Draft Proposed RISC-V Composable Custom Extensions Specification

Version 0.90.220320, 03/20/2022: Pre-release version

Table of Contents

Preface . . . 1
1. Introduction: a composable custom extension ecosystem . . . 2
1.1. Open, agile, interoperable instruction set innovation . . . 2
1.2. Examples . . . 3
1.3. Scope: reliable composition via strict isolation . . . 4
1.3.1. Stateless and stateful custom interfaces . . . 4
1.4. Standard interfaces and formats . . . 4
1.4.1. CFU Logic Interface (CFU-LI) . . . 5
1.4.2. CFU hardware-software interface . . . 5
1.4.3. Custom interface multiplexing . . . 6
1.4.4. `IStateContext` and serializable stateful custom interfaces . . . 6
1.4.5. CI Application Programming Interface and CI-ABI . . . 6
1.5. System composition . . . 7
1.5.1. Metadata and system manifest . . . 7
1.5.2. Composer . . . 7
1.5.3. Diversity of systems and operating systems . . . 8
1.6. Pushing the envelope . . . 8
1.7. Future directions, TODOs . . . 8
1.8. Acknowledgements . . . 8
2. Custom interfaces: the hardware-software interface . . . 9
2.1. Definitions . . . 9
2.2. New CFU control / status registers . . . 10
2.2.1. `mcfu_selector` CSR 0xBC0: select active CFU and state context . . . 10
2.2.2. `cfu_status` CSR 0x801: CFU status . . . 11
2.2.3. `mcfu_selector_table` CSR 0xBC1: CFU selector table base . . . 12
2.2.4. `cfu_selector_index` CSR 0x800: CFU selector index . . . 12
2.2.5. Implicit CFU CSR fences . . . 13
2.3. Custom function instruction encodings . . . 13
2.3.1. Custom-0 R-type encoding . . . 13
2.3.2. Custom-1 I-type encoding . . . 13
2.3.3. Custom-2 flex-type encoding . . . 14
2.4. Custom function instruction execution via custom interface multiplexing . . . 14
2.4.1. Precise exceptions . . . 15
2.5. `IStateContext`: the standard custom functions . . . 16
2.5.1. Interface state context status word . . . 17
2.5.2. `cfu_read_status` standard custom function instruction . . . 17
2.5.3. `cfu_write_status` standard custom function instruction . . . 18
2.5.4. `cfu_read_state` standard custom function instruction . . . 18
2.5.5. `cfu_write_state` standard custom function instruction . . . 18
2.6. Resource management and context switching . . . 19
2.7. CFU access control . . . 20

3. Custom Function Unit Logic Interface 22
3.1. Definitions 22
3.2. Example configured system 22
3.3. CFU-LI feature levels 23
3.3.1. CFU-L0: combinational CFU 23
3.3.2. CFU-L1: fixed latency CFU 23
3.3.3. CFU-L2: variable latency, request-only flow control CFU (variable latency CFU) 23
3.3.4. CFU-L3: variable latency, request/response flow control CFU (elastic CFU) 24
3.3.5. CFU-L4: reordering CFU 24
3.3.6. Feature levels summary 24
3.4. CFU-LI signaling 25
3.4.1. CFU-LI configuration parameters 25
3.4.2. Clock, reset, clock enable 26
3.4.3. Request and response valid-ready flow control 27
3.4.4. Response status / error checking 27
3.4.5. Raw instruction 28
3.4.6. Request-response ID 29
3.5. CFU-L0 combinational CFU signaling 29
3.5.1. CFU-L0 configuration parameters 29
3.5.2. CFU-L0 signals 29
3.5.3. CFU-L0 signaling protocol 30
3.5.4. CFU-L0 example 30
3.6. CFU-L1 fixed latency CFU signaling 30
3.6.1. CFU-L1 configuration parameters 30
3.6.2. CFU-L1 signals 31
3.6.3. CFU-L1 signaling protocol 31
3.6.4. CFU-L1 example 32
3.7. CFU-L2 variable latency CFU signaling 33
3.7.1. CFU-L2 configuration parameters 33
3.7.2. CFU-L2 signals 33
3.7.3. CFU-L2 signaling protocol 33
3.7.4. CFU-L2 example 34
3.8. CFU-L3 elastic CFU signaling 35
3.8.1. CFU-L3 configuration parameters 35
3.8.2. CFU-L3 signals 36
3.8.3. CFU-L3 signaling protocol 36
3.8.4. CFU-L3 example 37
3.9. CFU-L4 reordering CFU signaling 37
3.9.1. CFU-L4 configuration parameters 38
3.9.2. CFU-L4 signals 38
3.9.3. CFU-L4 signaling protocol 39
3.9.4. CFU-L4 example 39
3.10. CFU feature level adapters 40
3.10.1. `Cvt01`: raise CFU-L0 to CFU-L1 41

3.10.2. `Cvt02`: raise CFU-L0 to CFU-L2 . . . 41
3.10.3. `Cvt12`: raise CFU-L1 to CFU-L2 . . . 41
3.10.4. `Cvt03`: raise CFU-L0 to CFU-L3 . . . 41
3.10.5. `Cvt13`: raise CFU-L1 to CFU-L3 . . . 42
3.10.6. `Cvt23`: raise CFU-L2 to CFU-L3 . . . 42
3.11. CFU-LI-compliant CPUs . . . 42
3.11.1. CPUs and CFU-LI feature levels . . . 42
3.12. Example: CFU signaling in a composed system . . . 43
3.13. Composing CFUs with AXI4-Streams . . . 46
4. CFU Metadata (CFU-MD) . . . 48
4.1. CFU Metadata . . . 48
4.2. Example CFU metadata . . . 49
4.3. CPU Metadata . . . 49
4.4. Example CPU metadata . . . 49
4.5. System manifest . . . 50
5. TODO . . . 51
5.1. Open design problems (post 1.0) . . . 51
5.2. Other CFU-like mechanisms . . . 51
5.3. Example: a stateful extended precision ALU . . . 51
5.4. Cost model . . . 51
References . . . 52

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 custom instruction set extensions, hardware custom function units, and software libraries.

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

At present this is not a work product of a RISC-V International Working Group, Technical Committee, or subcommittee. Rather we share this work in the hope that it may motivate and inform a hypothetical *Composable Custom Extensions* RISC-V Extension Working Group.

(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, Maciej Kurc, Guy Lemieux, Charles Pappon, 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 now some 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 custom extensions may conflict in use of opcodes, or their implementations may require different CPU cores, pipeline structures, logic interfaces, models of computation, means of discovery, or error reporting regimes. Composition is difficult, impairing reuse of hardware and software, and fragmenting the RISC-V ecosystem.

The RISC-V Composable Custom Extensions Specification introduces a set of hardware-hardware and hardware-software interfaces and metadata designed to make it easy to create, compose, reuse, version, program, and deploy systems with multiple custom extensions and their libraries, enabling an open ecosystem, and marketplace, of custom extensions' hardware and software.

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 interfaces defined in this specification allow anyone, whether individual, organization, or consortium, to rapidly define, develop, and use:

- a *custom interface (CI):* a composable custom extension consisting of a set of *custom function (CF) instructions;*
- a *custom function unit (CFU):* a composable hardware core that implements a custom interface;
- an accelerated *custom interface library* that issues custom functions of custom interfaces;
- a processor that can use any CFU;
- 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. Custom interfaces, their CFUs and libraries, may be open or proprietary, of broad or narrow interest. A new processor can use existing CFUs and CI libraries. A new custom interface, CFU, and library can be used by existing CPUs and systems. Many CFUs may implement a given custom interface, and many libraries may use a custom interface.

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 interfaces and implementations evolve across years and decades.

Composition also requires an unlimited number of independently developed custom interfaces to coexist within a fixed ABI and ISA. This is achieved with *custom interface 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 CI library that issues custom instructions, of a cryptography custom interface, that execute on a cryptography custom function unit.

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

The compression library calls the CI Runtime to discover if compression acceleration is available. The runtime consults the CFU map for that CI_ID, finding the compression CFU_ID. Next the library uses the CI Runtime to *select* the compression interface, and its CFU, prior to issuing compression instructions to this CFU. Later the cryptography library uses the same CI Runtime API to discover and select the cryptography interface prior to issuing cryptography instructions to the cryptography CFU.

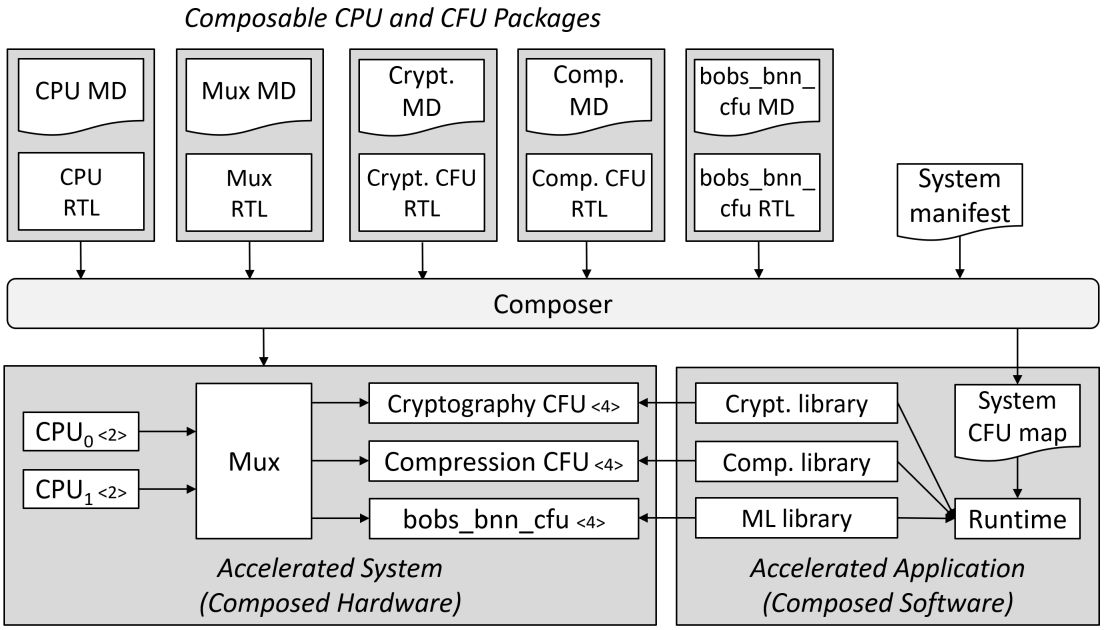

Figure 1. Bob's system, composed from CPU and CFU packages and custom interface libraries

Later, Bob takes Alice's system and adds an ML inference library. For acceleration, Bob defines a new binary neural network custom interface, `bnn_ci`, identified with a new CI_ID he mints. Bob's new BNN custom instructions reuse the standard custom instruction encodings, which is fine because they're scoped to `bnn_ci`. Bob develops `bobs_bnn_cfu` core, and CFU metadata that describes it. He adds that package to the system manifest and rebuilds the system, updating the CFU 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 CFUs.

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

1.3. Scope: reliable composition via strict isolation

To ensure that composition of custom interfaces and their CFUs does not subtly change the behavior of any interface, 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 interface's private state (if any), writing the result to a destination register.

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, 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 default error model accumulates CFU errors in a shared CFU status CSR.)

The isolated state of a custom interface can include private registers and private memories.

1.3.1. Stateless and stateful custom interfaces

A custom interface may be stateless or stateful. For a stateless interface, each CF is a pure function of its operands, whereas a stateful interface 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 interface (only).

Isolated state means that latency notwithstanding, 1) the behavior of the interface only depends upon the series of CF requests issued on that interface and never upon on any other operation of the system; and 2) besides updating interface state, the CFU 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 may update the current state context of the custom interface but has no effect upon another state context of that interface, nor that of any other interface.

A CFU implementing a stateful custom interface 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 CFU in a system may be configured with a different number of its state contexts.

A *serializable* stateful custom interface supports interface-agnostic context management.

Although custom intefaces never introduce nor use CSRs, the same effect can be obtained via custom functions that read or write facets of the interface state context.

1.4. Standard interfaces and formats

To facilitate an open ecosystem of composable custom interfaces, CFUs, libraries, and tools, the specification defines common interop interfaces and formats:

- the *CFU Logic Interface (CFU-LI)*,
- the *Custom Interface Hardware-Software Interface (CI-ABI), including CFU-extensions to RV-I (-Zicfu)*,
- the *Custom Interface Runtime API (CI-RT)*, and
- build-time *CFU Metadata (CFU-MD)*.

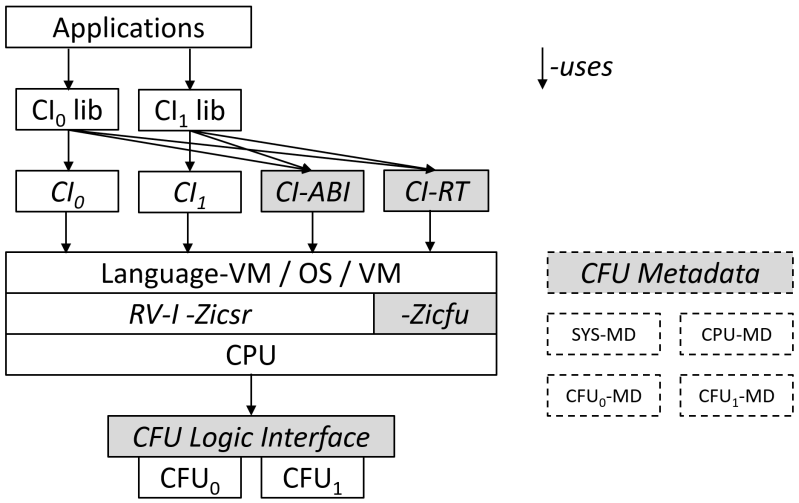

Figure 2. Hardware-software interfaces stack. New standard interfaces and formats are shaded.

