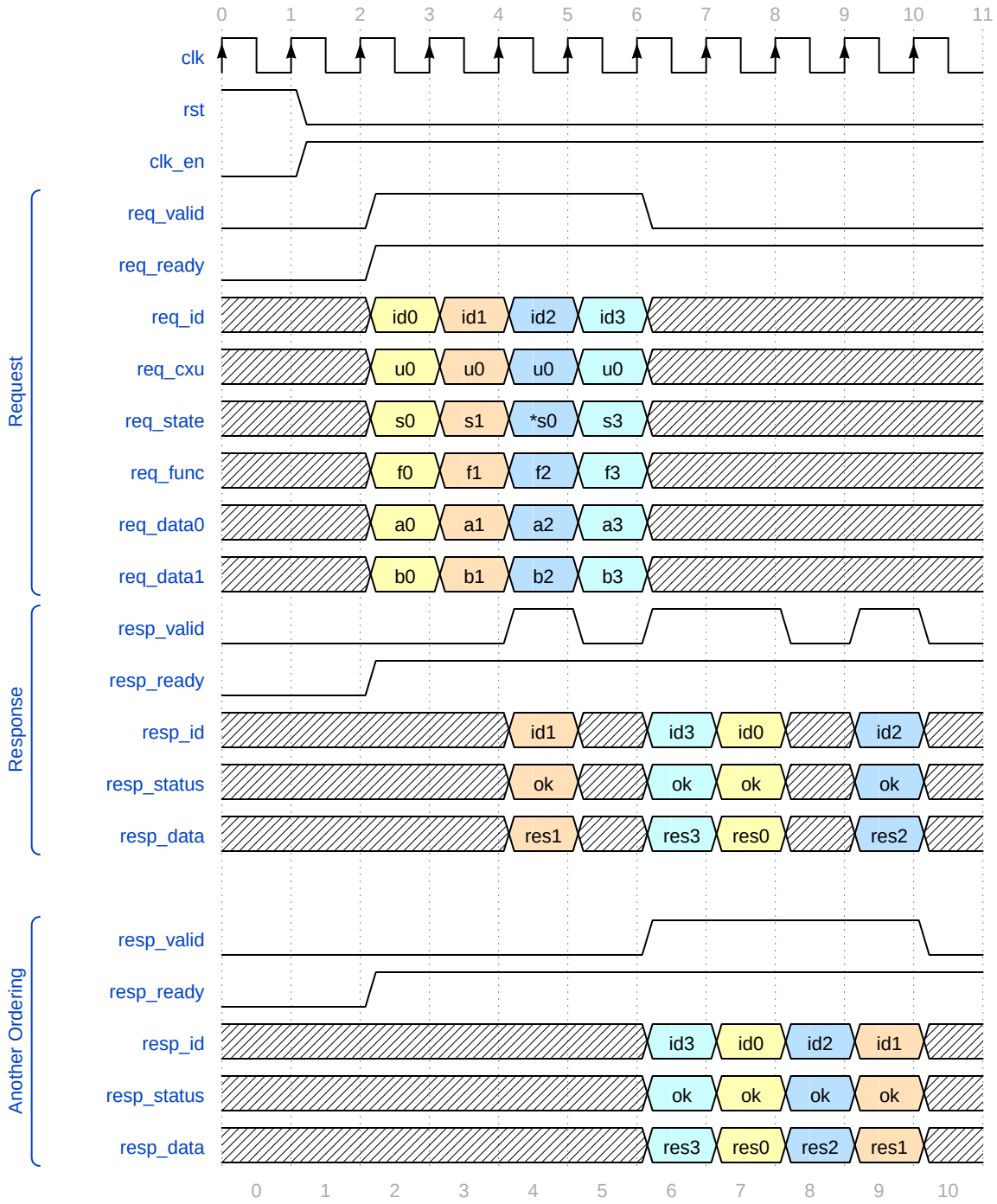


Figure 33. Example CXU-L3 signaling protocol waveform, with two of the possible response orderings

In the first example response, with signals labeled *Response*, the CXU receives requests (`f0`, `f1`, `f2`, `f3`) but responds in order (`f1`, `f3`, `f0`, `f2`). In the second example response, with signals labeled *Another Ordering*, the CXU responds in order (`f3`, `f0`, `f2`, `f1`). Bother orderings are valid because they preserve the order `f0`<`f2` caused by these two CXU requests using the same state `s0`.

3.9. CXU feature level adapters

A CXU feature level adapter is an intermediary CXU that receives requests and sends responses at one CXU-LI feature level and adapts them for and forwards them to a subordinate CXU at a lower CXU-LI feature level.

CXU-LI includes a set of configurable adapters to raise any CXU to any higher feature level, easing composition:

- `Cvt01`: raise L0 to L1: add configurable latency pipelining
- `Cvt02`, `Cvt12`: raise L0 or L1 to L2: add request-response flow control (may suspend requests)

TODO: Describe the L3 adapters, which are just L2 adapters with a request-response ID FIFO.

3.9.1. `Cvt01`: raise CXU-L0 to CXU-L1

A `Cvt01` adapter CXU implements CXU-L1, including its configuration parameters (§3.6.1), adapting L1 requests to and responses from a subordinate combinational L0 CXU.

When `CXU_LATENCY=0`, the adapter's request/response channels are directly coupled to the subordinate CXU request/response channels. Otherwise, these channels I/Os are registered and pipelined, with a total latency of `CXU_LATENCY` cycles.

