OpenHW Group Specification: Core-V eXtension interface (CV-X-IF) - Development

Release v1.0.0-rc.3-dev.1

OpenHW Group

Apr 18, 2024

CONTENTS:

CHAPTER

ONE

ACKNOWLEDGEMENTS

The specification has in part been supported by the TRISTAN project.

The TRISTAN project, nr. 101095947 is supported by Chips Joint Undertaking (CHIPS-JU) and its members Austria, Belgium, Bulgaria, Croatia, Cyprus, Czechia, Germany, Denmark, Estonia, Greece, Spain, Finland, France, Hungary, Ireland, Israel, Iceland, Italy, Lithuania, Luxembourg, Latvia, Malta, Netherlands, Norway, Poland, Portugal, Romania, Sweden, Slovenia, Slovakia, Turkey.

CHAPTER TWO

CONTRIBUTORS

The specification and its repository have received contributions from several authors:

- christian-herber-nxp

  110 contributions
- davideschiavone

  54 contributions
- Silabs-ArjanB

  50 contributions
- ganoam

  28 contributions
- DBees

  6 contributions
- davidmallasen

  5 contributions
- ASintzoff

  5 contributions
- MikeOpenHWGroup

  4 contributions
- moimfeld

  2 contributions
- michael-platzer

  1 contribution

CHAPTER

THREE

CHANGELOG

3.1 v1.0.0-rc.2: Second Release Candidate (post public review)

Released on 2024-04-15 - GitHub

3.2 v1.0.0-rc.1: First release candidate

Released on 2024-02-16 - GitHub

3.3 v0.2.0: Reworked specification

Released on 2024-02-16 - GitHub

3.4 v0.1.0: Initial draft

Released on 2024-02-13 - GitHub

CHAPTER FOUR

INTRODUCTION

The `Core-V eXtension interface`, also called `CV-X-IF`, is an interface aimed at extending a *CPU* with (custom or standardized) instructions implemented in a coprocessor.

It can be used to implement standard RISC-V extensions as for example B (Bit Manipulation), M (Integer Multiplication and Division), F (Single-Precision Floating Point) and D (Double-Precision Floating Point). It can also be used to implement custom extensions. Extensions implemented on the interface are unprivileged, i.e. implementing privileged extensions like H (Hypervisor) is not supported.

The goal of `CV-X-IF` is to enable the design and verification of instruction extensions in a coprocessor in a standardized manner without the need to modify the *CPU* itself. Having a common interface allows designers of RISC-V *CPUs* to reuse existing co-processor and vice versa. Please note that the *CPU* and coprocessor can have different license models. For example, the coprocessor could be closed source, connected to an open-source *CPU*.

4.1 History

The idea of an extension interface originated from the **PULP Project** at ETH Zurich and University of Bologna, where it was used to decouple the floating-point unit and the CPU design. The first version of this interface was called `apu interface`, and it was implemented in the **CV32E40P** to communicate with the **CVFPU** coprocessor. However, this interface was tightly coupled with the *CPU* pipeline, which meant that any other new coprocessor extension had to modify the *CPU* pipeline and decoder. Moreover, it was designed for a specific use-case. Later, the PULP team developed a more advanced interface for the **CVA6** project, which could handle more complex scenarios required by the **ARA** vector machine. This interface was further refined in the **Snitch** project, where it was made more modular and independent from the pipeline, requiring only minimal changes to the decoder of the *CPU*. The aim of `CV-X-IF` within the OpenHW Group is to take this interface to the next level and eliminate all dependencies between the *CPU* and the coprocessor. The interface is not only agnostic from the decoder and pipeline perspective, but also from the license and codebase standpoint, with the goal of becoming the standard interface that will enable wide reuse of RISC-V IPs. The first CPU implementing such interface is the **CV32E40X**, which can be found at https://github.com/openhwgroup/cv32e40x. The interface was also added as an option to **CVA6**, which can be found at https://github.com/openhwgroup/cva6.

4.2 License

4.3 Standards Compliance

The `CV-X-IF` specification depends on the unprivileged [RISC-V-UNPRIV] and privileged [RISC-V-PRIV] RISC-V specification.

4.4 Glossary

clk
: Clock signal

ISA
: Instruction set architecture

CPU
: Central processing unit

ALU
: Arithmetic logic unit

CSR
: Control and status register

GPR
: General purpose register

PMP
: Physical memory protection

PMA
: Physical memory attributes

MMU
: Memory management unit

NMI
: Non-maskable interrupt

UVM
: Universal Verification Methodology

RTL
: Register transfer language

ECS
: Extension Context Status

CHAPTER

FIVE

EXTENSION INTERFACE

The eXtension interface enables extending the *CPU* with (custom or standardized) instructions without the need to change the *RTL* of the *CPU* itself. An extension can be provided in a separate module external to the *CPU* and is integrated at system level by connecting it to the eXtension interface.

The eXtension interface provides low latency (tightly integrated) read and write access to the *CPU* register file. All opcodes which are not used (i.e. considered to be invalid) by the *CPU* can be used for extensions. It is recommended however that custom instructions do not use opcodes that are reserved/used by RISC-V International.

The eXtension interface enables extension of the *CPU* with:

- Custom *ALU* type instructions.
- Custom *CSRs* and related instructions.

Control-Transfer type instructions (e.g. branches and jumps) are not supported via the eXtension interface.

5.1 CV-X-IF

The terminology `eXtension interface` and `CV-X-IF` are used interchangeably.

5.2 Parameters

The CV-X-IF specification contains two kinds of parameters. The first kind of parameters is configured for the coprocessor. Not all possible values of parameter might be supported by the *CPU*, in which case it determines the legal values.

The second kind of parameter is a system parameter, i.e. it is determined based on the configuration of the *CPU* and the coprocessor. This includes `X_ID_WIDTH` and `X_HARTID_WIDTH`.