The hardware-software interfaces stack (Figure 2) shows how these interfaces and formats work together to compose user-defined custom interfaces CI_0 and CI_1, their libraries, and their CFUs into a system.

1.4.1. CFU Logic Interface (CFU-LI)

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

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

1.4.2. CFU hardware-software interface

The CFU hardware-software interface, -Zicfu, repurposes three custom function instruction formats and adds four CSRs. The three instruction formats reuse the *custom-0*, *custom-1*, and *custom-2* formats / major opcodes (Waterman & Asanović, 2019, p. 143) but (via custom interface multiplexing) compose correctly with any preexisting vendor-defined CPU-specific custom extensions and their custom instructions. The four new CFU CSRs are:

- `mcfu_selector`: selects the hart's current CFU_ID and STATE_ID, for custom interface multiplexing;
- `cfu_status`: accumulates CFU errors;
- `mcfu_selector_table`, `cfu_selector_index`: efficient access control to CFUs and CFU state.

`mcfu_selector_table` *is insufficient given various M/H/S/U privilege levels. This corner of the design requires additional work, and additional CSRs.*

1.4.3. Custom interface multiplexing

Custom interface multiplexing provides an inexhaustible collision-free opcode space for CF instructions for diverse custom interfaces without resort to any *central assigned opcodes authority*, and thereby facilitates direct reuse of CI library binaries.

A custom-interface-aware library, prior to issuing a CF instruction, must first CSR-write a *system and hart specific* CI selector value to `mcfu_selector`, routing subsequently issued CF instructions on this hart to its CFU and to a specific state context. Like the -V vector extension's `vsetvl` instructions, a CSR-write to `mcfu_selector` is a prefix that modifies the behavior of CF instructions that follow. With each CF instruction issued, the CPU sends a CFU request to the hart's current CFU and its current state. This request is routed by standard Mux CFU and adapter cores to the hart's *current* CFU, 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 `cfu_status`.

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

In a system with multiple CI libraries that invoke CF instructions on different interfaces, each library uses the CI Runtime to look up selectors for a CI_ID and update `mcfu_selector`, routing CF instructions to its interface's CFU and state context. Over time, across library calls, `mcfu_selector` is written again and again.

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

1.4.4. `IStateContext` and serializable stateful custom interfaces

The specification defines a custom interface `IStateContext` with four standard custom functions for serializable stateful custom interfaces:

```
interface IStateContext {
//  CF_ID       custom function
    [1023] int  cf_read_status ();
    [1022] void cf_write_status(int status);
    [1021] int  cf_read_state  (int index);
    [1020] void cf_write_state (int index, int state);
};
```

The CFU status indicates cumulative error flags, clean/dirty, and state context size. The read/write state functions access words of the state context.

These standard custom functions enable an interface-aware CI library to access stateful interface specific error status, and an interface-agnostic runtime or operating system to reset, save, and reload state context(s).

1.4.5. CI Application Programming Interface and CI-ABI

The CI-API consists of the *CI Runtime* API, and a calling convention rule. Both are necessary for correct discovery, operation, and composition of CI libraries. As described above (1.4.2) the current `mcfu_selector` CSR selects the

current custom interface/CFU and state context for the hart. However, a CI library should not directly create a CI selector value, nor directly access the CSR. Rather a CI library uses the CI Runtime to look up the CI selector value for its custom interface's CI_ID and to write it to `mcfu_selector`, prior to issuing CF instructions. For example, using a C++ *RAII* object ci to represent a (scoped) custom interface selection:

```
#include "ci.h"                          // CI Runtime: class use_ci { ... }
..
use_ci ci(my_bitmanip_ci_id);            // csrrw mcfu_selector
uint32_t count = cf(pcnt_cf, data, 0);   // cfu_reg cf_id, rd, rs1, rs2
```

The provisional CI-ABI defines a *callee-save* calling convention for `mcfu_selector`. For example, consider CI library functions `a()` and `b()`, for interfaces `A` and `B`, that issue CF instructions `af0`, `af1`, `bf0`, `bf1`, in this program:

```
main() { a(); }
a() { use_ci a_ci(A_ci_id); af0; b(); @1 af1; }
b() { use_ci b_ci(B_ci_id); bf0; bf1; }
```

with execution trace:

```
main() { a() { a_ci(); af0; b() { b_ci(); bf0; bf1; ~b_ci(); } @1 af1; ~a_ci(); }
```

With a callee-save discipline, at point `@1`, upon return from `b()`, the current custom interface must be `A` again. Thus the `b_ci()` constructor saves `a()'s mcfu_selector` value while overwriting it; later its `~b_ci()` destructor restores it. This *RAII* approach also correctly restores `mcfu_selector` in the event of an exception handling stack unwind.

1.5. System composition

1.5.1. Metadata and system manifest

To support automatic composition of CPUs and CFUs into working systems, this specification defines a standard CFU metadata format that details each core's properties, features, and configurable parameters, including CFU-LI feature level, data widths, response latency (or variable), and number of state contexts. Each CPU and CFU 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 CFUs, then uses (manual or automatic) constraint satisfaction to find feasible, optimal parameter settings across these components. The composer may also configure or generate mux and adapter CFUs to automatically interconnect the CPU and the CFUs.

For example, a system composed from a CPU that supports two or three cycle fixed latency CFUs, a CFU_1 that supports response latency of one or more cycles, a CFU_2 that has a fixed response latency of three cycles, and CFU_3 which is combinational (zero cycles latency), overall has a valid configuration with three cycles of CFU latency, with the CPU coupled to a mux CFU, coupled to CFU_1 and CFU_2 and to a *fixed latency adapter CFU*, coupled to CFU_3.

1.5.3. Diversity of systems and operating systems

Composable custom interfaces and CFUs are designed for use across a broad spectrum of RISC-V systems, from a simple RVI20U-Zicsr-Zicfu microcontroller running bare metal fully trusted firmware, to a multicore RVA20S Linux profile, running secure multi-programmed, multithreaded user processes running various CI libraries, and with privileged hypervisors and operating systems securely managing access control to CFUs and CFU state.

1.6. Pushing the envelope

The hardware-hardware and hardware-software interfaces 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 custom interfaces and their CFU implementations.

Whether or not the specific abstractions and interoperation interfaces proposed herein are adopted, we believe this specification motivates custom 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.

1.7. Future directions, TODOs

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

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

1.8. Acknowledgements

Custom Interfaces 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. Custom interfaces: the hardware-software interface

The Custom Interface abstraction bridges software and hardware, enabling diverse software libraries which target the same interface and diverse hardware CFU cores which implement the same interface. Then *custom interface 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 custom interface, identifying a custom function. A **valid CF_ID** is a value that identifies a CF instruction implemented by a configured interface.

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 custom interface 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 custom interface (only).

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

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

A **stateless custom interface** is a fixed named set of set of stateless custom functions.

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

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

`IStateContext` is a stateful custom interface, identified as `CI_ID_IStateContext`, and with four stateful custom functions: {`cf_read_status`, `cf_write_status`, `cf_read_state`, `cf_write_state` }, providing a standard way to manage a custom interface state context. A **serializable custom interface** is a stateful custom interface that inherits IStateContext.

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

A **configured interface subset** is a configured interface in which one or more custom functions of the interface are not implemented. The CF_IDs of unimplemented custom functions are invalid.

A **custom interface state context identifier (STATE_ID)** is an integer index, in the scope of a configured interface, in the range [0, no. of state contexts-1] identifying one of an interface's contexts in the system. A stateless interface

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 interface.

A **custom function instruction (CF instruction)** is a RISC-V custom instruction that executes a custom function using a custom function 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 CFU status CSR with the response status.

A **custom function unit (CFU)** is a core that implements one or more custom interfaces. A **stateful CFU** implements at least one stateful custom interface.

A **CFU_ID** is an integer, in the scope of a system, that identifies a configured interface implemented by a CFU. When one CFU implements multiple configured interfaces, different CFU_IDs identify the configured interfaces. A **valid CFU_ID** is a CFU_ID value that identifies a configured interface.

A **custom interface selector (CI selector, selector)** is a 32-bit value written to `mcfu_selector` CSR to enable custom interface multiplexing and specify the hart's current configured interface / CFU and current state context.

A **CI selector table** is a 4 KB aligned, 4 KB sized table of 1024 CI selectors. When CFU access control (§2.7) is supported, each hart has a `mcfu_selector_table` CSR to address its CI selector table.

A **selector index** is an integer that identifies an entry in a CI selector table (§2.7).

2.2. New CFU control / status registers

A -Zicfu compatible CPU shall implement the `mcfu_selector` and `cfu_status` CSRs for interface multiplexing and custom function execution.

When CFU access control (§2.7) is supported, a -Zicfu compatible CPU shall implement the `mcfu_selector_table` and `cfu_selector_index` CSRs.

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

All CFU CSRs are initialized to zero on reset.

2.2.1. `mcfu_selector` CSR 0xBC0: select active CFU and state context

The `mcfu_selector` CSR implements custom interface multiplexing. It is assigned various CI selectors over time. This enables or disables CI multiplexing and selects the hart's current CFU and state context (within that CFU). It may only be read or written in machine level.

In a privileged architecture system, user level read access to `mcfu_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?

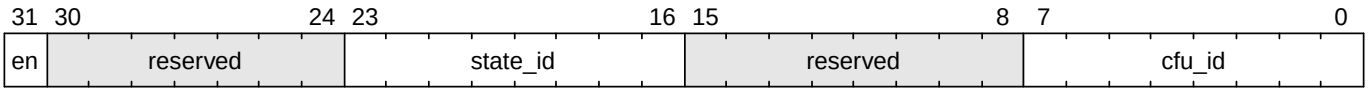

Figure 3. `mcfu_selector` CSR 0xBC0

The `mcfu_selector` CSR has the following fields:

`.en`: enable custom interface multiplexing

- When `en=0`, disable custom interface multiplexing. The `cfu_id` and `state_id` fields are ignored. No CFU is selected. `Custom-0`, `custom-1`, or `custom-2` instructions execute the CPU's built-in custom instructions.
- When `en=1`, enable custom interface multiplexing. The `cfu_id` and `state_id` fields select the current CFU and state context. `Custom-0`/`-1`/`-2` instructions issue CFU requests to the CFU identified by `cfu_id` and to the state context identified by `state_id`.

`.cfu_id`: select the hart's current CFU

- A valid `cfu_id` identifies a configured CFU.
- When enabled, when `cfu_id` does not identify a configured CFU, executing a CF instruction causes an invalid CFU_ID error. The `cfu_status.CI` error bit is set and the CF instruction's destination register, if any, is zeroed.

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

- A valid `state_id` identifies a state context of a CFU.
- When enabled, when `cfu_id` is valid, but `state_id` does not identify a state context of the current CFU, executing a CF instruction causes an invalid STATE_ID error. The `cfu_status.SI` error bit is set and the CF instruction's destination register, if any, is zeroed.

No error occurs when `mcfu_selector` is CSR-written with an invalid CI selector, i.e., when `.cfu_id` or `.state_id` are invalid. Rather, subsequently executing a CF instruction may cause a CFU_ID or STATE_ID error.

2.2.2. `cfu_status` CSR 0x801: CFU status

The `cfu_status` CSR accumulates CFU error flags. It may be written and read in all privilege levels.

Typical application software will write a CI selector to `mcfu_selector`, write 0 to `cfu_status`, execute some CF instructions, and read `cfu_status` to determine if there were any errors.

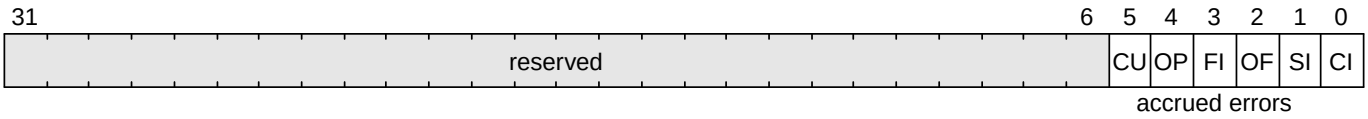

Figure 4. `cfu_status` CSR 0x801

The `cfu_status` CSR has the following fields:

`.CI`: invalid CFU_ID error

- Set by a CF instruction when `mcfu_selector.cfu_id` is invalid.

`.SI`: invalid STATE_ID error

- Set by a CF instruction when `mcfu_selector.cfu_id` is valid but `mcfu_selector`.state_id is invalid.

`.OF`: state context is off error

- Set by a CF instruction when `mcfu_selector.cfu_id` and `mcfu_selector.state_id` are valid but the selected state context is in the *off* state.

`.FI`: invalid CF_ID error

- Set by a CF instruction when `mcfu_selector.cfu_id` and `mcfu_selector.state_id` are valid but the instruction's CF_ID is invalid.

`.OP`: CFU operation error

- Set by a CF instruction when `mcfu_selector.cfu_id`, `mcfu_selector.state_id`, and its CF_ID are valid but there is an error in the requested operation or its operands, in lieu of custom error state.

`.CU`: custom CFU operation error

- Set by a CF instruction of a stateful interface when `mcfu_selector.cfu_id`, `mcfu_selector.state_id`, and its CF_ID are valid but there is an error in the requested operation or its operands, with custom (interface-defined) error state available.

The custom error state of a stateful interface may be obtained using custom functions of the interface. In addition, the custom error state of a serializable interface may also be obtained using `IStateContext` *custom functions* `cf_read_status` *and/or* `cf_read_state`.

2.2.3. `mcfu_selector_table` CSR 0xBC1: CFU selector table base

When CFU access control (§2.7) is supported, the `MXLEN`-bit-wide `mcfu_selector_table` CSR specifies the base address of thehart's CI selector table. The CSR may be read and written in machine level.