Automatic pipeline retiming may slice the combinational logic cone into several pipeline stages, achieving higher frequency operation.

3.9.2. `Cvt02`: raise CXU-L0 to CXU-L2

A `Cvt02` adapter CXU implements CXU-L2, including its configuration parameters (§3.7.1), adapting L2 requests to and responses from a subordinate combinational L0 CXU. The adapter has a fixed latency of one cycle — a response is sent one cycle after a request is received.

To avoid arbitrary CXU response queuing, yet keep signaling simple and frugal, the `Cvt02` adapter might negate `req_ready` on any cycle that it has a valid response waiting (asserting `resp_valid`) and the requester negates `resp_ready`.

3.9.3. `Cvt12`: raise CXU-L1 to CXU-L2

A `Cvt12` adapter CXU implements CXU-L2, including its configuration parameters (§3.7.1), plus `CXU_LATENCY` (§3.6.1), adapting L2 requests to and responses from a subordinate fixed latency L1 CXU.

The `CXU_LATENCY` parameter, which specifies the latency of the *subordinate L1 CXU*, typically configures the depth of a response FIFO — an entire response stream must be buffered when the requester, having just issued `CXU_LATENCY` of requests to the L1 CXU, negates `resp_ready` through as many clock cycles. Eventually, with response transfers paused, the response FIFO fills and the adapter CXU negates `req_ready`.

When `CXU_LATENCY=0`, the subordinate CXU response must be registered and therefore the adapter's response latency is at least one cycle.

3.10. CXU-LI-compliant CPUs

A CXU-LI-compliant CPU implements RISC-V RV-I -Zicsr + *(CX-ISA extension)* instruction set, sends CXU requests upon issuing custom operation instructions, and writes a destination register and CX status CSR in response to CXU responses.

3.10.1. CPUs and CXU-LI feature levels

CPUs, as CXU requesters, use specific CXU-LI feature levels.

An austere single-cycle CPU might use CXU-L0 with a combinational CXU (only).

A pipelined in-order CPU might use CXU-L1 with a fixed latency CXU configured for (e.g.) 2 cycles latency. It might also use CXU-L2 with a variable latency CXU, stalling the pipeline during cycles where CF instructions cannot issue because the selected CXU negates `req_ready`*, and itself negating* `resp_ready` *during write-back cycles when the register file's write port or other necessary resource is unavailable.*

An out-of-order completion CPU, i.e. one that may commit low latency instructions before prior high latency instructions, might issue CF instructions to a CXU-L2 variable latency CXU and in some future cycle retire the variable latency CXU response, here again negating `resp_ready` *when it is unable to accept a response to writeback.*

An OoO completion CPU, that handles reordered CXU responses, might use a CXU-L3 reordering CXU.

A CPU may have one or more sets of CXU request and response ports. For each such set, a CPU may send zero or one CXU request per cycle and receive zero or one CXU response per cycle.

Most CPUs send up to one request and receive up to one response. However, a CXU-LI compliant superscalar CPU might send multiple CXU requests and receive multiple CXU responses, to multiple CXUs of the same, or different, CXU-LI feature levels, in parallel, in the same cycle.

3.11. Example: CXU signaling in a composed system

Consider Figure 34, a system composed from two single-hart CPUs, two stateful CXUs, and a 2-input, 2-output Switch CXU. Fixed latency CXU_0 implements CXU-L1, configured with `CXU_LATENCY=1`. The CPUs, CXU_1, and `Switch22` use/implement CXU-L2. `Cvt12`, a CXU level converter, up-converts CXU_0 from CXU-L1 to CXU-L2.

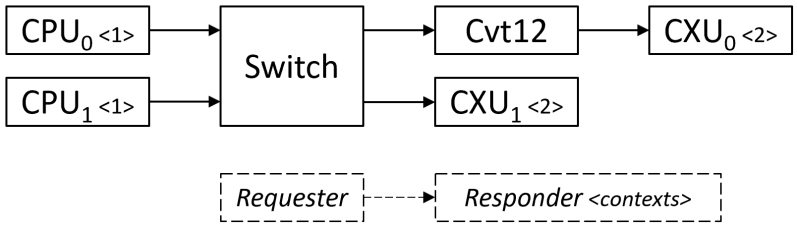


Figure 34. CXU-L2 system, with two CPUs, switch CXU, converter CXU, CXU_0 (L1), and CXU_1 (L2)

With one hart per CPU, the composable extensions' CXUs are configured with two state contexts each (<2>).

Both CPU_0 and CPU_1 are configured to issue CF instructions mapping $CX\_ID_0 \rightarrow CXU\_ID{=}0 \rightarrow CXU_0$ and $CX\_ID_1 \rightarrow CXU\_ID{=}1 \rightarrow CXU_1$.

The exemplary 2x2 Switch CXU is frugal, if low frequency, while sustaining one cycle initiation interval transfers of requests and responses. It multiplexes downstream request transfers and upstream response transfers. In both directions, the switch consists of input ports (not registered), output port registers, an approximately fair output port arbiter, and a 2x2 channel crossbar. Each cycle, the switch determines which output ports are *available* (i.e., are empty, or will transfer (`valid & ready`) this cycle) and which valid inputs are *eligible* to transfer, then asserts ready, and transfers, some eligible inputs to available output ports, based upon a rotating priority order.

A *request* input port is eligible to transfer if it is valid and if the target `req_cxu` CXU_ID is the same as the last request, or if there are no pending responses for this port. This ensures that responses for requests, routed to different CXUs with different latencies, are always returned in order to the requester, as required by CXU-L2.