Table 5.1: Interface parameters

Name	Type/Range	Default	Description
`X_NUM_RS`	int unsigned (2..3)	2	Number of register file read ports that can be used by the eXtension interface. Legal values are determined by the *CPU*.
`X_ID_WIDTH`	int unsigned (3..32)	4	Identification (`id`) width for the eXtension interface.
`X_RFR_WIDTH`	int unsigned (32, 64)	32	Register file read access width for the eXtension interface. Legal values are determined by the *CPU*. Must be at least XLEN. If XLEN = 32, then the legal values are 32 and 64 (e.g. for RV32P). If XLEN = 64, then the legal value is (only) 64.
`X_RFW_WIDTH`	int unsigned (32, 64)	32	Register file write access width for the eXtension interface. Legal values are determined by the *CPU*. Must be at least XLEN. If XLEN = 32, then the legal values are 32 and 64 (e.g. for RV32D). If XLEN = 64, then the legal value is (only) 64.
`X_NUM_HARTS`	int unsigned (1..2^MXLEN)	1	Number of harts (hardware threads) associated with the interface. Legal values are determined by the *CPU*.
`X_HARTID_WIDTH`	int unsigned (1..MXLEN)	1	Width of `hartid` signals. Must be at least 1. Limited by the RISC-V privileged specification to MXLEN. Legal values are determined by the *CPU*.
`X_MISA`	logic [25:0]	32'b0	MISA extensions implemented on the eXtension interface. Legal values are determined by the *CPU*.
`X_DUALREAD`	int unsigned (0..3)	0	Is dual read supported? 0: No, 1: Yes, for `rs1`, 2: Yes, for `rs1` - `rs2`, 3: Yes, for `rs1` - `rs3`. Legal values are determined by the *CPU*.
`X_DUALWRITE`	int unsigned (0..1)	0	Is dual write supported? 0: No, 1: Yes. Legal values are determined by the *CPU*.
`X_ISSUE_REGISTER_SPLIT`	int unsigned (0..1)	0	Are the issue interface and register interface split? 0: No, 1: Yes. Legal values are determined by the *CPU*. If 1, registers are provided after the issue of the instruction. If 0, registers are provided at the same time as issue.

The *CPU* shall set the `misa.Extensions` field to a value that is the result of an or operation of its own Extensions and the `X_MISA` parameter. Not all bits of `misa.Extensions` will be legal for a coprocessor to set, e.g. if this extension is already implemented in the *CPU* or if it is an extension not possible to implement as part of a coprocessor like privileged extensions.

Note: A *CPU* shall clearly document which `X_MISA` values it can support and there is no requirement that a *CPU* can support all possible `X_MISA` values. For example, if a *CPU* only supports machine mode, then it is not reasonable to expect that the *CPU* will additionally support user mode by just setting the `X_MISA[20]` (`U` bit) to 1.

Additionally, the following type definitions are defined to improve readability of the specification and ensure consistency between the interfaces:

Table 5.2: Interface type definitions

Name	Definition	Description
`readregflags_t`	logic [X_NUM_RS+X_DUALREAD-1:0]	Vector with a flag per possible source register. This depends upon the number of read ports and their ability to read register pairs. The bit positions map to registers as follows: Low indices correspond to low operand numbers, and the even part of the pair has a lower index than the odd one.
`writeregflags_t`	logic [X_DUALWRITE:0]	Bit vector indicating destination registers for write back. The width depends on the ability to perform dual write. If `X_DUALWRITE` = 0, this signal is a single bit. Bit 1 may only be set when bit 0 is also set. In this case, the vector indicates that a register pair is used.
`id_t`	logic [X_ID_WIDTH-1:0]	Identification of the offloaded instruction. See *Identification* for details on the identifiers
`hartid_t`	logic [X_HARTID_WIDTH-1:0]	Identification of the hart offloading the instruction. Only relevant in multi-hart systems. Hart IDs are not required to to be numbered continuously. The hart ID would usually correspond to `mhartid`, but it is not required to do so.

5.3 Major features

The major features of CV-X-IF are:

- Minimal requirements on extension instruction encoding.

  If an extension instruction relies on reading from or writing to the *CPU*'s general purpose register file, then the standard RISC-V bitfield locations for rs1, rs2, rs3, rd as used for non-compressed instructions ([RISC-V-UNPRIV]) must be used. Bitfields for unused read or write operands can be fully repurposed. Extension instructions can either use the compressed or uncompressed instruction format. For offloading compressed instructions the coprocessor must provide the *CPU* with the related non-compressed instructions.

- Support for dual write-back instructions (optional, based on `X_DUALWRITE`).

  CV-X-IF optionally supports implementation of (custom or standardized) *ISA* extensions mandating dual register file write-backs. Dual write-back is supported for even-odd register pairs (`Xn` and `Xn+1` with `n` being an even number extracted from instruction bits `[11:7]`).

  Dual register file write-back is only supported for `XLEN` = 32.

- Support for dual read instructions (per source operand) (optional, based on `X_DUALREAD`).

  CV-X-IF optionally supports implementation of (custom or standardized) *ISA* extensions mandating dual register file reads. Dual read is supported for even-odd register pairs. Dual read can therefore provide up to six 32-bit operands per instruction.

  When a dual read is performed with `n` = 0, the entire operand is 0, i.e. `x1` shall not need to be accessed by the *CPU*.

  Dual register file read is only supported for XLEN = 32.

- Support for ternary operations.

CV-X-IF optionally supports *ISA* extensions implementing instructions which use three source operands. RISC-V [RISC-V-UNPRIV] can implement ternary operations using the R-type instruction format (using `rd` as `rs3`) or with the R4-type instruction format.

- Support for instruction speculation.

  CV-X-IF indicates whether offloaded instructions are allowed to be committed (or should be killed).