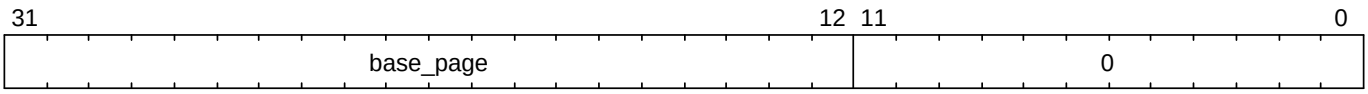

Figure 5. mcfu_selector_table CSR 0xBC1 (when MXLEN=32)

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

2.2.4. `cfu_selector_index` CSR 0x800: CFU selector index

When CFU access control (§2.7) is supported, the `cfu_selector_index` CSR selects an entry from the hart's CI selector table entry to write to the `mcfu_selector` CSR. The CSR may be read and written in all privilege levels.

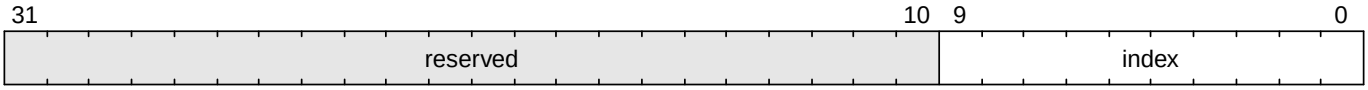

Figure 6. cfu_selector_index CSR 0x800

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

In response to CSR-write of `cfu_selector_index`, load the 32-bit CI selector at address (`mcfu_selector_table + cfu_selector_index.index*4`) and CSR-write the CI selector to `mcfu_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) (§2.7).

2.2.5. Implicit CFU CSR fences

Per hart, there is an implicit fence between any CFU CSR access and any series of `custom-0`/`-1`/`-2` instructions. All CFU CSR accesses happen before any CF instructions which follow, and all CF instructions happen before any CFU CSR accesses that follow.

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

2.3. Custom function instruction encodings

When `mcfu_selector.en=1`, software issues CF instructions to the current state context of the current interface (i.e., of the current configured CFU) 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: `cfu_reg cf_id,rd,rs1,rs2`

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

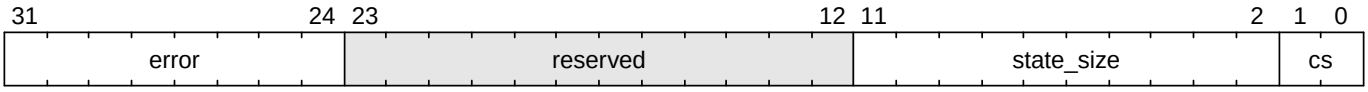

Figure 7. CFU R-type instruction encoding

2.3.2. Custom-1 I-type encoding

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

An I-type CF instruction issues a CFU request for a zero-extended 4-bit CF_ID `cf_id` with one source register operand identified by `rs1` and a signed-extended 8-bit immediate value `imm`. The CFU response is written to destination register `rd`.

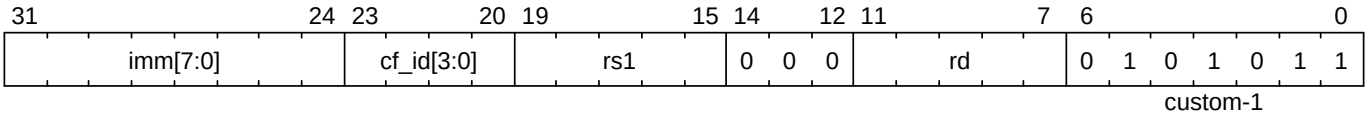

Figure 8. CFU I-type instruction encoding

This new, irregular immediate field encoding may have a disproportionate impact on area and critical path delay in the decode or execute pipeline stages of a RISC-V processor core.

Seven-eighths of the custom-1 encoding space is reserved for future custom function instruction encodings.

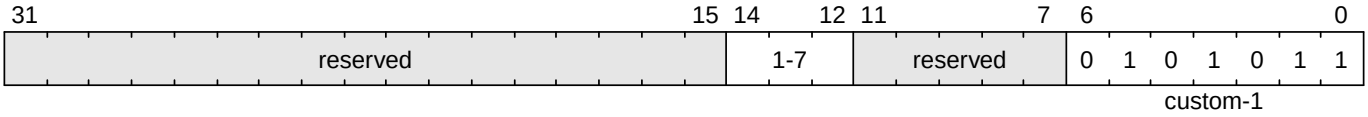

Figure 9. CFU reserved I-type instruction encodings

2.3.3. Custom-2 flex-type encoding

Assembly instruction: `cfu_flex cf_id,rs1,rs2`
Assembly instruction: `cfu_flex25 custom`

A flex-type CF instruction issues a CFU 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 CFU response data is discarded. The instruction is executed purely for its effect upon the selected state context of the selected CFU.

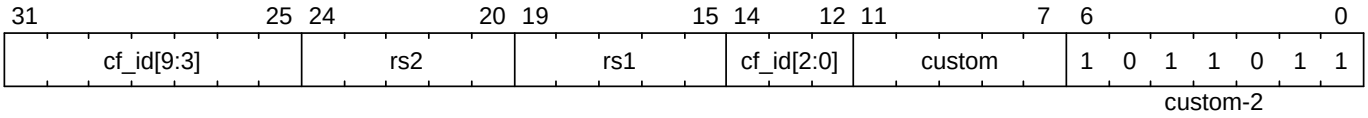

Figure 10. CFU flex-type instruction encoding

Alternatively, equivalently, the `cfu_flex25` form of instruction issues an arbitrary 25-bit custom instruction.

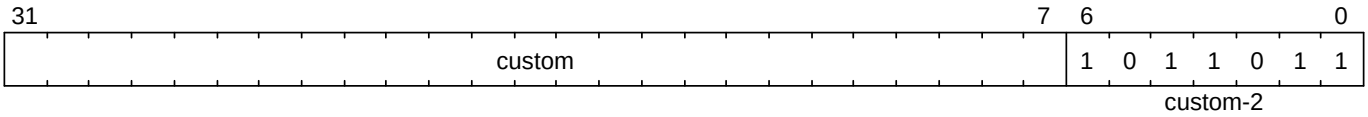

Figure 11. CFU flex-type instruction alternate encoding

A flex-type CF instruction may be used with a CFU-L2 request's raw instruction field `req_insn` (3.4.5) to provide an arbitrary 32-7=25-bit custom request to a CFU. The absence of an (integer) destination register field is a feature that provides added, CPU-uninterpreted, custom instruction bits to a CFU.

One disadvantage of this approach: when the selected CFU 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. Custom function instruction execution via custom interface multiplexing

Figure 12 illustrates how a custom function instruction and the CFU CSRs implement custom interface / CFU composition via custom interface multiplexing. When the CPU issues a custom function instruction, it produces a CFU request from the fields of the instruction, two source operands from the register file and/or an immediate field of the instruction, and the `cfu_id` and `state_id` fields of `mcfu_selector`. The CFU request may include the request ID cookie (defined by the CPU), the CFU_ID, STATE_ID, raw instruction, CF_ID, and operands. The CFU_ID identifies which CFU must process the request. The CFU includes state context(s) and a datapath. The STATE_ID selects the state context to use for this request. The CFU checks for errors in CFU_ID, STATE_ID, and CF_ID per 2.2.2, processes the request, possibly updating this state context, and produces a CFU response, which may include the same request ID cookie, a success/error status, and the response data. The CPU commits the custom function instruction by updating `cfu_status` (when response status is an error condition) and writing the response data to the destination register.

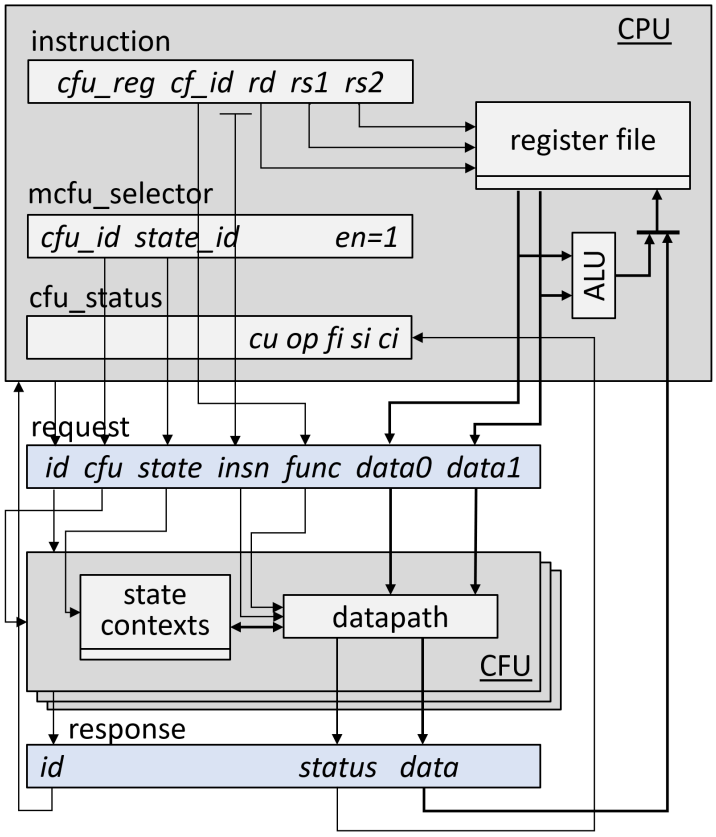

Figure 12. HW-SW interface: flow of information for execution of a custom function instruction

Multiple custom function instructions may be in flight at the same time, particularly in a system with pipelined CPUs or pipelined CFUs. 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` fields of the custom function instruction and the `mcfu_select` and `cfu_status` CSRs, to the CFU Logic Interface's request and response signals (§3.4).

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

CFU-LI signal	← Source or → Destination
`req_id`	← CPU
`req_cfu`	← `mcfu_select.cfu_id`
`req_state`	← `mcfu_select.state_id`
`req_insn`	← `insn`
`req_func`	← `insn.cf_id`
`req_data0`	← R[`insn.rs1`]
`req_data1`	← R[`insn.rs2`] {`custom-0`/`-2`} or `insn.imm` {`custom-1`}
`resp_id`	→ CPU
`resp_status`	→ `cfu_status` bits
`resp_data`	→ R[`insn.rd`] {`custom-0`/`-1`}

2.4.1. Precise exceptions

Custom function instruction execution preserves precise exception semantics. If an instruction preceding (in

execution order) a custom function instruction is an exception, the custom function instruction does not execute, and has no effect upon architected state, including the `cfu_status` CSR, and no effect on the current state context of the custom interface / CFU.

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

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

A CPU may not speculatively issue a CF instruction to a stateful CFU because the instruction may update the current state context and the CFU Logic Interface has no means to cancel a CFU request. In other words, a CF instruction of a stateful CFU, 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. CF instructions of stateful CFUs must not issue in the wake of an instruction that may yet trap.

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

How can a CPU core determine dynamically whether a CF instruction, or its custom interface, is stateless?

2.5. `IStateContext`: the standard custom functions

The `IStateContext` custom interface defines four standard custom functions to manage interface state context data. Stateful custom interfaces should (albeit not *must*) inherit from this interface, i.e., incorporate these four custom functions. `IStateContext` provides a standard, uniform way to access the interface's custom error state and enables an interface-agnostic runtime or operating system to reset, save, and reload state contexts.

Table 2. Standard stateful custom functions

Custom function	CF_ID	Assembly instruction	Encoding
`cf_read_status`	1023	`cfu_read_status rd`	`cfu_reg 1023,rd,x0,x0`
`cf_write_status`	1022	`cfu_write_status rs1`	`cfu_reg 1022,x0,rs1,x0`
`cf_read_state`	1021	`cfu_read_state rd,rs1`	`cfu_reg 1021,rd,rs1,x0`
`cf_write_state`	1020	`cfu_write_state rs1,rs2`	`cfu_reg 1020,x0,rs1,rs2`

CF_IDs 1008-1023 (0x3F0-0x3FF) are reserved for standard custom functions. It is recommended, not mandatory, that these CF_IDs not be used for another purpose.

Any CF instruction with CF_ID=1023 must be side effect free, i.e., never modify any CFU state.

2.5.1. Interface state context status word

The cf_read_status and cf_write_status functions access the selected interface state context's status word.

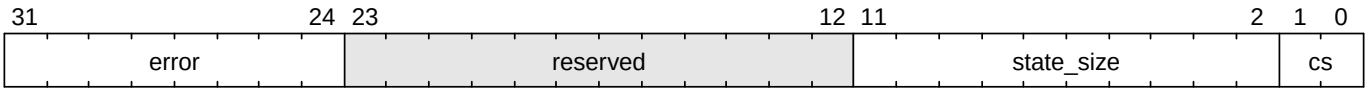

Figure 13. CFU state context status word

The interface state context status word has the following fields:

.cs: context status

- The state context has four context status values: { 0: off; 1: initial; 2: clean; 3: dirty } which correspond 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 serialable stateful interface CFU 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 CF instructions except cf_write_status and cf_read_status signal CFU_ERROR_OFF, until the state context status is set to another state by a subsequent cf_write_status.
- A write .cs=1 has the side effect of resetting the entirey to the *current* state context to its initial (power up) state.
- When a CF instruction modifies any aspect of the current state context of a serializable CFU, its state context status automatically changes to dirty.

.state_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.
- Writes return the previous size and cs status of the current state context.
- Different CFU implementations of the same custom interface may have different state context sizes.
- Different state contexts of the same CFU may have different state context sizes.
- At different times, the same state context of the same CFU may have different state context sizes.

.error: custom error status

- An 8-bit custom error status for the current interface / CFU and its state context.