Downstream request routing is per the request inputs' `req_cxu` elements: CXU_ID=0 routes to the first output port and CXU_ID=1 routes to the second output port. The switch itself responds to requests with invalid CXU_IDs with a `CXU_ERROR_CXU` response.

For upstream response routing, the Switch incorporates, for each subordinate CXU, a FIFO queue that records the requester port ID that issued each request to that CXU. As each (in order) response from that CXU is received, the requester port ID is dequeued from that FIFO and used to route the response to its corresponding requester.

In this example, assume each CPU decouples issue and commit using a scoreboarded register file enabling arbitrary extension unit latencies. Each CPU runs the same code (Listing 1):

1. Write `mcx_selector` for CXU_ID=0 and STATE_ID=*HART_ID*, issue two CF instructions to CXU_0;
2. Write `mcx_selector` for CXU_ID=1 and STATE_ID=*HART_ID*, issue two CF instructions to CXU_1;
3. Write `mcx_selector` for CXU_ID=0 and STATE_ID=*HART_ID*, issue one CF instruction to CXU_0.

Listing 1. Issue stateful CF instructions `f0` and `f1` to CXU_0, `f2` and `f3` to CXU_1, and `f4` to CXU_0 again.

```
csrw mcx_selector,x20   ; version=1, cxe=0, CXU_ID=0, STATE_ID=HART_ID
cx_reg 0,x3,x1,x2       ; u0.f0
cx_reg 1,x6,x5,x4       ; u0.f1

csrw mcx_selector,x21   ; version=1, cxe=0, CXU_ID=1, STATE_ID=HART_ID
cx_reg 2,x9,x7,x8       ; u1.f2
cx_reg 3,x12,x11,x10    ; u1.f3

csrw mcx_selector,x20   ; version=1, cxe=0, CXU_ID=0, STATE_ID=HART_ID again
cx_reg 4,x15,x13,x14    ; u0.f4
```

Figure 35 is an example waveform executing Listing 1 near-simultaneously on the two CPUs of Figure 34.

(*1:u2<3>.f4* denotes CXU request #1 with CXU_ID=2 STATE_ID=3 FUNC_ID=4)

In the narrative that follows, that *A sends B* means *A asserts B ahead of next posedge `clk`*, whereas *B transfers to C* means *during this cycle C receives and accepts it*. Recall with CXU-L2, request transfers occur when both `req_valid` and `req_ready` are asserted (§3.4.3), and response transfers occur when `resp_valid` and `resp_ready` are asserted.