Note: The interface does not provide a mechanism for providing and synchronizing the Extension Context Status (*ECS*, see [RISC-V-PRIV]). *ECS* might be needed if an extension has context that needs to be switched upon a task switch. Ensuring that the behavior of the overall system is compliant to [RISC-V-PRIV] is the responsibility of an integrator. It is the intention that future versions of this specification provide a general mechanism to deal with *ECS*.

CV-X-IF consists of the following interfaces:

- **Compressed interface**. Signaling of compressed instruction to be offloaded.
- **Issue (request/response) interface**. Signaling of the uncompressed instruction to be offloaded.
- **Register interface**. Signaling of *GPRs* and *CSRs*.
- **Commit interface**. Signaling of control signals related to whether instructions can be committed or should be killed.
- **Result interface**. Signaling of the instruction result(s).

5.4 Operating principle

CPU will attempt to offload every (compressed or non-compressed) instruction that it does not recognize as a legal instruction itself. In case of a compressed instruction the coprocessor must first provide the *CPU* with a matching un-compressed (i.e. 32-bit) instruction using the compressed interface. This non-compressed instruction is then attempted for offload via the issue interface.

Offloading of the (non-compressed, 32-bit) instructions happens via the issue interface. The external coprocessor can decide to accept or reject the instruction offload. In case of acceptation the coprocessor will further handle the instruction. In case of rejection the *CPU* will raise an illegal instruction exception. The *CPU* provides the required register file operand(s) to the coprocessor via the register interface. If an offloaded instruction uses any of the register file sources `rs1`, `rs2`, then these are always encoded in instruction bits `[19:15]` and `[24:20]`, respectively. If an offloaded instruction uses the register file source `rs3`, then these are encoded in instruction bits `[31:27]` if the instruction uses one of the major opcodes instruction uses the major opcodes MADD, MSUB, NMSUB, or NMADD (R4-type). Otherwise, `rs3` is expected to be encoded in bits `[11:7]`.

Note: The fused multiply add instructions of the floating point unit make use of the R4 instruction format. As this format consumes significant encoding space, other standard and custom extensions are expected to follow the R-type encoding, multiplexing `rd` and `rs3`.

The coprocessor only needs to wait for the register file operands that a specific instruction actually uses. The coprocessor informs the core to which register(s) in the register file it will write-back. The *CPU* uses this information to track data dependencies between instructions.

Offloaded instructions are speculative; *CPU* has not necessarily committed to them yet and might decide to kill them (e.g. because they are in the shadow of a taken branch or because they are flushed due to an exception in an earlier instruction). Via the commit interface the *CPU* will inform the coprocessor about whether an offloaded instruction will either need to be killed or whether the *CPU* will guarantee that the instruction is no longer speculative and is allowed to be committed.

The final result of an accepted offloaded instruction can be written back into the coprocessor itself or into the *CPU*'s register file. Either way, the result interface is used to signal to the *CPU* that the instruction has completed. Apart from a possible write-back into the register file, the result interface transaction is for example used in the *CPU* to increment the `minstret` *CSR*, to implement the fence instructions and to judge if instructions before a `WFI` instruction have fully completed (so that sleep mode can be entered if needed).

In short: From a functional perspective it should not matter whether an instruction is handled inside the *CPU* or inside a coprocessor. In both cases the instructions need to obey the same instruction dependency rules, memory consistency rules, load/store address checks, fences, etc.

5.5 Interfaces

This section describes the interfaces of CV-X-IF. Port directions are described as seen from the perspective of the *CPU*. The coprocessor will have opposite pin directions. Stated signals names are not mandatory, but it is highly recommended to at least include the stated names as part of actual signal names. It is for example allowed to add prefixes and/or postfixes (e.g. `x_` prefix or `_i`, `_o` postfixes) or to use different capitalization. A name mapping should be provided if non obvious renaming is applied.

5.5.1 Clocking and Signal Stability

The interfaces are required to be synchronous to a common clock (`clk`). The signals of the interface are sampled on the positive edge of `clk`.

When stability of signal is referred to in the specification of the interface transactions the following definition is followed. A signal is considered stable, if to consecutive samples of the signal have the same value. A signal's value may change between the samples and still be considered stable.

5.5.2 Identification

Most interfaces of CV-X-IF use a signal called `id`, which serves as a unique identification number for offloaded instructions. The same `id` value shall be used for all transaction packets on all interfaces that logically relate to the same instruction. An `id` value can be reused after an earlier instruction related to the same `id` value is no longer consider in-flight. The `id` values for in-flight offloaded instructions are required to be unique. The `id` values are required to be incremental from one issue transaction to the next. The increment may be greater than one. If the next `id` would be greater than the maximum value (`2**X_ID_WIDTH - 1`), the value of `id` wraps. A new `id` value is not allowed to be greater than the oldest in-flight instruction, if a wrap has occurred since the oldest in-flight instruction was issued. If the oldest in-flight instruction is id_o, and the newest is id_n, then the next instruction with id_{n+1} must satisfy the following conditions:

$$
\begin{aligned}
id_{n+1} > id_n \text{ or } id_{n+1} < id_o, \text{ if } id_n > id_o \\
id_{n+1} > id_n \text{ and } id_{n+1} < id_o, \text{ if } id_n < id_o
\end{aligned}
$$

The first condition applying to cases where the id_n has not wrapped since the oldest in-flight instruction was issued, and the second where one wrap occurred between id_o and id_n. The coprocessor is not required to check the validity of `id` values under these constraints. This has to be guaranteed by design of the CPU.

Note: IDs are not required to be incremental to support scenarios, in which a coprocessor does not see the entire instruction stream. This can be e.g. because offloaded instructions are routed towards different coprocessors.

To make sure feasible `id` values are available, `X_ID_WIDTH` needs to be sufficiently large. This can be achieved by calculating the maximum `id` increase during the lifetime of the longest executing instruction.

`id` values can only be introduced by the issue interface.