2.5.2. cfu_read_status standard custom function instruction

Assembly instruction: cfu_read_status rd

This instruction retrieves the state status word (§2.5.1) of the selected state context of the selected CFU and writes it to the rd destination register.

cfu_read_status can never modify the selected state context, nor modify the behavior of the interface.

The status word .state_size field may change as a side effect of executing a stateful CF instruction.

For the CF instruction sequence [`cfu_read_status`; `cfu_read_state`*; `cfu_read_status`], the first and second `cfu_read_status` must return the same `.state_size`.

For the CF instruction sequence [`cfu_read_status`, *any-other-CF-instruction* *, `cfu_read_status`], the first and second `cfu_read_status` need not return the same `.state_size`.

For most stateful CFUs, the size of a state context is fixed. For some stateful CFUs, the size of a state context may depend upon the sequence of CF instructions performed. For example, a stateful vector math CFU 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.

`cfu_read_status` may be used as a *probe* after a `mcfu_selector` write, to check whether the selector addresses a valid CFU and state context:

```
csrw mfcu_selector,x1   ; select some CFU and state context
csrw cfu_status,x0      ; clear cfu_status
cfu_read_status x0      ; probe, discarding state status word
csrr x2,cfu_status      ; retrieve cfu_status
...                     ; cfu_status.ci => invalid CFU_ID
...                     ; cfu_status.si => invalid STATE_ID
```

2.5.3. `cfu_write_status` standard custom function instruction

Assembly instruction: `cfu_write_status rs1`

This instruction writes the value of the `rs1` source register to the state status word of the selected state context of the selected CFU, and writes the previous value of the state context status word to the `rd` destination register.

A write `.cs=1` always has the side effect of resetting the selected state context to its initial (power up) state.

For the sequence [`cfu_write_status`; *; `cfu_read_status`] the value of `.state_size` read need not equal the last value written.

A `cfu_write_status` CF instruction never has any effect upon any other state context of the CFU, or of any ohter CFU.

2.5.4. `cfu_read_state` standard custom function instruction

Assembly instruction: `cfu_read_state rd,rs1`

This instruction reads one (XLEN-bit) word of state, at the index specified by the `rs1` source register, from the selected state context of the selected CFU, and writes it to the `rd` destination register.

2.5.5. `cfu_write_state` standard custom function instruction

Assembly instruction: `cfu_write_state rs1,rs2`

This instruction reads the value of the `rs2` source register and writes it to the selected state context of the selected CFU at the index specified by the value of the `rs1` source register. It also writes the value of the `rs2` source register to the `rd` destination register. It silently drops attempts to write state at an invalid state index.

2.6. Resource management and context switching

A software resource manager (e.g., thread pool, language runtime, language virtual machine, RTOS, operating system, hypervisor) multiplexes software loci of execution (e.g., request, worker, actor, activity, task, fiber, continuation, thread, process), *locus* for short, 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. Composable interfaces extension **-Zicfu** extends per-hart state with the CFU CSRs (§2.2) and the various configured state contexts of the stateful configured custom interfaces.

A CFU implementing a stateful custom interface is typically 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 CFU 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 (per-CFU) state contexts. The resource manager consults a *system CFU map* specifying the mapping CFU_IDs of the configured interfaces of the system, and for each interface/CFU, the no. of state contexts it is configured with. A stateless CFU has zero contexts.

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

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

```
li a1,{.error=0,.cs=1/*initialize*/}
lw a0,selectors[i]
csrw mcfu_selector,a0
cfu_write_status a1
```

This resets that state context to its initial state.

To save hart state, for each interface state context of the hart, first execute

```
lw a0,selectors[i]
csrw mcfu_selector,a0
cfu_read_status a0
sw a0,status[i]
```

to obtain `a0.state_size`, the size (in XLEN-bit words) of the serialized state context for the selected state context. Allocate array `save[i][]` to store the serialized state context. For each word in `a0.state_size`, execute

```
cfu_read_state a0,j
sw/sd a0, save[i][j]
```

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

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

```
lw a0, selectors[i]
csrw mcfu_selector, a0
lw a0, status[i]
cfu_write_status a0
```

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

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

to restore each word of the state context.

When different CFUs implement the same custom interface, 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 CFU. `cfu_selector_index` *may (but should not) change. However, resource manager must change* `mcfu_selector`.

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

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

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

2.7. CFU access control

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

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

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

Alternatively, the machine level illegal instruction exception handler can determine whether the new CI 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 custom interface multiplexing should take, at most, a few clock cycles.

The optional CFU access control CSRs `mcfu_selector_table` and `cfu_selector_index` allow less privileged code *user code* to rapidly multiplex custom interfaces, but only among those interfaces and state contexts granted access

by more privileged code *system code*.

CFU 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 *CI selector table*, 4 KB aligned, comprising 1024 32-bit CI selectors, which is read/write to system code and inaccessible from user code. Initially the table is zero filled, as zero is a valid CI selector (disables custom interface multiplexing). The system code CSR-writes its address to the hart's `mcfu_selector_table` CSR. Then in response to a system call requesting access to an interface, and one of its state contexts, system code determines whether the access is granted. If so, it determines the CI selector value for it, allocates an entry for that CI selector value in the CI 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 CI/CFU and its state, user code CSR-writes its index to `cfu_selector_index`. In response, the CPU loads from memory (at more privileged level) the CI selector word at that index in the selector table and CSR-writes it to `mcfu_selector` — no exception handling detour required.

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

3. Custom Function Unit Logic Interface

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

3.1. Definitions

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

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

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

A **CFU response status (response status, status)** is a CFU-LI success/error code produced by a CFU in response to receiving a CFU request, indicating success or else an error in the request's CFU_ID, CF_ID, STATE_ID, operation, or a custom interface specific error.

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

A **CPU** is a CFU requester that implements RISC-V RV-I-Zicsr-Zicfu instruction set, issues CFU requests upon issuing CF instructions, and writes a destination register and the CFU status CSR in response to CFU responses.

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

A **Mux CFU (mux)** is an intermediary CFU. For each request received, the mux either sends a response itself (e.g., a CFU_ERROR_CFU response) or arbitrates and forwards the request to a subordinate CFU, and later forwards the corresponding response to the original requester.

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

A **configured system (system)** is a computer system including one or more CPUs and zero or more CFUs that implement a set of configured custom interfaces.

3.2. Example configured system

Figure 14 illustrates a configured system composed of two CPUs and five CFUs, plus a mux, and a level adapter for CFU_3. Each CPU has two harts. CFUs 0-2 are stateful and CFUs 3-4 are stateless. Each stateful CFU has one state context per hart. CFU_1 has an additional state context per hart for isolated stateful requests from CFU_2.

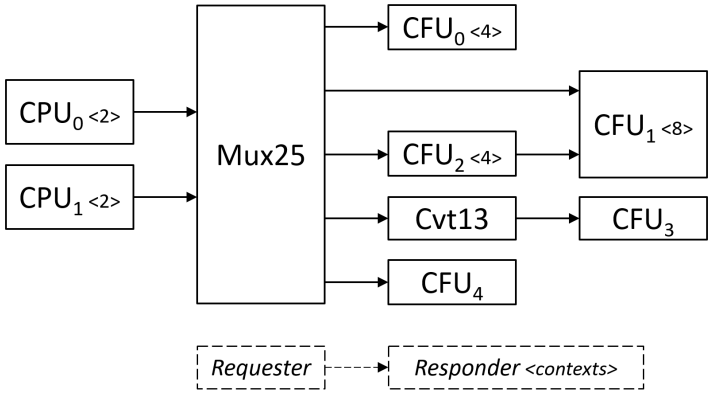

Figure 14. Configured system composed of two CPUs and five CFUs

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

3.3. CFU-LI feature levels

The CFU-LI is stratified into separate feature levels: -L0:combinational; -L1: fixed latency; -L2: variable latency; -L3:elastic; and -L4:reordering. Each feature level adds yet more CFU 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. CFU-L0: combinational CFU

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

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

3.3.2. CFU-L1: fixed latency CFU

Each cycle, the CFU computes a function of the CFU request **and the specified state context, if any, updating the context**, sending a CFU 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. CFU_INIT_INTERVAL=1+.

3.3.3. CFU-L2: variable latency, request-only flow control CFU (variable latency CFU)

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

Example: a multiply-divide unit with a variable-latency multi-cycle divide, with early-out, and which is not ready to receive requests while the divider is busy.

3.3.4. CFU-L3: variable latency, request/response flow control CFU (elastic CFU)

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

Example: the above multiply-divide unit but also with response queueing or elastic pipeline control to support response backpressure.

Perhaps **streaming CFU** *would be a better name.*

3.3.5. CFU-L4: reordering CFU

The CFU computes a function of the CFU request and the specified state context, if any, updating the context, and sending a CFU 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.

CFU-L4 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.

3.3.6. Feature levels summary

In summary, all CFU-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, request flow control, request ID, and raw instruction. Level 3 adds response flow control. Level 4 adds reordering. (Table 3.)

Table 3. CFU-LI feature levels summary

Level	CFU type	Req valid, func, data, resp data, status	Clock, reset, clock enable, state ID, resp valid	Req ready, raw insn, resp ID	Resp ready	Reorder responses, req ID
0	combinational	Y				
1	fixed latency	Y	Y			
2	variable latency	Y	Y	Y		
3	elastic	Y	Y	Y	Y	
4	reordering	Y	Y	Y	Y	Y

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

3.4. CFU-LI signaling

CFU cores *of a particular feature level* implement a common set of request and response signals. Table 4 lists all CFU-LI signals of all feature levels in a canonical order: transaction signals (request/response valid, ready, REQ_ID), context (CFU_ID, STATE_ID), function (raw instruction, CF_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 CFU (§3.4.1).

Table 4. All CFU-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
4	in	req_id	CFU_REQ_ID_W	request REQ_ID
	in	req_cfu	CFU_CFU_ID_W	request CFU_ID
1+	in	req_state	CFU_STATE_ID_W	request STATE_ID
2+	in	req_insn	CFU_INSN_W	request raw instruction
	in	req_func	CFU_FUNC_ID_W	request CF_ID
	in	req_data0	CFU_DATA_W	request operand data 0
	in	req_data1	CFU_DATA_W	request operand data 1
1+	out	resp_valid		response valid
3+	in	resp_ready		response ready
4	out	resp_id	CFU_REQ_ID_W	response ID
	out	resp_status	CFU_STATUS_W	response status
	out	resp_data	CFU_DATA_W	response data

All signals are positive-true logic.

It is unfortunate the custom function ID is **CF_ID** *in the HW-SW interface and* **FUNC_ID** *in the CFU-LI.*

3.4.1. CFU-LI configuration parameters

Table 5 presents CFU-LI bit vector width parameters and ranges of possible values.

Table 5. CFU-LI width configuration parameters

Level	Quantity	Width Parameter	Range	Default	Description
4	REQ_ID	CFU_REQ_ID_W	0-64	0	request/response ID width
	CFU_ID	CFU_CFU_ID_W	0-16	0	CFU_ID width
1+	STATE_ID	CFU_STATE_ID_W	0-16	0	STATE_ID width
2+	insn	CFU_INSN_W	0, 32	0	raw instruction width
	CF_ID	CFU_FUNC_ID_W	0-10	10	CF_ID width
	data	CFU_DATA_W	32, 64	32	request/response data width
	status	CFU_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.

When CFU_FUNC_ID_W<10, how do standard custom functions (CF_ID in [0x3F0..0x3FF]) work?

Table 6 presents other CFU configuration parameters.

Table 6. CFU-LI: other CFU configuration parameters

Level	Parameter	Range	Default	Description
	CFU_VERSION	100	100	CFU-LI version; 100 == 1.00
	CFU_CFU_ID_MAX	1+	1	number of CFUs at/below this CFU
1+	CFU_STATE_ID_MAX	0+	0	number of custom interface state contexts
1	CFU_LATENCY	0+	1	latency (clock cycles) from a request to its response
1	CFU_RESET_LATENCY	0+	0	min. latency (clock cycles) from negation of reset to first request

CFU_VERSION is the CFU-LI version number the CFU is configured to implement, encoded as 100: the decimal version number.

This records, in code, the CFU-LI version implemented by a CFU, and anticipates evolution of CFU-LI.

CFU_CFU_ID_MAX is the number of logical CFUs at/below this CFU. For a leaf CFU this may be more than one when the CFU implements multiple custom interfaces (including multiple versions of one custom interface).

CFU_STATE_ID_MAX is the number of custom interface state contexts for every stateful interface implemented by this CFU. It must be 0 if every custom interface implemented by the CFU is stateless. It must be 1+ if any custom interface implemented by the CFU is stateful. When a leaf CFU implements multiple stateful custom interfaces, i.e. CFU_CFU_ID_MAX>1, each must be configured with the same number of state contexts.

CFU_LATENCY and CFU_RESET_LATENCY are specific to CFU-L1 fixed latency CFUs. See §3.3.2.

3.4.2. Clock, reset, clock enable

CFU-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 supercedes all other CFU-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 CFU-L1 CFU (which does not have a req_ready output) must be ready to receive its first request after no more than its configured CFU_RESET_LATENCY clock cycles following negation of rst.

A clock enable input signal clk_en facilitates clock gating of a CFU. When clk_en is asserted on posedge clk, synchronous elements of the CFU (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 CFU. CFU operation is suspended. Therefore, when negating clk_en, a CFU requester must disregard all CFU 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 CFU 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 `CFU_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 CFUs.

3.4.3. Request and response valid-ready flow control

CFU-L2, -L3, and -L4 provide CFU request channel synchronous valid-ready flow control. CFU-L3 and -L4 also provide CFU response channel synchronous valid-ready flow control.

With synchronous valid-ready flow control, 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.