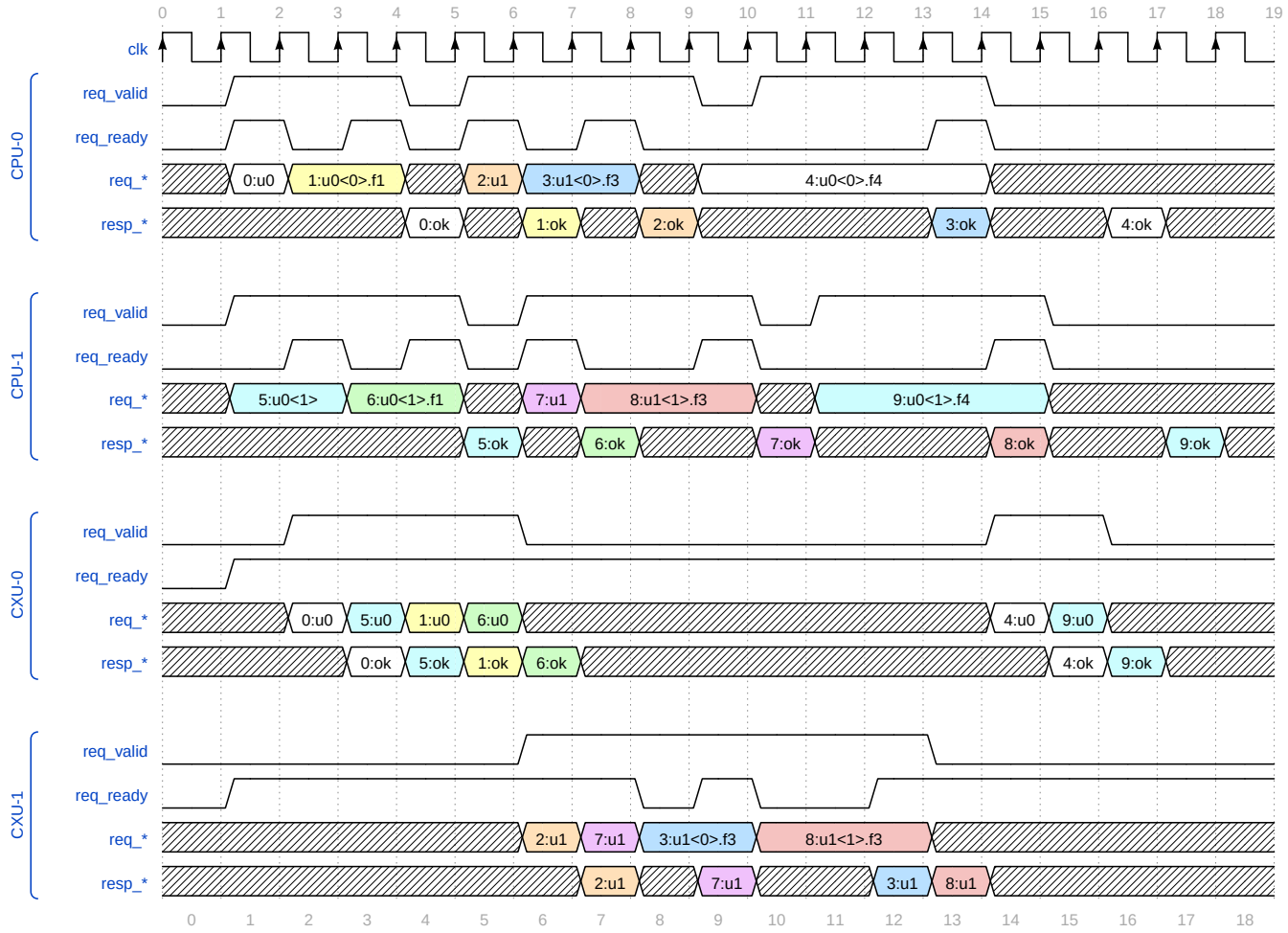


Figure 35. Example 2-input 2-output CXU-L2 Switch CXU signaling protocol waveform

Cycle-by-cycle:

0. Both CPUs CSR-write their hart's `mcx_selector` registers, selecting CXU_ID=0=CXU_0, and their hart's STATE_ID.
   Both CPUs issue the first CF instruction (`f0`).

0. CPU_0 sends first CXU request (request #0): CXU_ID=0 STATE_ID=0 FUNC_ID=0, a.k.a. `0:u0<0>.f0`.
   CPU_1 sends first CXU request (request #5): CXU_ID=0 STATE_ID=1 FUNC_ID=0, a.k.a. `5:u0<1>.f0`.

1. CPU_0's first request, destined for CXU_0, wins arbitration for Switch output port 0.
   Switch asserts CPU_0's `req_ready` and negates CPU_1's `req_ready`.
   CPU_0's first request `0:u0<0>.f0` transfers to Switch.
   Switch sends CPU_0's first request to `Cvt12(`CXU_0`)`
   CPU_0 sends second CXU request: `1:u0<0>.f1`.

2. CPU_1's first request, destined for CXU_0, wins arbitration for Switch output port 0.
   Switch asserts CPU_1's `req_ready` and negates CPU_0's `req_ready`.
   CPU_1's first request `5:u0<1>.f0` transfers to Switch.
   Switch sends CPU_1's first request to `Cvt12(`CXU_0`)`.
   CPU_1 sends second CXU request: `6:u0<0>.f1`.
   CPU_0's first request `0:u0<0>.f0` transfers to CXU_0.
   CXU_0 executes `0:f0`, updates state `<0>`, sends response to Switch.

3. CPU_0 sends no CXU request this cycle, due to its second `csrw` execution cycle.

CPU$_0$'s second request `1:u0<0>.f1`, wins arbitration, transfers to Switch, is sent to `Cvt12(CXU0)`.
CPU$_1$'s first request `5:u0<1>.f0` transfers to CXU$_0$, executes, updates `<1>`, sends response to Switch.
CXU$_0$'s response to CPU$_0$'s first request transfers to Switch, is sent to CPU$_0$.