An `id` becomes in-flight in the first cycle that `issue_valid` is 1 for that `id`.

An `id` ends being in-flight when one of the following scenarios apply:

- the corresponding issue request transaction is retracted.
- the corresponding issue request transaction is not accepted and the corresponding commit handshake has been performed.
- the corresponding result transaction has been performed.

For the purpose of relative identification, an instruction is considered to be preceding another instruction, if it was accepted in an issue transaction at an earlier time. The other instruction is thus succeeding the earlier one.

5.5.3 Multiple coprocessors

This specification defines a point-to-point connection between a *CPU* and a coprocessor, that is defined in a way that facilitates the integration of multiple coprocessors. The combined interface of the coprocessors must adhere to this specification and thus must behave like a single coprocessor from the *CPU* point of view. Any implementation is correct, if the *CPU* is not able to determine that multiple coprocessors are connected. For recommendations, please refer to *Recommendations for implementing multiple coprocessors on a shared interface*

5.5.4 Multiple Harts

The interface can be used in systems with multiple harts (hardware threads). This includes scenarios with multiple *CPUs* and multi-threaded implementations of *CPUs*. RISC-V distinguishes between harts using `hartid`, which we also introduce to the interface. It is required to identify the source of the offloaded instruction, as multiple harts might be able to offload via a shared interface. No duplicates of the combination of `hartid` and `id` may be in flight at any time within one instance of the interface. Any state within the coprocessor (e.g. custom *CSRs*) must be duplicated according to the number of harts (indicated by the `X_NUM_HARTS` parameter). Execution units may be shared among threads of the coprocessor, and conflicts around such resources must be managed by the coprocessor.

Note: The interface can be used in scenarios where the *CPU* is superscalar, i.e. it can issue more than one instruction per cycle. In such scenarios, the coprocessor is usually required to also be able to accept more than one instruction per cycle. Our expectation is that implementers will duplicate the interface according to the issue width.

5.5.5 Compressed interface

Table 5.3 describes the compressed interface signals.

Table 5.3: Compressed interface signals

Signal	Type	Direction (*CPU*)	Description
compressed_valid	logic	output	Compressed request valid. Request to uncompress a compressed instruction.
compressed_ready	logic	input	Compressed request ready. The transactions signaled via compressed_req and compressed_resp are accepted when compressed_valid and compressed_ready are both 1.
compressed_req	x_compressed_req_t	output	Compressed request packet.
compressed_resp	x_compressed_resp_t	input	Compressed response packet.

Table 5.4 describes the x_compressed_req_t type.

Table 5.4: Compressed request type

Signal	Type	Description
instr	logic [15:0]	Offloaded compressed instruction.
hartid	*hartid_t*	Identification of the hart offloading the instruction.

The instr[15:0] signal is used to signal compressed instructions that are considered illegal by *CPU* itself. A coprocessor can provide an uncompressed instruction in response to receiving this.

Note: It is not required for a *CPU* to ensure that the offloaded instruction is a valid 16-bit encoding.

A compressed request transaction is defined as the combination of all compressed_req signals during which compressed_valid is 1 and compressed_req remains unchanged. A *CPU* is allowed to retract its compressed request transaction before it is accepted with compressed_ready = 1 and it can do so in the following ways:

- Set compressed_valid = 0.
- Keep compressed_valid = 1, but change any of the signals in compressed_req.

The signals in compressed_req are valid when compressed_valid is 1. These signals remain stable during a compressed request transaction.

Table 5.5 describes the x_compressed_resp_t type.

Table 5.5: Compressed response type

Signal	Type	Description
instr	logic [31:0]	Uncompressed instruction.
accept	logic	Is the offloaded compressed instruction (id) accepted by the coprocessor?

The signals in compressed_resp are valid when compressed_valid and compressed_ready are both 1. There are no stability requirements.

The *CPU* will attempt to offload every compressed instruction that it does not recognize as a legal instruction itself. A *CPU* might also attempt to offload compressed instructions that it does recognize as legal instructions itself.

A coprocessor may only accept valid 16-bit instructions, i.e. bits [1:0] must not be binary 11.

The *CPU* shall cause an illegal instruction fault when attempting to execute (commit) an instruction that:

- is considered to be valid by the *CPU* and accepted by the coprocessor (`accept` = 1).
- is considered neither to be valid by the *CPU* nor accepted by the coprocessor (`accept` = 0).

The `accept` signal of the *compressed* interface merely indicates that the coprocessor accepts the compressed instruction as an instruction that it implements and translates into its uncompressed counterpart. Typically an accepted transaction over the compressed interface will be followed by a corresponding transaction over the issue interface, but there is no requirement on the *CPU* to do so (as the instructions offloaded over the compressed interface and issue interface are allowed to be speculative). Only when an `accept` is signaled over the *issue* interface, then an instruction is considered *accepted for offload*.

Explicitly, the coprocessor shall not execute the instruction after receiving it via the compressed interface.

The coprocessor shall not take the `mstatus` based extension context status (see ([RISC-V-PRIV])) into account when generating the `accept` signal on its *compressed* interface (but it shall take it into account when generating the `accept` signal on its *issue* interface).

5.5.6 Issue interface

Table 5.6 describes the issue interface signals.

Table 5.6: Issue interface signals

Signal	Type	Direction (*CPU*)	Description
`issue_valid`	logic	output	Issue request valid. Indicates that *CPU* wants to offload an instruction.
`issue_ready`	logic	input	Issue request ready. The transaction signaled via `issue_req` and `issue_resp` is accepted when `issue_valid` and `issue_ready` are both 1.
`issue_req`	x_issue_req_t	output	Issue request packet.
`issue_resp`	x_issue_resp_t	input	Issue response packet.

Table 5.7 describes the `x_issue_req_t` type.