For feature levels that include both request and response flow control, a requester may 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 CFU request, the CFU error-checks the request data, performs the request, and outputs the first (i.e., lowest numbered) `[2:0] resp_status` condition that applies:

Table 7. CFU response status values and conditions

Name	Value	Condition
`CFU_OK`	0	no errors occurred processing request
`CFU_ERROR_CFU`	1	`req_cfu` is not a CFU_ID implemented by CFU
`CFU_ERROR_STATE`	2	`req_state` is not a valid STATE_ID for `req_cfu`
`CFU_ERROR_OFF`	3	`req_state` is valid but this *serializable* state context is in the *off* state
`CFU_ERROR_FUNC`	4	`req_func` is not a valid CF_ID for `req_cfu`
`CFU_ERROR_OP`	5	request operand(s) or state are a domain error for the custom function
`CFU_ERROR_CUSTOM`	6	request causes a custom error (of a serializable custom interface)

When parameter `CPU_CFU_ID_W=0`, `req_cfu` is ignored: no `CFU_ERROR_CFU` errors.

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

`STATE_ID=0` is the only valid STATE_ID for the CFU of a stateless custom interface.

CFU state may change if and only if the response status is one of `CFU_OK`, `CFU_ERROR_OP`, or `CFU_ERROR_CUSTOM`.

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

In response to receiving `resp_status` of `CFU_ERROR_CFU`, `CFU_ERROR_STATE`, `CFU_ERROR_OFF`, or `CFU_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-0`/`-1` but not `custom-2`), the result of the CF instruction is written to the register, irrespective of the CFU 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 { CFU, state, function } ID errors are well behaved at the hardware-software interface. By making the CPU responsible for zeroing such results, each CFU in a system's CFU DAG need not incur redundant logic and delay to respond `resp_data=0` on these three errors. For synchronously signaled CFU-LI levels, in an FPGA, with reset-able flip-flops, a registered `resp_data` input may be zeroed for negligible cost.

3.4.5. Raw instruction

At CFU-LI feature level 2, or higher, CFU requests may be configured (`CFU_INSN_W=32`) to include the raw instruction word (`req_insn`) of the CF instruction issued the CFU request, if the request originates from a CF instruction, or all zeroes otherwise. A CFU may use the raw instruction data to help perform a custom function, or it may ignore the raw instruction entirely.

The raw instruction complements the CF_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 CFU, and potentially reduces verification complexity.

However, access to the raw CF instruction word can enable additional use cases. As an example, consider a CFU with a private vector, matrix, or complex number register file. When this CFU receives a CFU 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 CF instruction bits, to determine which of its CFU register file entries to read. Similarly, the CFU can decode the raw instruction word to recover an `rd` field to determine which CFU-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 CFUs, snoopily watching the dynamic instruction stream go by, even when `req_valid` is negated.

Half-baked idea (not recommended): Imagine a dynamic facility by which any arbitrary instruction word, not just `custom-0`/`-1`/`-2` format instructions, may be a CF instruction, issued to a CFU. This might be a table of (mask,pattern) tuples, or a 32-bit `mcfu_opcodes_mask` CSR bitvector of 5-bit major opcodes, identifying instructions to divert to the current CFU. Or perhaps, in the hardware domain, a CPU might frst issue each instruction to the current CFU, and only execute the instruction in the CPU if the CFU delegates it back to the CPU.

3.4.6. Request-response ID

CFU-LI feature level 4 (reordering CFU) 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 CFU receives the REQ_ID as `req_id`, and later, with each response, the CFU sends back the same REQ_ID as `resp_id`. For each request/response pair, the CFU must send the requester the identical request-response ID value that the requester previously sent to the CFU.

Operation and behavior of a CFU 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. CFU-L0 combinational CFU signaling

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

3.5.1. CFU-L0 configuration parameters

Table 8. CFU-L0 configuration parameters

Parameter	Description
`CFU_VERSION`	CFU-LI version number
`CFU_CFU_ID_MAX`	number of CFUs at/below this CFU

For `CFU_VERSION` and `CFU_CFU_ID_MAX`, see §3.4.1.

3.5.2. CFU-L0 signals

Table 9. CFU-L0 signals

Dir	Port	Width Parameter	Description
in	`req_valid`		request valid
in	`req_cfu`	`CFU_CFU_ID_W`	request CFU_ID: selects the requested CFU
in	`req_func`	`CFU_FUNC_ID_W`	request CF_ID
in	`req_data0`	`CFU_DATA_W`	request operand data 0
in	`req_data1`	`CFU_DATA_W`	request operand data 1
out	`resp_status`	`CFU_STATUS_W`	response status
out	`resp_data`	`CFU_DATA_W`	response data

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

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

3.5.3. CFU-L0 signaling protocol

Protocol:

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

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

3.5.4. CFU-L0 example

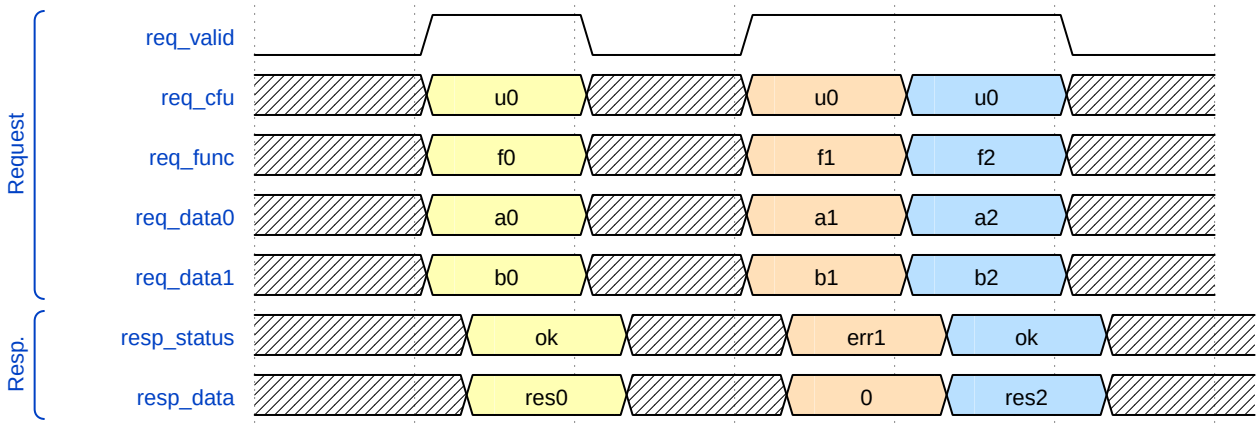

Figure 15. Example CFU-L0 signaling protocol waveform

Figure 15 is an example waveform for three CFU-L0 requests and responses, arising from executing CF instructions `f0(a0,b0)`, `f1(a1,b1)`, and `f2(a2,b2)`. All three instructions issue to the same CFU `u0`. Function `f1` incurs an error.

3.6. CFU-L1 fixed latency CFU signaling

Each cycle, a fixed latency CFU computes a function of the CFU request **and the specified state context, if any, updating the context**, sending a CFU 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 CFU-L1 CFU is configured with multiple requesters, requesters must not send multiple simultaneous requests.

3.6.1. CFU-L1 configuration parameters

Table 10. CFU-L1 configuration parameters

Parameter	Description
CFU_VERSION	CFU-LI version number
CFU_CFU_ID_MAX	number of CFUs at/below this CFU
CFU_STATE_ID_MAX	number of custom interface state contexts
CFU_LATENCY	latency (clock cycles) from a request to its response
CFU_RESET_LATENCY	minimum latency (clock cycles) from negation of reset to first request

For CFU_VERSION, CFU_CFU_ID_MAX, and CFU_STATE_ID_MAX, see §3.4.1.

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

A CFI-L1 CFU with CFU_LATENCY=0 resembles a CFU-L0 combinational CFU, except it may implement a stateful custom interface.

Example: an extended precision arithmetic CFU 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.*

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

3.6.2. CFU-L1 signals

Table 11. CFU-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_cfu	CFU_CFU_ID_W	request CFU_ID
in	req_state	CFU_STATE_ID_W	request STATE_ID
in	req_func	CFU_FUNC_ID_W	request CF_ID
in	req_data0	CFU_DATA_W	request operand data 0
in	req_data1	CFU_DATA_W	request operand data 1
out	resp_valid		response valid
out	resp_status	CFU_STATUS_W	response status
out	resp_data	CFU_DATA_W	response data

3.6.3. CFU-L1 signaling protocol

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

Protocol:

1. Request transfer.

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

b. `CFU_LATENCY=0`: CFU receives CFU request asynchronously.
`CFU_LATENCY>0`: CFU receives CFU request on posedge `clk`.

2. Custom function execution.

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

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

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

3. Response transfer.

a. `CFU_LATENCY=0`:

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

ii. Requester receives CFU response asynchronously.

b. `CFU_LATENCY>0`:

i. After (`CFU_LATENCY-1`) cycles, CFU asserts `resp_valid`, `resp_status`, and `resp_data`.

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

3.6.4. CFU-L1 example

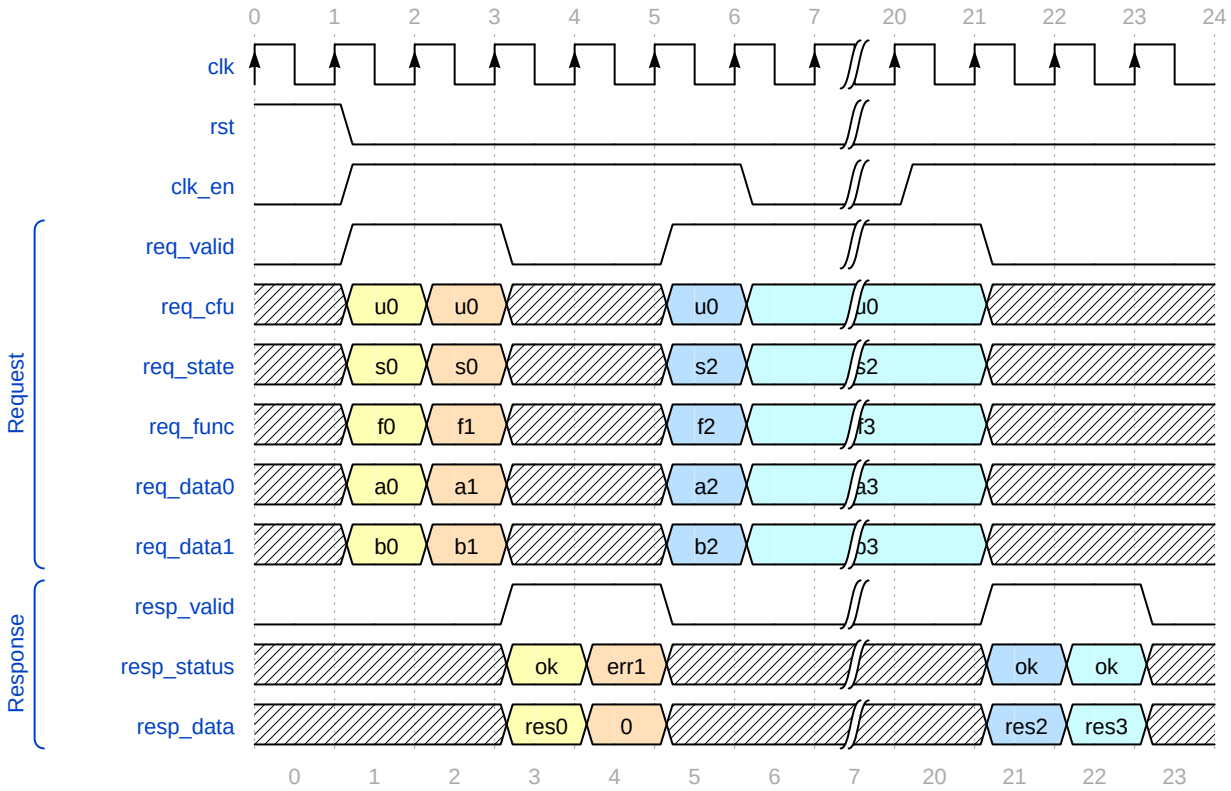

Figure 16. Example CFU-L1 signaling protocol waveform (`CFU_LATENCY=2`, `CFU_RESET_LATENCY=0`)

Figure 16 is an example waveform for four CFU-L1 CFU requests and responses, arising from executing four CF instructions `f0`-`f3`. Since `CFU_RESET_LATENCY=0`, the CFU is ready for request `f0` in cycle 1, the same cycle `rst` is negated. With `CFU_LATENCY=2`, each response occurs 2 (enabled) clock cycles after each request is received. Each instruction issues a CFU request to the same CFU `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 CFU 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. CFU-L2 variable latency CFU signaling

A variable latency CFU computes a function of the CFU request and the specified state context, if any, updating the context, sending a CFU response, **in order, in a later clock cycle**. There is **request flow control**.

When the requester is a CPU, use of CFU-L2 means the CPU must be ready to accept a response from the CFU on any cycle. This simplifies the design of the CFU but may complicate the design of the CPU pipeline and its register file write arbitration logic.

3.7.1. CFU-L2 configuration parameters

Table 12. CFU-L2 configuration parameters

Parameter	Description
`CFU_VERSION`	CFU-LI version number
`CFU_CFU_ID_MAX`	number of CFUs at/below this CFU
`CFU_STATE_ID_MAX`	number of custom interface state contexts

For `CFU_VERSION`, `CFU_CFU_ID_MAX`, and `CFU_STATE_ID_MAX`, see §3.4.1.