4. CPU$_1$ sends no CXU request this cycle, due to its second `csrw` execution cycle.
   CPU$_1$'s second request `6:u0<0>.f1`, wins arbitration, transfers to Switch, is sent to `Cvt12(CXU0)`.
   CPU$_0$'s second request `1:u0<1>.f1` transfers to CXU$_0$, executes, updates `<0>`, sends response to Switch.
   CXU$_0$'s response to CPU$_1$'s first request transfers to Switch, is sent to CPU$_1$.
   CXU$_0$'s response to CPU$_0$'s first request transfers to CPU$_0$.
5. CPU$_0$ bubble in CXU request issue due to its second `csrw` execution cycle.
   CPU$_1$ sends third request `2:u1<1>.f2`, with CXU_ID=1, destined for CXU$_1$.
   CPU$_0$'s third request `2:u1<0>.f2`, transfers to Switch, is sent to CXU$_1$.
   CPU$_0$ sends fourth request `3:u1<0>.f3`, with CXU_ID=1, destined for CXU$_1$.
   CPU$_1$'s second request `6:u0<1>.f1` transfers to CXU$_0$, executes, updates `<1>`, sends response to Switch.
   CXU$_0$'s response to CPU$_0$'s second request transfers to Switch, is sent to CPU$_0$.
   CXU$_0$'s response to CPU$_1$'s first request transfers to CPU$_1$.
6. CPU$_1$'s third request `7:u1<0>.f2` wins arbitration, transfers to Switch, is sent to CXU$_1$.
   CPU$_1$ sends fourth request `8:u1<0>.f3`, with CXU_ID=1, destined for CXU$_1$.
   CPU$_0$'s third request `2:u1<0>.f2` transfers to CXU$_1$, executes, updates `<0>`, sends response to Switch.
   CXU$_0$'s response to CPU$_1$'s second request transfers to Switch, is sent to CPU$_1$.
   CXU$_0$'s response to CPU$_0$'s second request transfers to CPU$_0$.
7. CPU$_0$ sends no CXU request this cycle, due to its third `csrw` execution cycle.
   CPU$_0$'s fourth request `3:u1<0>.f3` wins arbitration, transfers to Switch, is sent to CXU$_1$.
   CPU$_1$'s third request `7:u1<1>.f2` transfers to CXU$_1$, begins execution.
   CXU$_1$'s response to CPU$_0$'s third request transfers to Switch, is sent to CPU$_0$.
   CXU$_0$'s response to CPU$_1$'s second request transfers to CPU$_1$.
8. CPU$_1$ sends no CXU request this cycle, due to its third `csrw` execution cycle.
   CPU$_0$ sends fifth request `4:u0<0>.f4`, with CXU_ID=0, destined for CXU$_0$.
   At CXU$_1$, CPU$_1$'s third request `7:u1<0>.f2` completes execution, updates `<1>`, sends response to Switch.
   CXU$_1$'s response to CPU$_0$'s third request transfers to CPU$_0$.
9. CPU$_0$'s fifth CXU request is *ineligible* to transfer because CPU$_0$ has pending requests to CXU$_1$. It becomes eligible at cycle 13.
   CPU$_1$'s fourth request `8:u1<0>.f3` transfers to Switch, is sent to CXU$_1$.
   CPU$_0$'s fourth request `3:u1<0>.f3` transfers to CXU$_1$, begins execution.
   CXU$_1$'s response to CPU$_1$'s third request transfers to Switch, is sent to CPU$_1$.
10. CPU$_1$ sends fifth request `9:u0<1>.f4`, with CXU_ID=0, destined for CXU$_0$.
    CPU$_0$'s fourth CXU request `3:u1<0>.f3` continues execution.
    CXU$_1$'s response to CPU$_1$'s third request transfers CPU$_1$.
11. CPU$_1$'s fifth CXU request is *ineligible* to transfer because CPU$_1$ has pending requests to CXU$_1$. It becomes eligible at cycle 14.
    CPU$_0$'s fourth CXU request `3:u1<0>.f3` completes execution, updates `<0>`, sends response to Switch.
12. CPU$_1$'s fourth request `8:u1<1>.f3` transfers to CXU$_1$, executes, updates `<1>`, sends response to Switch.
    CXU$_1$'s response to CPU$_0$'s fourth request transfers to Switch, is sent to CPU$_0$.
13. CXU$_1$'s response to CPU$_0$'s fourth request transfers to CPU$_0$.
    CPU$_0$'s fifth request `4:u0<0>.f4` becomes eligible, transfers to Switch, is sent to CXU$_0$.
14. CXU$_1$'s response to CPU$_1$'s fourth request transfers to CPU$_1$.
    CPU$_1$'s fifth request `9:u0<1>.f4` becomes eligible, transfers to Switch, is sent to CXU$_1$.
    CPU$_0$'s fifth request `4:u0<0>.f4` transfers to CXU$_0$, executes, updates `<0>`, sends response to Switch.

15. CPU_1's fifth request `9:u0<1>.f4` transfers to CXU_0, executes, updates `<1>`, sends response to Switch. CXU_0's response to CPU_0's fifth request transfers to Switch, is sent to CPU_0.
16. CXU_0's response to CPU_1's fifth request transfers to Switch, is sent to CPU_1. CXU_0's response to CPU_0's fifth request transfers to CPU_0.
17. CXU_0's response to CPU_1's fifth request transfers to CPU_1.

3.12. Composing CXUs with AXI4-Streams

In some configured systems, preexisting infrastructure components that implement AXI4-Stream protocol may be used to help compose CPUs and CXUs. A fully flow controlled CXU-LI -L2 or -L3 transfer may be transported over two AXI4-Stream (AXI-S) streams, one for requests and one for responses.

For example, in a AMD/Xilinx Versal FPGA, a CPU might transfer CXU requests, via CXU-L2-to-AXI-S bridge, AXI-S-to-NOC bridge, Versal NOC, NOC-to-AXI-S bridge, AXI-S-to-CXU-L2 bridge, to a CXU at the far corner of the FPGA fabric, later transferring CXU responses back to the distant CPU by the same means.

Table 16 presents a recommended canonical mapping between CXU-LI signals and the two AXI-S streams.

Table 16. Recommended mapping between CXU-L2/-L3 and request/response AXI4-Streams

Dir	CXU-LI Port	Width	AXI-S Port
in	`clk`		`aclk`
in	`rst`		`aresetn` *(inverted)*
in	`clk_en`		-
in	`req_valid`		`reqs_tvalid`
out	`req_ready`		`reqs_tready`
in	`req_id`	`CXU_REQ_ID_W`	`reqs_tid` *or* `reqs_tdest`
in	`req_cxu`	`CXU_CXU_ID_W`	`reqs_tuser` *or* `reqs_tdest`
in	`req_state`	`CXU_STATE_ID_W`	`reqs_tuser`
in	`req_func`	`CXU_FUNC_ID_W`	`reqs_tuser`
in	`req_insn`	`CXU_INSN_W`	`reqs_tuser`
in	`req_data0`	`CXU_DATA_W`	`reqs_tdata`
in	`req_data1`	`CXU_DATA_W`	`reqs_tdata`
in	-		`reqs_tlast` *optional*
in	-	*	`reqs_tstrb` *optional*
in	-	*	`reqs_tkeep` *optional*
out	`resp_valid`		`resps_tvalid`
in	`resp_ready`		`resps_tready`
out	`resp_id`	`CXU_REQ_ID_W`	`resps_tid` *or* `resps_tdest`
out	`resp_status`	`CXU_STATUS_W`	`resps_tuser`
out	`resp_data`	`CXU_DATA_W`	`resps_tdata`
out	-		`resps_tlast` *optional*
out	-	*	`resps_tstrb` *optional*
out	-	*	`resps_tkeep` *optional*

When several CXU-LI signals map to a single AXI-S port, the signals are to be concatenated in order, each signal

assigned successively more significant bits. For example, using Verilog concatenation:

```
reqs_tuser = { req_insn,req_func,req_state,req_cxu };
reqs_tdata = { req_data1,req_data0 };
```

Use `reqs_tdest` when `req_id` and/or `req_cxu` indicate/encode a specific AXI-S destination (of a bridge to a CXU). Use `resps_tdest` when of `resp_id` indicates a specific AXI-S destination (of a bridge to a requester, e.g., CPU).