Table 5.7: Issue request type

Signal	Type	Description
`instr`	logic [31:0]	Offloaded instruction.
`hartid`	*hartid_t*	Identification of the hart offloading the instruction.
`id`	*id_t*	Identification of the offloaded instruction.

An issue request transaction is defined as the combination of all `issue_req` signals during which `issue_valid` is 1, and the `id` and `hartid` remain unchanged. A *CPU* is allowed to retract its issue request transaction before it is accepted with `issue_ready` = 1 and it can do so in the following ways:

- Set `issue_valid` = 0.
- Keep `issue_valid` = 1, but change the `id` or `hartid` signal (and if desired change the other signals in `issue_req`).

The `instr`, `hartid`, and `id` signals are valid when `issue_valid` is 1. The `instr` signal remains stable during an issue request transaction.

Table 5.8 describes the x_issue_resp_t type.

Table 5.8: Issue response type

Signal	Type	Description
accept	logic	Is the offloaded instruction (id and hartid) accepted (1) by the coprocessor or rejected (0)?
writeback	*writeregflags_t*	Will the coprocessor perform a write-back in the *CPU* to rd? Write-back to x0 or the x0, x1 pair is allowed by the coprocessor, but will be ignored by the *CPU*. Write-back to a register pair is only allowed if X_DUALWRITE = 1 and instruction bits [11:7] are even.
register_read	*readregflags_t*	Will the coprocessor perform require specific registers to be read? A coprocessor may only request an odd register of a pair, if it also requests the even register of a pair.

The *CPU* shall attempt to offload instructions via the issue interface for the following two main scenarios:

- The instruction is originally non-compressed and it is not recognized as a valid instruction by the *CPU*'s non-compressed instruction decoder.
- The instruction is originally compressed and the coprocessor accepted the compressed instruction and provided a 32-bit uncompressed instruction. In this case the 32-bit uncompressed instruction will be attempted for offload even if it matches in the *CPU*'s non-compressed instruction decoder.

Apart from the above two main scenarios a *CPU* may also attempt to offload (compressed/uncompressed) instructions that it does recognize as legal instructions itself. In case that both the *CPU* and the coprocessor accept the same instruction as being valid, the instruction will cause an illegal instruction fault upon execution.

In all cases, the *CPU* must decode the instruction. The *CPU* shall cause an illegal instruction fault when attempting to execute (commit) an instruction that:

- is considered to be valid by the *CPU* and accepted by the coprocessor (accept = 1).
- is considered neither to be valid by the *CPU* nor accepted by the coprocessor (accept = 0).

A coprocessor can delay accept accepting an instruction via issue_ready in the presence of structural hazards that would prevent execution. A coprocessor can (only) accept an offloaded instruction when it can handle the instruction (based on decoding instr).

A transaction is considered offloaded/accepted on the positive edge of clk when issue_valid, issue_ready are asserted and accept is 1. A transaction is considered not offloaded/rejected on the positive edge of clk when issue_valid and issue_ready are asserted while accept is 0.

The signals in issue_resp are valid when issue_valid and issue_ready are both 1. There are no stability requirements.

5.5.7 Register interface

Table 5.9 describes the register interface signals.

Table 5.9: Register interface signals

Signal	Type	Direction (*CPU*)	Description
`register_valid`	logic	output	Register request valid. Indicates that *CPU* provides register contents related to an instruction.
`register_ready`	logic	input	Register request ready. The transaction signaled via `register_req` is accepted when `register_valid` and `register_ready` are both 1.
`register`	x_register_t	output	Register packet.

Table 5.10 describes the `x_register_t` type.

Table 5.10: Register type

Signal	Type	Description
`hartid`	*hartid_t*	Identification of the hart offloading the instruction.
`id`	*id_t*	Identification of the offloaded instruction.
`rs[X_NUM_RS-1:0]`	logic [X_RFR_WIDTH-1:0]	Register file source operands for the offloaded instruction.
`rs_valid`	*readregflags_t*	Validity of the register file source operand(s). If register pairs are supported, the validity is signaled for each register within the pair individually.

There are two main scenarios, in how the register interface will be used. They are selected by `X_ISSUE_REGISTER_SPLIT`:

1. `X_ISSUE_REGISTER_SPLIT` = 0: A register transaction can be started in the same clock cycle as the issue transaction (`issue_valid = register_valid`, `issue_ready = register_ready`, `issue_req.hartid = register.hartid` and `issue_req.id = register.id`). In this case, the *CPU* will speculatively provide all possible source registers via `register.rs` when they become available (signalled via the respective `rs_valid` signals). The coprocessor will delay accepting the instruction until all necessary registers are provided, and only then assert `issue_ready` and `register_ready`. The `rs_valid` bits are not required to be stable during the transaction. Each bit can transition from 0 to 1, but is not allowed to transition back to 0 during a transaction. A coprocessor is not expected to wait for all `rs_valid` bits to be 1, but only for those registers it intends to read. The `rs` signals are only required to be stable during the part of a transaction in which these signals are considered to be valid.
2. `X_ISSUE_REGISTER_SPLIT` = 1: For a *CPU* which splits the issue and register interface into subsequent pipeline stages (e.g. because it has a dedicated read registers (RR) stage), the registers will be provided after the issue transaction completed. The *CPU* initiates the register transaction once all registers are available. If the coprocessor is able to accept multiple issue transactions before receiving the registers, the register transaction can occur in a different order. This allows the *CPU* to reorder instructions based on the availability of operands. The coprocessor is always expected to be ready to retrieve its operands via the register interface after accepting the issue of an instruction. Therefore, `register_ready` is tied to 1. The `register_valid` signal will be 1 for one cycle, and `rs_valid` is guaranteed to be equal to the corresponding `issue_resp.register_read`. Thus, a coprocessor can ignore `rs_valid` in this case and a *CPU* may chose to not implement the signal.