3.7.2. CFU-L2 signals

Table 13. CFU-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_cfu`	`CFU_CFU_ID_W`	request CFU_ID
in	`req_state`	`CFU_STATE_ID_W`	request STATE_ID
in	`req_insn`	`CFU_INSN_W`	request raw instruction
in	`req_func`	`CFU_FUNC_ID_W`	request CF_ID
in	`req_data0`	`CFU_DATA_W`	request operand data 0
in	`req_data1`	`CFU_DATA_W`	request operand data 1
out	`resp_valid`		response valid
out	`resp_status`	`CFU_STATUS_W`	response status
out	`resp_data`	`CFU_DATA_W`	response data

3.7.3. CFU-L2 signaling protocol

CFU-L2 is synchronous to posedge `clk`. See §3.4.2. CFU-L2 includes the request's raw instruction. See §3.4.5.

Protocol:

1. Request transfer.
   a. Requester asserts CFU request signals `req_*` and asserts `req_valid`.
   b. Responder may assert `req_ready`.
   c. CFU receives CFU request on posedge `clk` when `req_valid` and `req_ready` are both asserted, per §3.4.3.
2. Custom function execution.
   a. CFU performs response status / error checking per §3.4.4.
   b. CFU performs a custom function of the operands and the selected state context.
   c. CFU 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) CFU asserts `resp_status` and `resp_data` and asserts `resp_valid`.
   b. Requester receives CFU response on posedge `clk`.

3.7.4. CFU-L2 example

Figure 17 is an example waveform for three CFU-L2 CFU requests and responses, arising from executing three CF instructions `f0`-`f2`. (Assume `CFU_INSN_W=0`, no `req_insn`.) Each instruction issues a CFU request to the same CFU `u0`. Instructions `f0` and `f1` use state context `s0`; `f2` uses state context `s2`. The CFU receives request `f0` in cycle 2 and responds in cycle 3, a latency of 1 cycle. Requester asserts request `f1` in cycle 3, but it is not received by the CFU until it reasserts `req_ready` in cycle 4. The CFU responds to `f1` in cycle 6, with an error response, a latency of 2 cycles. Requester asserts request `f2` in cycle 6, but it is not received by the CFU until it reasserts `req_ready` in cycle 7. The CFU responds to `f2` in cycle 10, a latency of 3 cycles.

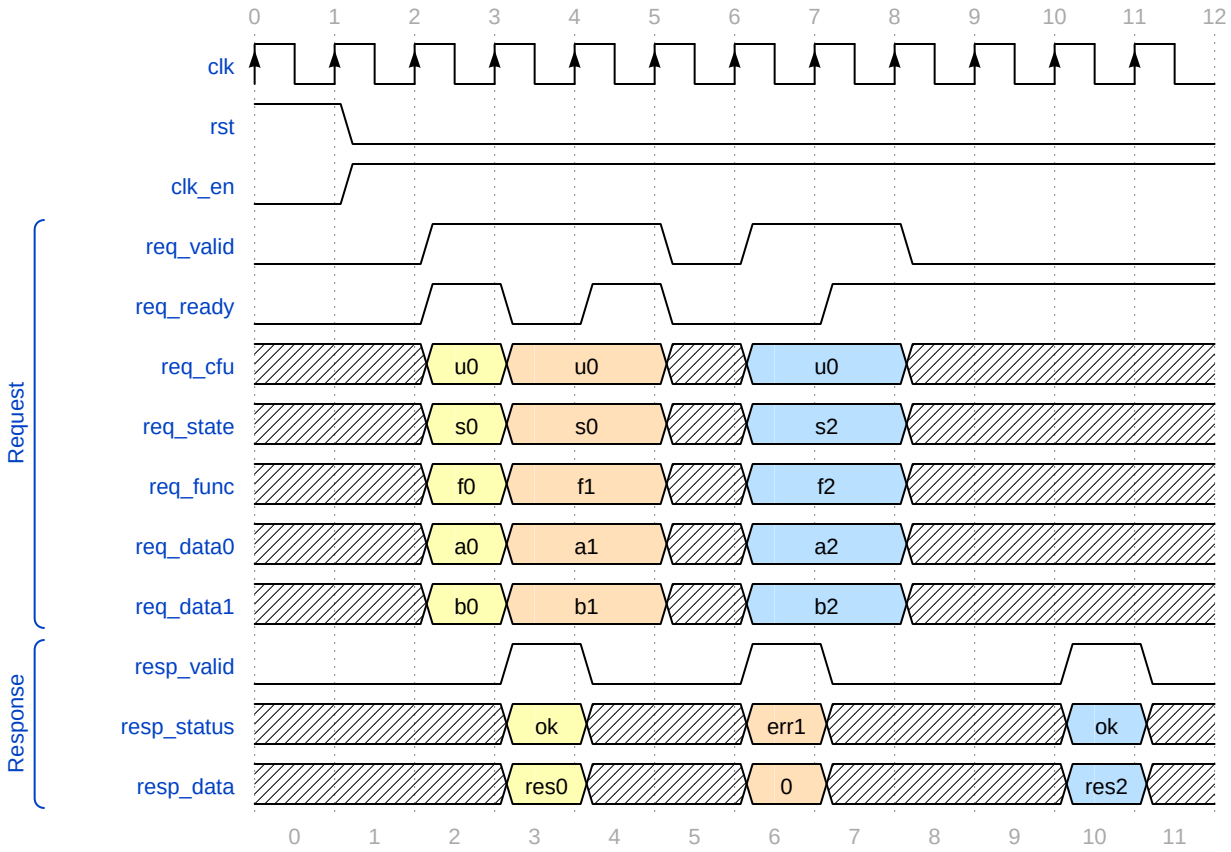

Figure 17. Example CFU-L2 signaling protocol waveform

3.8. CFU-L3 elastic CFU signaling

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

When the requester is a CPU, use of CFU-L3 allows the CPU to delay receipt of a CFU response. This affords the CPU pipeline greater flexibility to dynamically prioritize other units' accesses to register file write port(s). Conversely, CFU-L3 may complicate design of the CFU, which may 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.*

3.8.1. CFU-L3 configuration parameters

Table 14. CFU-L3 configuration parameters

Parameter	Description
`CFU_VERSION`	CFU-LI version number
`CFU_CFU_ID_MAX`	number of CFUs at/below this CFU
`CFU_STATE_ID_MAX`	number of custom interface state contexts

For `CFU_VERSION`, `CFU_CFU_ID_MAX`, and `CFU_STATE_ID_MAX`, see §3.4.1.

3.8.2. CFU-L3 signals

Table 15. CFU-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_cfu	CFU_CFU_ID_W	request CFU_ID
in	req_state	CFU_STATE_ID_W	request STATE_ID
in	req_insn	CFU_INSN_W	request raw instruction
in	req_func	CFU_FUNC_ID_W	request CF_ID
in	req_data0	CFU_DATA_W	request operand data 0
in	req_data1	CFU_DATA_W	request operand data 1
out	resp_valid		response valid
in	resp_ready		response ready
out	resp_status	CFU_STATUS_W	response status
out	resp_data	CFU_DATA_W	response data

3.8.3. CFU-L3 signaling protocol

CFU-L3 is synchronous to posedge clk. See §3.4.2. CFU-L3 includes the request's raw instruction. See §3.4.5.

Protocol:

1. Request transfer.
   a. Requester asserts CFU request signals req_* and asserts req_valid.
   b. Responder may assert req_ready.
   c. CFU receives CFU request on posedge clk when req_valid and req_ready are both asserted, per §3.4.3.
2. Custom function execution.
   a. CFU performs response status / error checking per §3.4.4.
   b. CFU performs a custom function of the operands and the selected state context.
   c. CFU 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) CFU asserts resp_status and resp_data and asserts resp_valid.
   b. Requester may assert resp_ready.
   c. Requester receives CFU response on posedge clk when resp_valid and resp_ready are both asserted, per §3.4.3.

3.8.4. CFU-L3 example

Figure 18 is an example waveform for four CFU-L3 CFU requests and responses, arising from executing four CF instructions `f0`-`f3`. (Assume `CFU_INSN_W=0`, no `req_insn`.) Each instruction issues a CFU request to the same CFU `u0`. Instructions `f0` and `f1` use state context `s0`; `f2` and `f3` use state context `s2`.

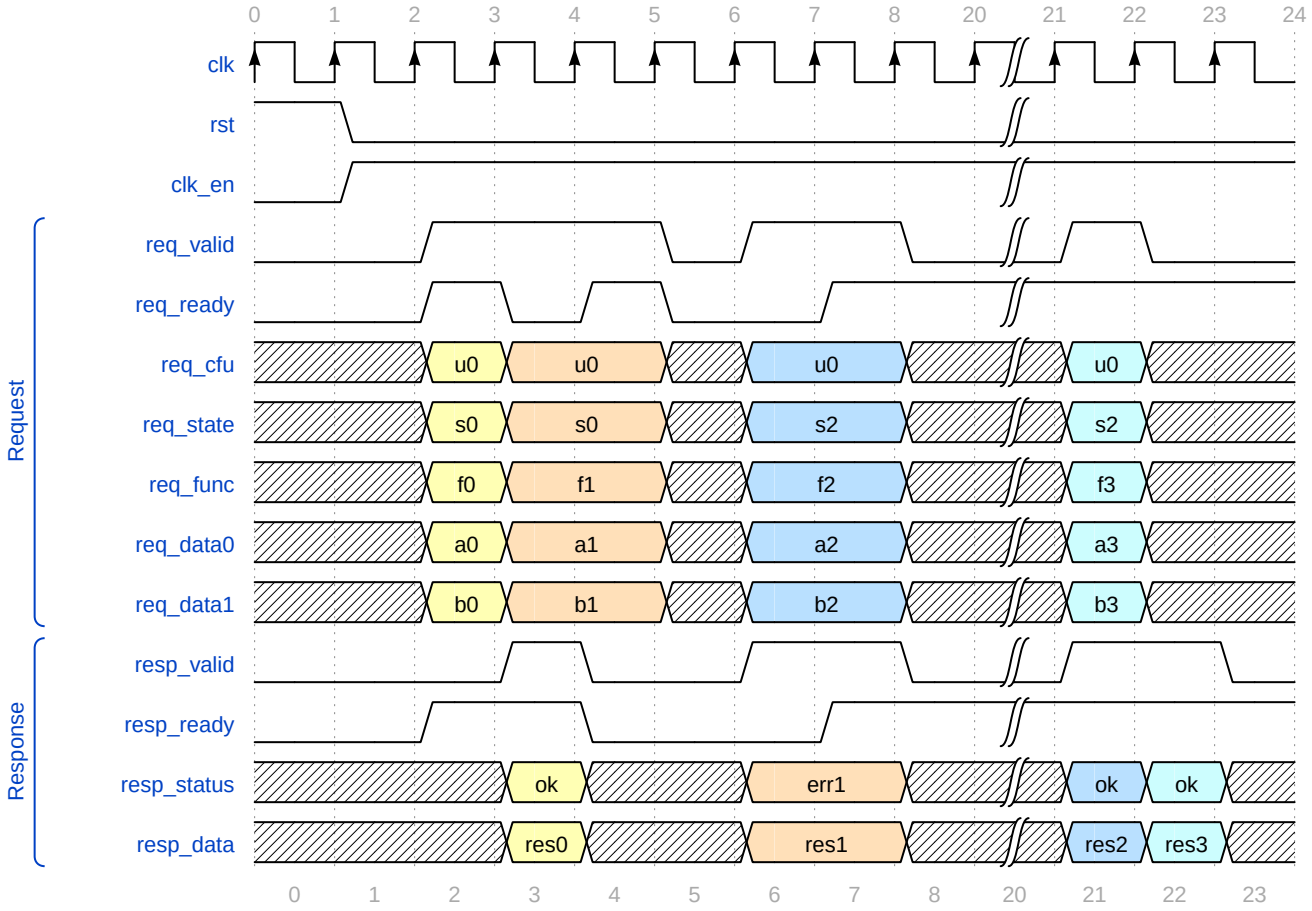

Figure 18. Example CFU-L3 signaling protocol waveform

The CFU 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 CFU until it asserts `req_ready` in cycle 4. The CFU 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 CFU until it asserts `req_ready` in cycle 7. The CFU responds to `f2` in cycle 21, a latency of 14 cycles.

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

3.9. CFU-L4 reordering CFU signaling

A reordering CFU computes a function of the CFU request and the specified state context, if any, updating the context, and sending a CFU 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.

CFU-L4 incorporates a request-response ID for the requester to correlate responses received to requests sent.

This CFU-LI feature level is motivated by past experience building floating point CFUs. 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 custom interface 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 CFU logic by reducing average CFU latency, enabling greater CFU parallelism, and reducing request blocking and response queueing.

When a custom interface 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 CFU implementations of the same custom interface.

3.9.1. CFU-L4 configuration parameters

Table 16. CFU-L4 configuration parameters

Parameter	Description
CFU_VERSION	CFU-LI version number
CFU_CFU_ID_MAX	number of CFUs at/below this CFU
CFU_STATE_ID_MAX	number of custom interface state contexts

For CFU_VERSION, CFU_CFU_ID_MAX, and CFU_STATE_ID_MAX, see §3.4.1.

3.9.2. CFU-L4 signals

Table 17. CFU-L4 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	CFU_REQ_ID_W	request REQ_ID
in	req_cfu	CFU_CFU_ID_W	request CFU_ID
in	req_state	CFU_STATE_ID_W	request STATE_ID
in	req_insn	CFU_INSN_W	request raw instruction
in	req_func	CFU_FUNC_ID_W	request CF_ID
in	req_data0	CFU_DATA_W	request operand data 0
in	req_data1	CFU_DATA_W	request operand data 1
out	resp_valid		response valid
in	resp_ready		response ready
out	resp_id	CFU_REQ_ID_W	response ID
out	resp_status	CFU_STATUS_W	response status
out	resp_data	CFU_DATA_W	response data

3.9.3. CFU-L4 signaling protocol

CFU-L4 is synchronous to posedge `clk`. See §3.4.2. CFU-L4 includes a request-response ID. See §3.4.6. CFU-L4 includes the request's raw instruction. See §3.4.5.