4. CX-ABI: CX Application Binary Interface

4.1. Basic ABI

Each thread has a current CX selection that determines how custom instructions and custom CSR accesses ("custom operations") are performed. The current selection may be:

1. Legacy mode: CX multiplexing is *disabled*: custom operations issue pre-existing built-in custom operations.
2. CX mode: CX multiplexing is *enabled*: custom operations issue to the selected CX and CX state context.

The CX-ABI defines these rules, which must be implemented explicitly in code or automatically by CX-ABI aware compilers:

1. **[ABI-INIT]**: Initially, the selection is legacy mode.
2. **[ABI-ENTRY]**: On entry to a function, or following a function call, the selection is legacy mode.
3. **[ABI-SELECT-CX]**: Code **must** select a CX prior to issuing that CX's custom operations.
4. **[ABI-DESELECT-CX]**: Code that selects a CX **must** select legacy mode prior to calling a function, returning, or stack unwinding.
5. **[ABI-SELECT-LEGACY]**: Code **should** select legacy mode prior to issuing built-in custom operations.

4.2. ABI with CX functions—provisional

This provisional ABI section is non-normative.

At present there is a proposal to designate certain functions as *CX functions* which **must** be called in CX mode, with selected CX and state context.

Therefore for this "CX-ABI with CX functions" there are two types of functions:

1. Ordinary functions. Ordinary functions expect legacy mode.
2. CX functions. CX functions are so-designated by a language-specific attribute, declaration, or type specifier. CX functions require CX mode.

This provisional ABI defines these rules, which must be implemented explicitly in code or automatically by CX-ABI aware compilers:

1. **[ABI-INIT]**: Initially, the selection is legacy mode.
2. **[ABI-ENTRY]**: On entry to an *ordinary* function, or following *any* function call, the selection is legacy mode.
3. **[ABI-CX-ENTRY]**: On entry to a *CX* function, the selection is CX mode.
4. **[ABI-SELECT-CX]**: Code **must** select a CX prior to issuing that CX's custom operations *or calling a CX function*.
5. **[ABI-DESELECT-CX]**: Code that selects a CX **must** select legacy mode prior to calling an *ordinary* function, returning, or stack unwinding.
6. **[ABI-SELECT-LEGACY]**: Code **should** select legacy mode prior to issuing built-in custom operations.

4.3. Rationale

This Rationale section is non-normative.

The CX ABI ensures correct composition of independently authored CX libraries and legacy custom extension libraries under CX multiplexing.

CX multiplexing operates by setting the hart's CX selector (`mcx_selector` write), or setting it indirectly (`cx_index` write), to select a specific CX and CX state context to issue custom instructions or accessing custom CSRs (together, *custom operations*).

Per §2.2.1, `mcx_selector.version` determines whether CX multiplexing is enabled or disabled:

- "When `version=0`, disable composable extension multiplexing. … `Custom-[0123]` instructions execute the CPU's built-in custom instructions."
- "When `version=1`, enable … composable extension multiplexing. … `Custom-[012]` instructions issue CXU requests to the CXU identified by `cxu_id` and to the state context identified by `state_id`."

So software must write the CX selector prior to issuing CX custom operations, and software must also ensure the CX selector is disabled prior to issuing legacy custom operations.

Since application software is composed of dozens or hundreds of separately authored, sometimes separately versioned libraries, we require an application binary interface (ABI) that ensures dependable disciplined use of the thread's shared CX selector, so that whenever software performs custom operations, these are performed against exactly the expected CX or legacy custom.

4.3.1. Callee-save won't do

An older version of the spec defined a provisional CX ABI that managed the CX selector with a *callee-save* discipline:

1. Initial selection is legacy mode.
2. Any code that writes CX selector must save the prior value and restore it upon return / stack unwind.
3. Code which does not change selectors need not save or restore selectors.

This callee-save discipline has the advantages that:

- It provides correct nested composition of CX libraries.
  - If CX A lib selects CX A, performs A custom operations then calls lib B, which selects CX B and performs B custom operations, B must re-select the previous selection (CX A) prior to returning to lib A, which can then happily perform more A custom operations.
- Compared to *caller-save*, wherein a CX library defensively re-selects its CX after every function call out, the CX library trusts the transitive callees to restore its CX selection, minimizing the number of CX selector writes in a given code path.

What's wrong with callee-save?

It breaks legacy custom code. Under callee-save you must always select your CX (or legacy mode) prior to issuing custom operations. However preexisting legacy custom code, and legacy compilers, both predating CX, do not include the required CX selector writes to select legacy mode, nor a matching CX selector write to restore the caller's selection. So any explicit or implicit use of a legacy custom operation may instead forward to some other selected CX — a disaster! In summary, callee-save discipline is incompatible in general with legacy libraries and compilers

that use legacy custom operations at will. Since legacy custom code may appear almost anywhere, it follows the default, ambient CX selection should be legacy mode.