In both scenarios, the following applies:

A register transaction is defined as the combination of all `register` signals during which `register_valid` is 1, and the `id` and `hartid` remain unchanged. A *CPU* is allowed to retract its register transaction before it is accepted with register_ready = 1 and it can do so in the following ways:

- Set register_valid = 0.
- Keep register_valid = 1, but change the id or hartid signal (and if desired change the other signals in register).

The hartid, id, and rs_valid signals are valid when register_valid is 1. The rs signal is only considered valid when register_valid is 1 and the corresponding bit in rs_valid is 1 as well.

The rs[X_NUM_RS-1:0] signals provide the register file operand(s) to the coprocessor. In case that XLEN = X_RFR_WIDTH, then the regular register file operands corresponding to rs1, rs2 or rs3 are provided. In case XLEN != X_RFR_WIDTH (i.e. XLEN = 32 and X_RFR_WIDTH = 64), then the rs[X_NUM_RS-1:0] signals provide two 32-bit register file operands per index (corresponding to even/odd register pairs) with the even register specified in rs1, rs2 or rs3. The register file operand for the even register file index is provided in the lower 32 bits; the register file operand for the odd register file index is provided in the upper 32 bits. When reading from the x0, x1 pair, then a value of 0 is returned for the entire operand. The X_DUALREAD parameter defines whether dual read is supported and for which register file sources it is supported.

5.5.8 Commit interface

Table 5.11 describes the commit interface signals.

Table 5.11: Commit interface signals

Signal	Type	Direction (*CPU*)	Description
commit_valid	logic	output	Commit request valid. Indicates that *CPU* has valid commit or kill information for an offloaded instruction. There is no corresponding ready signal (it is implicit and assumed 1). The coprocessor shall be ready to observe the commit_valid and commit_kill signals at any time coincident or after an issue transaction initiation.
commit	x_commit_t	output	Commit packet.

Table 5.12 describes the x_commit_t type.

Table 5.12: Commit packet type

Signal	Type	Description
hartid	*hartid_t*	Identification of the hart offloading the instruction.
id	*id_t*	Identification of the offloaded instruction. Valid when commit_valid is 1.
commit_kill	logic	If commit_valid is 1 and commit_kill is 0, then the *CPU* guarantees that the offloaded instruction (id) and any older (i.e. preceding) instructions are no longer speculative, will not get killed (e.g. due to misspeculation or an exception in a preceding instruction), and are allowed to be committed. If commit_valid is 1 and commit_kill is 1, then the offloaded instruction (id) and any newer (i.e. succeeding) instructions shall be killed in the coprocessor and the coprocessor must guarantee that the related instructions do/did not change architectural state. The taken action only applies to instructions offloaded with the specified hartid.

The commit_valid signal will be 1 exactly one clk cycle. It is not required that a commit transaction is performed for each offloaded instruction individually. Instructions can be signalled to be non-speculative or to be killed in batch. E.g. signalling the oldest instruction to be killed is equivalent to requesting a flush of the coprocessor. The first instruction to be considered not-to-be-killed after a commit transaction with commit_kill as 1, is at earliest an instruction with successful issue transaction starting at least one clock cycle later.

Note: If an instruction is marked in the coprocessor as killed or committed, the coprocessor shall ignore any subsequent commit transaction related to that instruction.

Note: A coprocessor must be tolerant to any possible commit.id, whether this represents and in-flight instruction or not. In this case, the coprocessor may still need to process the request by considering the relevant instructions (either preceding or succeeding) as no longer speculative or to be killed. This behavior supports scenarios in which more than one coprocessor is connected to an issue interface.

A *CPU* is required to mark every instruction that has completed the issue transaction as either killed or non-speculative. This includes accepted (issue_resp.accept = 1) and rejected instructions (issue_resp.accept = 0).

A coprocessor does not have to wait for commit_valid to become asserted. It can speculate that an offloaded accepted instruction will not get killed, but in case this speculation turns out to be wrong because the instruction actually did get killed, then the coprocessor must undo any of its internal architectural state changes that are due to the killed instruction.

A coprocessor is not allowed to perform speculative result transactions and shall therefore never initiate a result transaction for instructions that have not yet received a commit transaction with commit_kill = 0. The earliest point at which a coprocessor can initiate a result handshake for an instruction is therefore the cycle in which commit_valid = 1 and commit_kill = 0 for that instruction.

The signals in commit are valid when commit_valid is 1.

5.5.9 Memory (request/response) interface

The memory (request/response) interface is not included in this version of the specification

5.5.10 Memory result interface

The memory (request/response) interface is not included in this version of the specification

5.5.11 Result interface

Table 5.13 describes the result interface signals.

Table 5.13: Result interface signals

Signal	Type	Direction (*CPU*)	Description
`result_valid`	logic	input	Result request valid. Indicates that the coprocessor has a valid result (write data or exception) for an offloaded instruction.
`result_ready`	logic	output	Result request ready. The result signaled via `result` is accepted by the *CPU* when `result_valid` and `result_ready` are both 1.
`result`	x_result_t	input	Result packet.

The coprocessor shall provide results to the *CPU* via the result interface. A coprocessor is allowed to provide results to the *CPU* in an out of order fashion. A coprocessor is only allowed to provide a result for an instruction once the *CPU* has indicated (via the commit interface) that this instruction is allowed to be committed. Each accepted offloaded (committed and not killed) instruction shall have exactly one result transaction (even if no data needs to be written back to the *CPU*'s register file). No result transaction shall be performed for instructions which have not been accepted for offload or for instructions that have been killed.

Table 5.14 describes the `x_result_t` type.

Table 5.14: Result packet type

Signal	Type	Description
`hartid`	*hartid_t*	Identification of the hart offloading the instruction.
`id`	*id_t*	Identification of the offloaded instruction.
`data`	logic [X_RFW_WIDTH-1:0]	Register file write data value(s).
`rd`	logic [4:0]	Register file destination address(es).
`we`	*writeregflags_t*	Register file write enable(s).