Protocol:

1. Request transfer.
   a. Requester asserts CFU request signals `req_*` (including new CFU-L4 signal `req_id`) and asserts `req_valid`.
   b. Responder may assert `req_ready`.
   c. CFU receives CFU request on posedge `clk` when `req_valid` and `req_ready` are both asserted, per §3.4.3
2. Custom function execution.
   a. CFU performs response status / error checking per §3.4.4.
   b. CFU performs a custom function of the operands and the selected state context.
   c. CFU 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) CFU asserts `resp_id`, `resp_status`, `resp_data` and asserts `resp_valid`.
   b. Requester may assert `resp_ready`.
   c. Requester receives CFU response on posedge `clk` when `resp_valid` and `resp_ready` are both asserted, per §3.4.3.

3.9.4. CFU-L4 example

Figure 19 is an example waveform for four CFU-L4 CFU requests, illustrating two different valid out-of-order response sequences, arising from executing four CF instructions `f0`-`f3`. (Assume `CFU_INSN_W=0`, no `req_insn`.) Each instruction issues a CFU request to the same CFU `u0`, but with various state contexts `s0`, `s1`, `s0` (again), and `s3`. This constrains the CFU to respond to request `f0` with state `s0`, before responding to subsequent request `f2` for state `s0`.

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

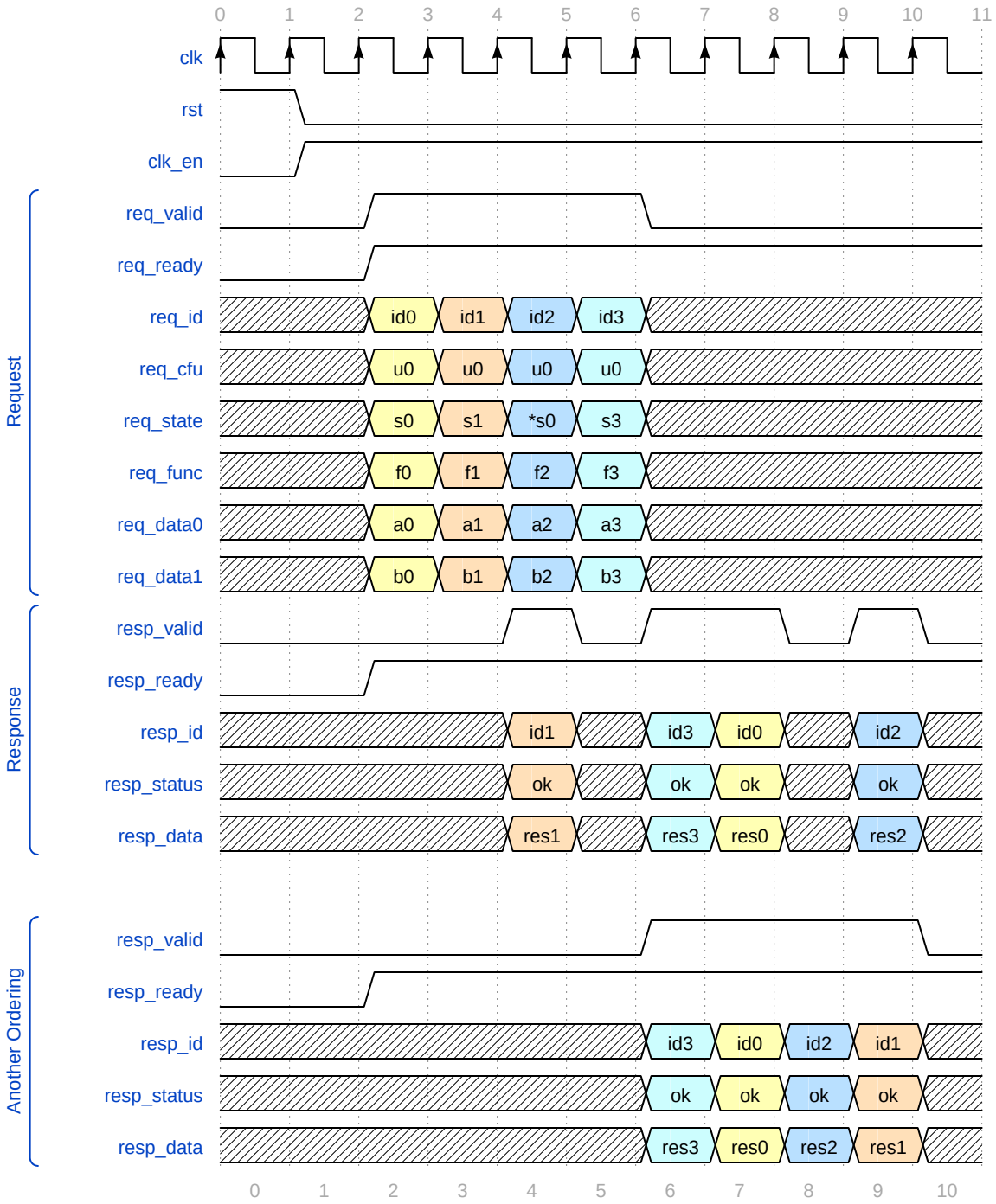

Figure 19. Example CFU-L4 signaling protocol waveform, with two of the possible response orderings

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

3.10. CFU feature level adapters

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

CFU-LI includes a set of configurable adapters to raise any CFU 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 flow control (always accepts requests)
- `Cvt03`, `Cvt13`, `Cvt23`: raise L0-L2 to L3: add response flow control (may suspend requests)

TODO: Write up the L4 adapters, which are just L3 adapters with a request-response ID FIFO.

3.10.1. `Cvt01`: raise CFU-L0 to CFU-L1

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

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

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

3.10.2. `Cvt02`: raise CFU-L0 to CFU-L2

A `Cvt02` adapter CFU implements CFU-L2, including its configuration parameters (§3.7.1), adapting L2 requests to and responses from a subordinate combinational L0 CFU. It implements request (non) flow control by permanently asserting `req_ready`. For each request received, it sends a response, asserting `resp_valid`, `resp_status`, and `resp_data` on next posedge `clk`.

3.10.3. `Cvt12`: raise CFU-L1 to CFU-L2

A `Cvt12` adapter CFU implements CFU-L2, including its configuration parameters (§3.7.1), plus `CFU_LATENCY` (§ 3.6.1), adapting L2 requests to and responses from a subordinate fixed latency L1 CFU. The `CFU_LATENCY` parameter specifies the latency of the *subordinate CFU*. The adapter implements request (non) flow control by permanently asserting `req_ready`. For each request received, it sends a response, asserting `resp_valid`, `resp_status`, and `resp_data` on posedge `clk` after no fewer than `CFU_LATENCY` cycles.

When `CFU_LATENCY=0`, the subordinate CFU response must be registered, so the adapter's response latency is one cycle.

3.10.4. `Cvt03`: raise CFU-L0 to CFU-L3

A `Cvt03` adapter CFU implements CFU-L3, including its configuration parameters (§3.8.1), adapting L3 requests to and responses from a subordinate combinational L0 CFU. The adapter has a fixed latency of one cycle — a response is sent one cycle after a request is received.

To avoid arbitrary CFU response queuing, yet keep signaling simple and frugal, the `Cvt03` adapter might negate `req_ready` on any cycle that it has a valid response waiting (asserting `resp_valid`) and the requester negates `resp_ready`.

3.10.5. `Cvt13`: raise CFU-L1 to CFU-L3

A `Cvt13` adapter CFU implements CFU-L3, including its configuration parameters (§3.8.1), plus `CFU_LATENCY` (§3.6.1), adapting L3 requests to and responses from a subordinate fixed latency L1 CFU.

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

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

3.10.6. `Cvt23`: raise CFU-L2 to CFU-L3

A `Cvt23` adapter CFU implements CFU-L3, including its configuration parameters (§3.8.1), adapting L3 requests to and responses from a subordinate variable latency L2 CFU.

In one implementation, sans response FIFO queueing, the adapter negates `req_ready` on any cycle that it has a valid response waiting (asserting `resp_valid`) and the requester negates `resp_ready`.

3.11. CFU-LI-compliant CPUs

A CFU-LI-compliant CPU implements RISC-V RV-I -Zicsr **-Zicfu** instruction set, sends CFU requests upon issuing CF instructions, and writes a destination register and CFU status CSR in response to CFU responses.

3.11.1. CPUs and CFU-LI feature levels

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

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

A pipelined in-order CPU might use CFU-L1 with a fixed latency CFU configured for (e.g.) 2 cycles latency. It might also use CFU-L2 with a variable latency CFU, stalling in WB-stage (writeback) if awaiting a slow CFU response.

An out-of-order completion CPU might use a CFU-L2 variable latency CFU or a -L3 elastic CFU, the latter if its WB-stage register file write arbiter cannot always accept a CFU response writeback on any cycle.

An OoO completion CPU, that handles reordered CFU responses, might use a CFU-L4 reordering CFU.

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

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

3.12. Example: CFU signaling in a composed system

Consider Figure 20, a system composed from two single-hart CPUs, two stateful CFUs, and a 2-input, 2-output Mux CFU. Fixed latency CFU_0 implements CFU-L1, configured with `CFU_LATENCY=1`. The CPUs, CFU_1, and `Mux22` use/implement CFU-L2. `Cvt12`, a CFU level converter, up-converts CFU_0 from CFU-L1 to CFU-L2.

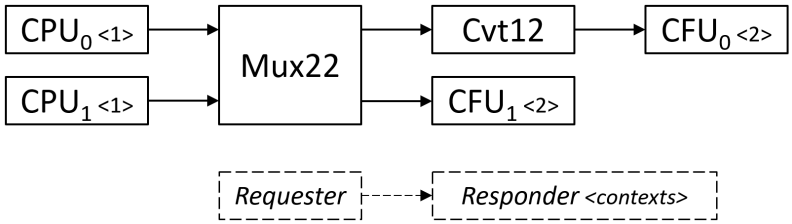

Figure 20. CFU-L2 system, with two CPUs, mux CFU, converter CFU, CFU_0 (L1), and CFU_1 (L2)

With one hart per CPU, the custom interfaces' CFUs are configured with two state contexts each (<2>).

Both CPU_0 and CPU_1 are configured to issue CF instructions mapping $CI\_ID_0 \rightarrow CFU\_ID{=}0 \rightarrow CFU_0$ and $CI\_ID_1 \rightarrow CFU\_ID{=}1 \rightarrow CFU_1$.

The exemplary 2x2 Mux CFU 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 mux consists of input ports (not registered), output port registers, an approximately fair output port arbiter, and a 2x2 channel crossbar. Each cycle, the mux 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_cfu` CFU_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 CFUs with different latencies, are always returned in order to the requester, as required by CFU-L2.

Downstream request routing is per the request inputs' `req_cfu` elements: CFU_ID=0 routes to the first output port and CFU_ID=1 routes to the second output port. The mux itself responds to requests with invalid CFU_IDs with a `CFU_ERROR_CFU` response.

For upstream response routing, the Mux incorporates, for each subordinate CFU, a FIFO queue that records the requester port ID that issued each request to that CFU. As each (in order) response from that CFU 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 interface unit latencies. Each CPU runs the same code (Listing 1):

1. Write `mcfu_selector` for CFU_ID=0 and STATE_ID=*HART_ID*, issue two CF instructions to CFU_0;
2. Write `mcfu_selector` for CFU_ID=1 and STATE_ID=*HART_ID*, issue two CF instructions to CFU_1;
3. Write `mcfu_selector` for CFU_ID=0 and STATE_ID=*HART_ID*, issue one CF instruction to CFU_0.

Listing 1. Issue stateful CF instructions `f0` and `f1` to CFU_0, `f2` and `f3` to CFU_1, and `f4` to CFU_0 again.

```
csrw mcfu_selector,x20  ; select CFU_ID=0 and STATE_ID=HART_ID
cfu_reg 0,x3,x1,x2      ; u0.f0
cfu_reg 1,x6,x5,x4      ; u0.f1

csrw mcfu_selector,x21  ; select CFU_ID=1 and STATE_ID=HART_ID
cfu_reg 2,x9,x7,x8      ; u1.f2
cfu_reg 3,x12,x11,x10   ; u1.f3

csrw mcfu_selector,x20  ; select CFU_ID=0 and STATE_ID=HART_ID again
cfu_reg 4,x15,x13,x14   ; u0.f4
```

Figure 21 is an example waveform executing Listing 1 near-simultaneously on the two CPUs of Figure 20.

(*1:u2<3>.f4* denotes CFU request #1 with CFU_ID=2 STATE_ID=3 CF_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 CFU-L2, request transfers occur when both `req_valid` and `req_ready` are asserted (§3.4.3), whereas response transfers occur when `resp_valid` is asserted.