Wrong trust model. Callee-save CX selection means you must trust code you do not control to preserve your current CX selection. In general any C function capable ABI assumes that a callee will not corrupt the stack or do other undefined behavior that corrupts the caller, but CX multiplexing is a sharp knife, and if the callee violates the callee-save ABI and returns with a different CX selector in place, the caller may issue custom operations to the wrong CX or CX state context. In applications comprising separately authored, separately versioned libraries, it is not possible to inspect the transitive call graph from a CX library to ensure callers preserve the CX selection as required. A more secure, more defensive, more amenable to program analysis ABI merits a greater degree of paranoia in each CX library, by assuming that callees do not preserve the current CX selection.

4.3.2. Ambient legacy mode ABI

These two problems with callee-save discipline led to a redesign of the ABI. The revised design tenets are:

1. Support composition of CX libraries, including nested composition of CX libraries, alongside legacy custom extension libraries.
2. Support preexisting legacy custom extension libraries, even when they don't explicitly manage (disable) CX muxing.
3. Minimize the CX selection "trust surface" to that of the current function (or perhaps, current library).
4. Minimize the number of CX selector writes.

The present ABI endeavors to maintain an ambient legacy mode selection when not actively issuing CX custom operations. This ensures, to the greatest extent possible, that legacy custom code, unaware of CX multiplexing, and lacking the code to select legacy custom mode, nevertheless always operates in legacy custom mode.

For CX libraries, this code supports composition and nested composition. Composition works because each library selects its CX prior to issuing its custom operations. Nested composition also works, because, after following a function call (**[ABI-ENTRY]**) the caller must re-select its CX (**[ABI-SELECT-CX]**) prior to issuing additional custom operations:

```
CX A lib sets CX selection to CX A, issues A operations
CX A lib sets CX selection to legacy mode, calls CX B lib
CX B lib sets CX selection to CX B, issues B operations
CX B lib sets CX selection to legacy mode, returns
CX A lib sets CX select to CX A, issues more A operations
CX A lib sets CX selection to legacy mode, returns.
```

Also, all is well when a CX A lib calls legacy custom code:

```
CX A lib sets CX selection to CX A, issues A operations
CX A lib sets CX selection to legacy mode, calls legacy lib
legacy lib issues its legacy custom operations
legacy lib returns
CX A lib sets CX select to CX A, issues more A operations
CX A lib sets CX selection to legacy mode, returns.
```

Rule **[ABI-DESELECT-CX]** helps ensure that following a brief excursion into a CX lib which changes the CX selection, we immediately return to legacy mode in case we encounter selection-less legacy custom code.

There is still an attack surface caused by malicious code violating the ABI by selecting some CX, then calling (or returning to) selection-less legacy custom code, which issues custom operations which are not the legacy custom operations it intends causing unboundedly undefined behavior.

Rule **[ABI-SELECT-LEGACY]** helps defend against this. To the extent practical or necessary, legacy custom code should be compiled defensively to set legacy mode on entry and after function calls prior to issuing its custom operations.

Unlike the deprecated callee-save ABI, these rules will incur additional unnecessary CX selection writes and will give up a little bit of performance (which after all may be the reason for using that CX in the first place.) For example, in the CX A lib + CX B lib nested example above, several CX selector writes are unnecessary. It is possible for a CX enlightened compiler+linker to analyze control flow within a monolithic CX library and optimize the generated code by eliding provably unnecessary defensive CX selector writes.

5. CXU Metadata (CXU-MD)

To help automate system composition, each composable hardware core (each CPU and CXU) shall include a metadata file which defines the properties, features, and supported values of its configuration parameters.

For each core, for each configuration parameter, metadata may specify a subset of the set of legal configuration parameter values defined in §3.4.1.

Metadata configuration parameter values are encoded as either a single value, a list of values, or a range of values. For a continuous range of integer values, the parameter value is range, and the inclusive range of values is found in a corresponding parameter whose name ends in `_range`. For example,

```
parameter1: 0           # single value (scalar)
parameter2: [32, 64]    # list of allowed values (sequence)
parameter3: range       # range, via parameter3_range
parameter3_range: [5,9] # inclusive range of integer values. Expands to [5,6,7,8,9]
```

5.1. CXU Metadata

Listing 2 specifies the CXU metadata format, in YAML. Each legal configuration parameter range of §3.4.1 `CXU_PARAM` may be overridden (subsetted) through a YAML parameter line `param:` .

The CXU metadata may also be used to specify `other` custom (non-standard / CXU specific) configuration parameter settings.

Listing 2. CXU metadata format

```
cxu_name: string
cxu_li:
    feature_level: scalar                   # required.  allowed: 0-3
    state_id_max: scalar | list | 'range'   # level:any. default: any. 0 => stateless
    req_id_w: scalar | list | 'range'       # level:2+.  default: 0
    cxu_id_w: scalar | list | 'range'       # level:any. default: 0
    state_id_w: scalar | list | 'range'     # level:1+.  default: 0
    insn_w: scalar | list | 'range'         # level:1+.  default: 0
    func_id_w: scalar | list | 'range'      # level:any. default: 10
    data_w: scalar | list                   # level:any. default: 32
    latency: scalar | list | 'range'        # level:1.   default: 1
    reset_latency: scalar | list | 'range'  # level:1.   default: 0
    xyz_range: [min,max]                    # when parameter xyz is range
```

Need some stronger naming of CXUs and CPUs here. Perhaps a GUID, perhaps a URL.