A result transaction starts in the cycle that `result_valid` = 1 and ends in the cycle that both `result_valid` = 1 and `result_ready` = 1. The signals in `result` are valid when `result_valid` is 1. The signals in `result` shall remain stable during a result transaction.

`we` is 2 bits wide when `XLEN` = 32 and `X_RFW_WIDTH` = 64, and 1 bit wide otherwise. The *CPU* shall ignore write-back to `x0`. When a dual write-back is performed to the `x0`, `x1` pair, the entire write shall be ignored, i.e. neither `x0` nor `x1` shall be written by the *CPU*. For an instruction instance, the `we` signal must be the same as `issue_resp.write-back`. The *CPU* is not required to check that these signals match.

Note: `issue_resp.write-back` and `result.we` carry the same information. Nevertheless, `result.we` is provided to simplify the *CPU* logic. Without this signal, the *CPU* would have to look this information up based on the instruction `id`.

5.6 Interface dependencies

The following rules apply to the relative ordering of the interface handshakes:

- The compressed interface transactions are in program order (possibly a subset) and the *CPU* will at least attempt to offload compressed instructions that it does not consider to be valid itself.
- The issue interface transactions are in program order (possibly a subset) and the *CPU* will at least attempt to offload instructions that it does not consider to be valid itself.
- Every issue interface transaction has an associated register interface transaction, if the instruction is not killed before the register transaction. It is not required for register transactions to be in the same order as the issue transactions.
- A register interface transaction cannot be initiated before the corresponding issue interface handshake is initiated.
  - If `X_ISSUE_REGISTER_SPLIT` = 0, it must be initiated a the same time.
  - If `X_ISSUE_REGISTER_SPLIT` = 1, it can only be initiated after the corresponding issue interface handshake is completed.
- Every issue interface transaction (whether accepted or not) must be marked as non-speculative or to be killed by a commit interface transaction.
- If an offloaded instruction is accepted and allowed to commit, then for each such instruction one result transaction must occur via the result interface (even if no write-back needs to happen to the *CPU*'s register file). The transaction ordering on the result interface does not have to correspond to the transaction ordering on the issue interface.
- A commit interface handshake cannot be initiated before the corresponding issue interface handshake is initiated. It is allowed to be initiated at the same time or later.

Note: There is no required ordering between commit and register in case of `X_ISSUE_REGISTER_SPLIT` = 1. In this case, implementations must be tolerant to commit before register and register before commit transaction.

- A result interface handshake cannot be initiated before the corresponding register interface handshake is initiated. It is allowed to be initiated at the same time or later.
- A result interface handshake cannot be initiated before the corresponding instruction has been marked as non-speculative by a commit transaction. It is allowed to be initiated at the same time or later.
- A result interface handshake cannot be (or have been) initiated for killed instructions.

5.7 Handshake rules

The following handshake pairs exist on the eXtension interface:

- `compressed_valid` with `compressed_ready`.
- `issue_valid` with `issue_ready`.
- `register_valid` with `register_ready`.
- `commit_valid` with implicit always ready signal.
- `result_valid` with `result_ready`.

The only rule related to `*_valid` and `*_ready` signals is that:

- A transaction is considered accepted on the positive `clk` edge when both valid and (implicit or explicit) ready are 1.

Note:

- The `*_valid` signals are allowed to be retracted by a *CPU* (e.g. in case that the related instruction is killed in the *CPU*'s pipeline before the corresponding `*_ready` is signaled).
- It is defined per interface, if and how the *CPU* can start a new transaction while a transaction is ongoing (`*_valid` = 1). In most interfaces, it can be started by changing the `hartid` and/or `id` signal and keeping the `*_valid` signal asserted (thereby possibly terminating a previous transaction before it completed).
- The `*_valid` signals are not allowed to be retracted by a coprocessor (e.g. once `result_valid` is asserted it must remain asserted until the handshake with `result_ready` has been performed). A new transaction can therefore not be started by a coprocessor by just changing the `hartid` and/or `id` signal and keeping the valid signal asserted if no `*_ready` has been received yet for the original transaction. The cycle after receiving the `*_ready` signal, a next (back-to-back) transaction is allowed to be started by just keeping the `*_valid` signal high and changing the `hartid` and/or `id` to that of the next transaction.
- The `*_ready` signals are allowed to be 1 when the corresponding `*_valid` signal is 0.
- The `*_valid` signals are allowed to transition from 0 to 1 independent of the `*_ready` signals' states.

5.8 Signal dependencies

A *CPU* shall not have combinatorial paths from its eXtension interface input signals to its eXtension interface output signals, except for the following allowed paths:

- paths from `result_valid`, `result` to `register_valid`, `rs`, `rs_valid`.

Note: The above implies that the non-compressed instruction `instr[31:0]` received via the compressed interface is not allowed to combinatorially feed into the issue interface's `instr[31:0]` instruction.

A coprocessor is allowed (and expected) to have combinatorial paths from its eXtension interface input signals to its eXtension interface output signals. In order to prevent combinatorial loops the following combinatorial paths are not allowed in a coprocessor:

- paths from `register_valid`, `rs`, `rs_valid` to `result_valid`, `result`.

Note: The above implies that a coprocessor has a pipeline stage separating the register file operands from its result generating circuit (similar to the separation between decode stage and execute stage found in many *CPUs*).

Note: As a *CPU* is allowed to retract transactions on its compressed, issue, and register interfaces, the `compressed_ready`, `issue_ready`, and `register_ready` signals will have to depend on signals received from the *CPU* in a combinatorial manner (otherwise these ready signals might be signaled for the wrong `hartid` and `id`).

5.9 System level deadlock avoidance

In order to avoid system level deadlock both the *CPU* and the coprocessor shall obey the following rules:

- The `valid` signal of a transaction shall not be dependent on the corresponding `ready` signal.
- The only allowed dependencies between interfaces for transactions with the same `hartid` and `id` are:
  - Issue may depend on Compressed (e.g. `issue_req.instr` depends on `compressed_resp.instr`)
  - Register may depend on Issue (e.g. `register.rs` may depend on `issue_resp.register_read`) and Compressed
  - Commit may depend on Issue and Compressed
  - Result may depend on Commit, Register (e.g. `result.data` may depend on `register.rs`), Issue (e.g. `result.we` depends on `issue_resp.writeback`), and Compressed

Note: In case of `X_ISSUE_REGISTER_SPLIT` = 0, the issue and register interfaces are coupled. Because commit depends on issue, it is implied that register also cannot depend on commit.

- Transactions with an earlier issued `hartid` and `id` shall not depend on transactions with a later issued `hartid` and `id` (e.g. a coprocessor is not allowed to delay generating `result_valid` = 1 because it first wants to see `commit_valid` = 1 for a newer instruction).

Note: The use of the words *depend* and *dependent* relate to logical relationships, which is broader than combinatorial relationships.

CHAPTER

SIX

APPENDIX

This appendix contains several useful, non-normative pieces of information that help implementing the eXtension Interface.

6.1 SystemVerilog example

In the `src` folder of this project, the file https://github.com/openhwgroup/core-v-xif/blob/main/src/core_v_xif.sv contains a non-normative realization of this specification based on SystemVerilog interfaces. Of course the use of SystemVerilog (interfaces) is not mandatory.

6.2 Coprocessor recommendations

A coprocessor is recommended (but not required) to follow the following suggestions to maximize its re-use potential:

- Avoid using opcodes that are reserved or already used by RISC-V International unless for supporting a standard RISC-V extension.
- Make it easy to change opcode assignments such that a coprocessor can easily be updated if it conflicts with another coprocessor.
- Clearly document the supported and required parameter values.

6.3 Recommendations for implementing multiple coprocessors on a shared interface

It is possible to implement multiple coprocessors, which connect to a single *CPU*. There is no required implementation to do this, but the specification is written with the intention of enabling this scenario. This section provides details per interface on a possible path of integration.

In general, the combination of multiple coprocessors will require de-multiplexing of their signals. The de-multiplexing logic can be reduced to a simple OR combination, if the output signals of the coprocessors not mapped to the instruction are 0. This applies to the compressed interface, the issue interface, and the result interface.

- Compressed interface

  The `compressed_valid` and `compressed_req` signals can be broadcasted to all coprocessors. Each coprocessor drives its `compressed_ready` and `compressed_resp` signals. It is recommended that coprocessors provide the response within the same cycle. In this case, both will be driving `compressed_ready` the same way. The `compressed_resp` signals need to be de-multiplexed based on the `compressed_resp.accept` signals. More than one coprocessor accepting an instruction must be prevented by design.

- Issue interface

  The issue_valid and issue_req signals can be broadcasted to all coprocessors. Each coprocessor drives its issue_ready and issue_resp signals. It is recommended that coprocessors provide the response within the same cycle. In this case, both will be driving issue_ready the same way. The issue_resp signals need to be de-multiplexed based on the issue_resp.accept signals. More than one coprocessor accepting an instruction must be prevented by design.

- Register interface

  The register_valid and register_req signals can be broadcasted to all coprocessors. Each coprocessor drives its register_ready signal. The transition of a coprocessors register_ready signal to 1 should only occur when it is clear that the next transaction is targeting an instruction accepted by it. This can be the case, if there is a clear sequence from issue to register interface, or when register_valid = 1 and id is matching an instruction accepted by the coprocessor. It is possible to provide an OR combination of the register_ready signals as a combined signal to the *CPU*.

- Commit interface

  The commit transaction is unidirectional from the *CPU* to the coprocessor. All signals will be broadcasted to all coprocessors. The definition of the commit interface requires the coprocessors to be functional if faced with id values it did not accept.

- Result interface

  The result_ready signal can be broadcasted to all coprocessors. Each coprocessor drives its result_valid signal. If all instructions in all coprocessors complete execution in a fixed number of *CPU* clock cycles after their register interface transaction completed, and writeback is never stalled (i.e. result_ready is 1 at that time), it is possible to de-multiplex the result based on the result_valid signals. If this cannot be guaranteed, e.g. because a coprocessor implements long executing instructions, out-of-order completion etc., it is necessary to arbitrate and multiplex the result transactions requested by each coprocessor.

6.4 Timing recommendations

The integration of the eXtension interface will vary from *CPU* to *CPU*, and thus require its own set of timing constraints.

CV32E40X eXtension timing budget shows the recommended timing budgets for the coprocessor and (optional) interconnect for the case in which a coprocessor is paired with the CV32E40X (https://github.com/openhwgroup/cv32e40x) processor. As is shown in that timing budget, the coprocessor only receives a small part of the timing budget on the paths through xif_issue_if.issue_req.rs*. This enables the coprocessor to source its operands directly from the CV32E40X register file bypass network, thereby preventing stall cycles in case an offloaded instruction depends on the result of a preceding non-offloaded instruction. This implies that, if a coprocessor is intended for pairing with the CV32E40X, it will be beneficial timing wise if the coprocessor does not directly operate on the rs* source inputs, but registers them instead. To maximize utilization of a coprocessor with various *CPUs*, such registers could be made optional via a parameter.

6.5 Verification

A *UVM* agent for the interface was developed for the verification of CVA6. It can be accessed under https://github.com/openhwgroup/core-v-verif/tree/master/lib/uvm_agents/uvma_cvxif.

BIBLIOGRAPHY

[RISC-V-UNPRIV] The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213, Editors Andrew Waterman and Krste Asanovíc, RISC-V Foundation, December 2019.

[RISC-V-PRIV] The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203, Editors Andrew Waterman, Krste Asanovíc, and John Hauser, RISC-V International, December 2021.

INDEX