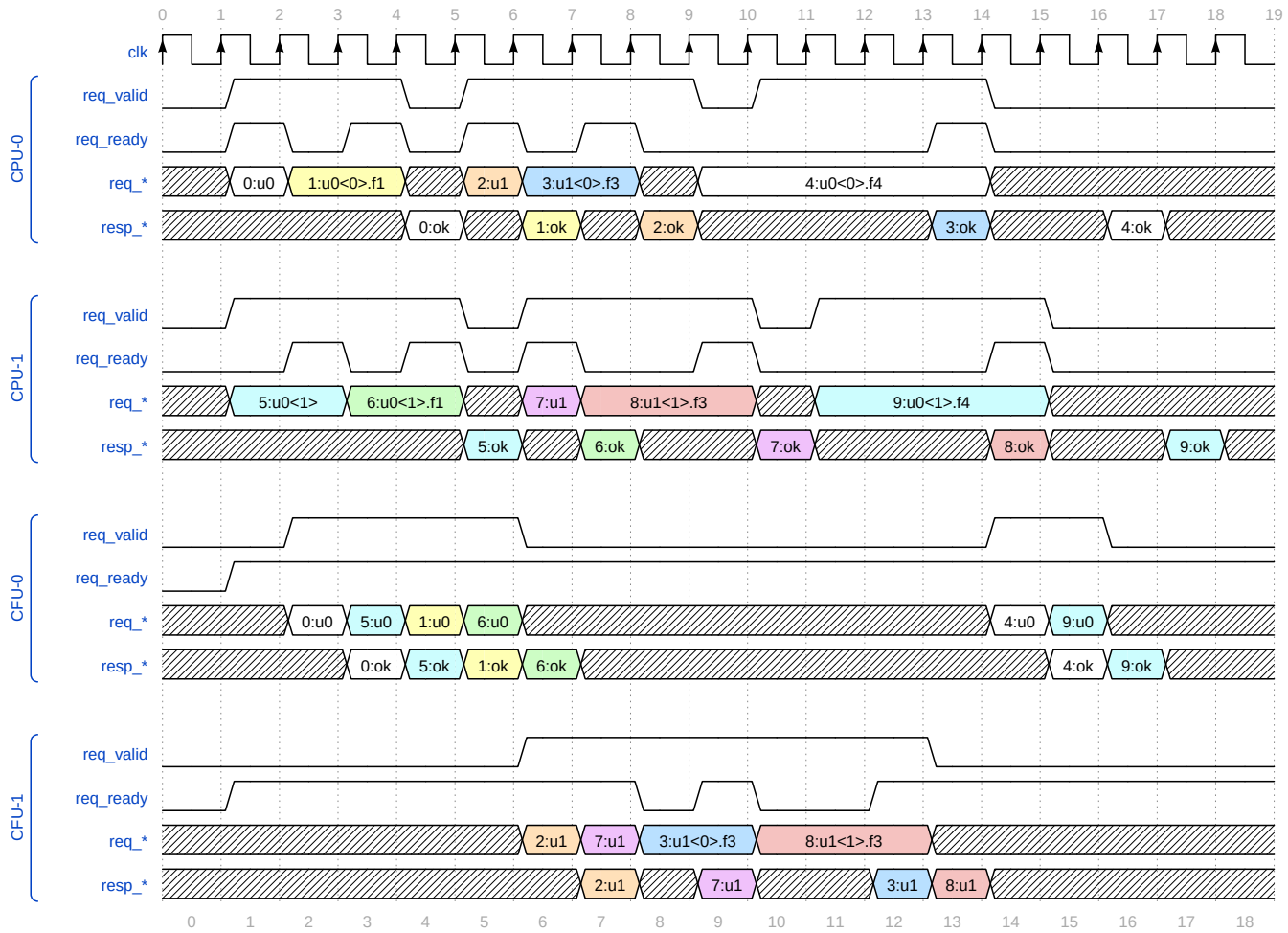

Figure 21. Example 2-input 2-output CFU-L2 Mux CFU signaling protocol waveform

Cycle-by-cycle:

0. Both CPUs CSR-write their hart's `mcfu_selector` registers, selecting CFU_ID=0=CFU_0, and their hart's STATE_ID.
   Both CPUs issue the first CF instruction (`f0`).

0. CPU_0 sends first CFU request (request #0): CFU_ID=0 STATE_ID=0 CF_ID=0, a.k.a. `0:u0<0>.f0`.
   CPU_1 sends first CFU request (request #5): CFU_ID=0 STATE_ID=1 CF_ID=0, a.k.a. `5:u0<1>.f0`.

1. CPU_0's first request, destined for CFU_0, wins arbitration for Mux output port 0.
   Mux asserts CPU_0's `req_ready` and negates CPU_1's `req_ready`.
   CPU_0's first request `0:u0<0>.f0` transfers to Mux.
   Mux sends CPU_0's first request to `Cvt12(`CFU_0`)`
   CPU_0 sends second CFU request: `1:u0<0>.f1`.

2. CPU_1's first request, destined for CFU_0, wins arbitration for Mux output port 0.
   Mux asserts CPU_1's `req_ready` and negates CPU_0's `req_ready`.
   CPU_1's first request `5:u0<1>.f0` transfers to Mux.
   Mux sends CPU_1's first request to `Cvt12(`CFU_0`)`.
   CPU_1 sends second CFU request: `6:u0<0>.f1`.
   CPU_0's first request `0:u0<0>.f0` transfers to CFU_0.
   CFU_0 executes `0:f0`, updates state `<0>`, sends response to Mux.

3. CPU_0 sends no CFU request this cycle, due to its second `csrw` execution cycle.
   CPU_0's second request `1:u0<0>.f1`, wins arbitration, transfers to Mux, is sent to `Cvt12(`CFU_0`)`.
   CPU_1's first request `5:u0<1>.f0` transfers to CFU_0, executes, updates `<1>`, sends response to Mux.
   CFU_0's response to CPU_0's first request transfers to Mux, is sent to CPU_0.

4. CPU_1 sends no CFU request this cycle, due to its second `csrw` execution cycle.
   CPU_1's second request `6:u0<0>.f1`, wins arbitration, transfers to Mux, is sent to `Cvt12(`CFU_0`)`.
   CPU_0's second request `1:u0<1>.f1` transfers to CFU_0, executes, updates `<0>`, sends response to Mux.
   CFU_0's response to CPU_1's first request transfers to Mux, is sent to CPU_1.
   CFU_0's response to CPU_0's first request transfers to CPU_0.

5. CPU_0 bubble in CFU request issue due to its second `csrw` execution cycle.
   CPU_1 sends third request `2:u1<1>.f2`, with CFU_ID=1, destined for CFU_1.
   CPU_0's third request `2:u1<0>.f2`, transfers to Mux, is sent to CFU_1.
   CPU_0 sends fourth request `3:u1<0>.f3`, with CFU_ID=1, destined for CFU_1.
   CPU_1's second request `6:u0<1>.f1` transfers to CFU_0, executes, updates `<1>`, sends response to Mux.
   CFU_0's response to CPU_0's second request transfers to Mux, is sent to CPU_0.
   CFU_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 Mux, is sent to CFU_1.
   CPU_1 sends fourth request `8:u1<0>.f3`, with CFU_ID=1, destined for CFU_1.
   CPU_0's third request `2:u1<0>.f2` transfers to CFU_1, executes, updates `<0>`, sends response to Mux.
   CFU_0's response to CPU_1's second request transfers to Mux, is sent to CPU_1.
   CFU_0's response to CPU_0's second request transfers to CPU_0.

7. CPU_0 sends no CFU request this cycle, due to its third `csrw` execution cycle.
   CPU_0's fourth request `3:u1<0>.f3` wins arbitration, transfers to Mux, is sent to CFU_1.
   CPU_1's third request `7:u1<1>.f2` transfers to CFU_1, begins execution.
   CFU_1's response to CPU_0's third request transfers to Mux, is sent to CPU_0.
   CFU_0's response to CPU_1's second request transfers to CPU_1.

8. CPU_1 sends no CFU request this cycle, due to its third `csrw` execution cycle.
   CPU_0 sends fifth request `4:u0<0>.f4`, with CFU_ID=0, destined for CFU_0.
   At CFU_1, CPU_1's third request `7:u1<0>.f2` completes execution, updates `<1>`, sends response to Mux.
   CFU_1's response to CPU_0's third request transfers to CPU_0.

9. CPU_0's fifth CFU request is *ineligible* to transfer because CPU_0 has pending requests to CFU_1. It becomes eligible at cycle 13.
   CPU_1's fourth request `8:u1<0>.f3` transfers to Mux, is sent to CFU_1.
   CPU_0's fourth request `3:u1<0>.f3` transfers to CFU_1, begins execution.
   CFU_1's response to CPU_1's third request transfers to Mux, is sent to CPU_1.
10. CPU_1 sends fifth request `9:u0<1>.f4`, with CFU_ID=0, destined for CFU_0.
    CPU_0's fourth CFU request `3:u1<0>.f3` continues execution.
    CFU_1's response to CPU_1's third request transfers CPU_1.
11. CPU_1's fifth CFU request is *ineligible* to transfer because CPU_1 has pending requests to CFU_1. It becomes eligible at cycle 14.
    CPU_0's fourth CFU request `3:u1<0>.f3` completes execution, updates `<0>`, sends response to Mux.
12. CPU_1's fourth request `8:u1<1>.f3` transfers to CFU_1, executes, updates `<1>`, sends response to Mux.
    CFU_1's response to CPU_0's fourth request transfers to Mux, is sent to CPU_0.
13. CFU_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 Mux, is sent to CFU_0.
14. CFU_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 Mux, is sent to CFU_1.
    CPU_0's fifth request `4:u0<0>.f4` transfers to CFU_0, executes, updates `<0>`, sends response to Mux.
15. CPU_1's fifth request `9:u0<1>.f4` transfers to CFU_0, executes, updates `<1>`, sends response to Mux.
    CFU_0's response to CPU_0's fifth request transfers to Mux, is sent to CPU_0.
16. CFU_0's response to CPU_1's fifth request transfers to Mux, is sent to CPU_1.
    CFU_0's response to CPU_0's fifth request transfers to CPU_0.
17. CFU_0's response to CPU_1's fifth request transfers to CPU_1.

3.13. Composing CFUs with AXI4-Streams

In some configured systems, preexisting infrastructure components that implement AXI4-Stream protocol may be used to help compose CPUs and CFUs. A fully flow controlled CFU-LI -L3 or -L4 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 CFU requests, via CFU-L3-to-AXI-S bridge, AXI-S-to-NOC bridge, Versal NOC, NOC-to-AXI-S bridge, AXI-S-to-CFU-L3 bridge, to a CFU at the far corner of the FPGA fabric, later transferring CFU responses back to the distant CPU by the same means.

Table 18 presents a recommended canonical mapping between CFU-LI signals and the two AXI-S streams.

Table 18. Recommended mapping between CFU-L3/-L4 and request/response AXI4-Streams

Dir	CFU-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`	`CFU_REQ_ID_W`	`reqs_tid` *or* `reqs_tdest`
in	`req_cfu`	`CFU_CFU_ID_W`	`reqs_tuser` *or* `reqs_tdest`

Dir	CFU-LI Port	Width	AXI-S Port
in	req_state	CFU_STATE_ID_W	reqs_tuser
in	req_func	CFU_FUNC_ID_W	reqs_tuser
in	req_insn	CFU_INSN_W	reqs_tuser
in	req_data0	CFU_DATA_W	reqs_tdata
in	req_data1	CFU_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	CFU_REQ_ID_W	resps_tid *or* resps_tdest
out	resp_data	CFU_DATA_W	resps_tdata
out	resp_status	CFU_STATUS_W	resps_tuser
out	-		resps_tlast *optional*
out	-	*	resps_tstrb *optional*
out	-	*	resps_tkeep *optional*

When several CFU-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_cfu };
reqs_tdata = { req_data1,req_data0 };
```

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

4. CFU Metadata (CFU-MD)

To help automate system composition, each composable hardware core (each CPU and CFU) 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]
```

4.1. CFU Metadata

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

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

Listing 2. CFU metadata format

```
cfu_name: string
cfu_li:
    feature_level: scalar                   # required.  allowed: 0-4
    state_id_max: scalar | list | 'range'   # level:any. default: any. 0 => stateless
    req_id_w: scalar | list | 'range'       # level:2+.  default: 0
    cfu_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 CFUs and CPUs here. Perhaps a GUID, perhaps a URL.

Do we need to specify here which CI_IDs the CFU implements?

4.2. Example CFU metadata

Listing 3 is example CFU metadata for a CFU-L1 CFU 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 CFU metadata (CFU-L1)

```
cfu_name: bobs_bnn_cfu
cfu_li:
    feature_level: 1
    state_id_max: 1          # only supports 1 state context
    req_id_w:                # any req_id is fine
    cfu_id_w: 0              # no req_cfu
    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
```

4.3. CPU Metadata

As described in §3.11, CPUs, as CFU requesters, use specific CFU-LI feature levels. As with CFUs, CPUs use CFU metadata to override configuration parameter defaults, in this case to define what the CPU requires or accepts of its CFU (which is, generally, the root of the DAG of CFUs).

Listing 4. CPU metadata format

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

4.4. Example CPU metadata

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

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

```
cpu_name: carols_simple_scalar_cpu
cfu_li:
    feature_level: 0    # L0 combinational CFUs only
    state_id_max:       # L0: n/a
    req_id_w:           # L0: n/a
    cfu_id_w:           # supports arbitrary CFU_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
```

4.5. System manifest

TODO

Consider CI library metadata too. "I may use use this subset { CF_IDs } of the CF_IDs of interface CI_ID."

5. TODO

Todo:

- Chapter on CI Runtime (runtime) API
- How CI and CFU versioning works; how CFU-LI versioning works

5.1. Open design problems (post 1.0)

- Developing, running accelerated libraries on systems where there is no custom interface / CFU implementation.
- Developer tooling recommendations for disassembly, debugging, profiling, perf monitoring.

5.2. Other CFU-like mechanisms

- Intel Nios II Custom Instruction User Guide*
- Xilinx LogiCORE IP Fast Simplex Link Bus*
- Xilinx Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link*
- Rocket Custom Coprocessor (RoCC) Extension*
- PicoRV "Pico Co-Processor Extension" (PCPI)
- Core-V *cv-x-if* (extension interface) (Tim Callahan's notes)

5.3. Example: a stateful extended precision ALU

Here write up a fully worked example of a CFU-L1 extended precision ALU, its custom interface, CFU, and library code. (A follow up to a discussion on the RISC-V mailing list that didn't go anywhere.)

5.4. Cost model

Here write up a brief estimate of the FPGA area overhead of various -Zicfu and CFU-LI mechanisms and behaviors.

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