Do we need to specify here which CX_IDs the CXU implements?

5.2. Example CXU metadata

Listing 3 is example CXU metadata for a CXU-L1 CXU which supports only one state context, requires at least 5-bit CF_IDs, requires XLEN=32, and supports a response latency of 2-4 cycles.

Listing 3. Example CXU metadata (CXU-L1)

```
cxu_name: bobs_bnn_cxu
cxu_li:
    feature_level: 1
    state_id_max: 1          # only supports 1 state context
    req_id_w:                # any req_id is fine
    cxu_id_w: 0              # no req_cxu
    state_id_w: 0            # no req_state_id
    insn_w: 0                # no req_insn
    func_id_w: range         # need >= 5-bit CF_IDs
    func_id_w_range: [5,10] # so [5,6,7,8,9,10] are OK
    data_w: 64               # XLEN=64-bit only
    latency: [2,3,4]         # configurable w/ 2-4 cycles of latency
    reset_latency: 1         # requires at least 1 cycle of reset latency
other:
    adder_tree: [0,1]        # non-standard config parameter
    element_w: [4,8,16,32]   # non-standard config parameter
```

5.3. CPU Metadata

As described in §3.10, CPUs, as CXU requesters, use specific CXU-LI feature levels. As with CXUs, CPUs use CXU metadata to override configuration parameter defaults, in this case to define what the CPU requires or accepts of its CXU (which is, generally, the root of the DAG of CXUs).

Listing 4. CPU metadata format

```
cpu_name: string
cxu_li: # see [Listing 1].
```

5.4. Example CPU metadata

Listing 5 is example CXU metadata for a CPU that requires and supports only 32-bit combinational CXUs.

Listing 5. Example CPU metadata (requires a CXU-L0 CXU DAG)

```
cpu_name: carols_simple_scalar_cpu
cxu_li:
    feature_level: 0    # L0 combinational CXUs only
    state_id_max:       # L0: n/a
    req_id_w:           # L0: n/a
    cxu_id_w:           # supports arbitrary CXU_IDs
    state_id_w:         # L0: n/a
    insn_w:             # L0: n/a
    func_id_w:          # supports arbitrary CF_IDs
    data_w: 32          # XLEN=32-bit only
```

5.5. System manifest

TODO

Consider CX library metadata too. "I may use this subset { CF_IDs } of the CF_IDs of extension CX_ID."

6. TODO

Todo:

- Chapter on CX Runtime (runtime) API
- How CX and CXU versioning works; how CXU-LI versioning works
- A place for miscellaneous design notes

6.1. Open design problems (post 1.0)

- Developer tooling recommendations for disassembly, debugging, profiling, perf monitoring.

6.2. Cost model

Here write up a brief estimate of the FPGA area overhead of various CX-ISA and CXU-LI mechanisms and behaviors.

7. Specification Change History

7.1. Version 0.95.240403, 2024-04-03: Add CX CSRs

Add CX-scoped custom CSRs:

1. Extend CX multiplexing to multiplex *both* custom opcode instructions and now also custom CSR access instructions.
2. Extend CXU-LI to convey CX CSR access requests and responses.
3. Change `cx_imm` CX custom function instruction format to follow the format of `addi` and `csrrw`, with a 12-bit imm field.
4. Rename `mcx_selector.cte` to `mcx_selector.cxe` (*custom operation exception enable*) — enables illegal-instruction exceptions on use of custom operations; resize `mcx_selector.version` from 4-bits to 3-bits.

7.2. Version 0.94.240327, 2024-03-27: Add mcx_selector.cte.

Add `mcx_selector.cte`, *custom operation trap enable*.

This enables emulation of absent custom instructions, emulation of absent composable extensions, and virtualization of stateful composable extensions state contexts.

7.3. Version 0.93.240310, 2024-03-10: Revise CX ABI.

Add a CX ABI chapter. Update CX ABI section of Introduction.

The old CX ABI was *callee-save* management of the CX mux CSRs, optimizing for minimum CX mux CSR writes.

The new CX ABI is *ambient legacy mode*, maximally paranoid and backwards compatible, keeping CX muxing *off* (e.g., selecting for legacy built-in custom extensions) except when actively issuing CX custom instructions.

7.4. Version 0.92.231111, 2023-11-11: Add extension multiplexing *version*.

Introduce _CX version, improving CX forward compatibility. Replace `mcx_selector.en` with `mcx_selector.version`. Add `cx_status.iv` error field. Replace `cx_status` field names CI/SI/FI with IC/IS/IF.

7.5. Version 0.91.230803, 2023-08-03: Simplify and improve terminology.

Replace term *Custom Interface (CI)* with *Composable Extension (CX)*. Similarly replace *CFU* with *CXU*. And so forth.

From	To
Custom Interface (CI)	Composable Extension (CX)
Custom Function Unit (CFU)	Composable Extension Unit (CXU)
-Zicfu	-Zicx

From	To
`mcfu_*` and `cfu_*` CSRs	`mcx_*` and `cx_*` CSRs

7.6. Version 0.90.220327, 2022-03-27: First complete draft.

References

Microsoft. (2020). *Component Object Model: Interfaces and Interface Implementations*. docs.microsoft.com/en-us/windows/win32/com/interfaces-and-interface-implementations

Waterman, A., & Asanović, K. (2019). *RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, v. 20191213*. github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

Waterman, A., Asanović, K., & Hauser, J. (2021). *RISC-V Instruction Set Manual, Volume II: Privileged ISA, v. 20211203*. github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf