CORE-V-Docs Documentation

Davide Schiavone

Apr 07, 2022

CONTENTS:

Editor: **Davide Schiavone** davide@openhwgroup.org

CHAPTER

ONE

INTRODUCTION

CV32E41P is a 4-stage in-order 32-bit RISC-V processor core. The ISA of CV32E41P has been extended to support multiple additional instructions including hardware loops, post-increment load and store instructions and additional ALU instructions that are not part of the standard RISC-V ISA. Figure 1.1 shows a block diagram of the core.

Figure 1.1: Block Diagram of CV32E41P RISC-V Core

1.1 License

1.2 Standards Compliance

CV32E41P is a standards-compliant 32-bit RISC-V processor. It follows these specifications:

- RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213 (December 13, 2019)
- RISC-V Instruction Set Manual, Volume II: Privileged Architecture, document version 20190608-Base-Ratified (June 8, 2019). CV32E41P implements the Machine ISA version 1.11.
- RISC-V External Debug Support, version 0.13.2

Many features in the RISC-V specification are optional, and CV32E41P can be parameterized to enable or disable some of them.

CV32E41P supports the following base instruction set.

- The RV32I Base Integer Instruction Set, version 2.1

In addition, the following standard instruction set extensions are available.

Table 1.1: CV32E41P Standard Instruction Set Extensions

Standard Extension	Version	Configurability
C: Standard Extension for Compressed Instructions	2.0	always enabled
M: Standard Extension for Integer Multiplication and Division	2.0	always enabled
Zicount: Performance Counters	2.0	always enabled
Zicsr: Control and Status Register Instructions	2.0	always enabled
Zifencei: Instruction-Fetch Fence	2.0	always enabled
F: Single-Precision Floating-Point using F registers	2.2	optionally enabled with the `FPU` parameter
Zfinx: Single-Precision Floating-Point using X registers	1.0	optionally enabled with the `ZFINX` parameter (also requires the `FPU` parameter)

The following custom instruction set extensions are available.

Table 1.2: CV32E41P Custom Instruction Set Extensions

Custom Extension	Version	Configurability
Xcorev: CORE-V ISA Extensions (excluding **cv.elw**)	1.0	optionally enabled with the `PULP_XPULP` parameter
Xpulpcluster: PULP Cluster Extension	1.0	optionally enabled with the `PULP_CLUSTER` parameter

Most content of the RISC-V privileged specification is optional. CV32E41P currently supports the following features according to the RISC-V Privileged Specification, version 1.11.

- M-Mode
- All CSRs listed in *Control and Status Registers*
- Hardware Performance Counters as described in *Performance Counters* controlled by the `NUM_MHPMCOUNTERS` parameter
- Trap handling supporting direct mode or vectored mode as described at *Exceptions and Interrupts*

1.3 Synthesis guidelines

The CV32E41P core is fully synthesizable. It has been designed mainly for ASIC designs, but FPGA synthesis is supported as well.

All the files in the `rtl` and `rtl/include` folders are synthesizable. The user should first decide whether to use the flip-flop or latch-based register-file (see *Register File*). Secondly, the user must provide a clock-gating module that instantiates the clock-gating cells of the target technology. This file must have the same interface and module name of the one provided for simulation-only purposes at `bhv/cv32e41p_sim_clock_gate.sv` (see *Clock Gating Cell*).

The `constraints/cv32e41p_core.sdc` file provides an example of synthesis constraints.

1.3.1 ASIC Synthesis

ASIC synthesis is supported for CV32E41P. The whole design is completely synchronous and uses positive-edge triggered flip-flops, except for the register file, which can be implemented either with latches or with flip-flops. See *Register File* for more details. The core occupies an area of about 50 kGE when the latch based register file is used. With the FPU, the area increases to about 90 kGE (30 kGE FPU, 10 kGE additional register file). A technology specific implementation of a clock gating cell as described in *Clock Gating Cell* needs to be provided.

1.3.2 FPGA Synthesis

FPGA synthesis is only supported for CV32E41P when the flip-flop based register file is used as latches are not well supported on FPGAs.

The user needs to provide a technology specific implementation of a clock gating cell as described in *Clock Gating Cell*.

1.4 Verification

The verification environment (testbenches, testcases, etc.) for the CV32E41P core can be found at core-v-verif. It is recommended that you start by reviewing the CORE-V Verification Strategy.

1.5 Contents

- *Getting Started with CV32E41P* discusses the requirements and initial steps to start using CV32E41P.
- *Core Integration* provides the instantiation template and gives descriptions of the design parameters as well as the input and output ports.
- *CV32E41P Pipeline* described the overal pipeline structure.
- The instruction and data interfaces of CV32E41P are explained in *Instruction Fetch* and *Load-Store-Unit (LSU)*, respectively.
- The two register-file flavors are described in *Register File*.
- *Auxiliary Processing Unit (APU)* describes the Auxiliary Processing Unit (APU).
- *Floating Point Unit (FPU)* describes the Floating Point Unit (FPU).
- *Sleep Unit* describes the Sleep unit including the PULP Cluster extension.
- *CORE-V Hardware Loop Extensions* describes the PULP Hardware Loop extension.
- The control and status registers are explained in *Control and Status Registers*.
- *Performance Counters* gives an overview of the performance monitors and event counters available in CV32E41P.
- *Exceptions and Interrupts* deals with the infrastructure for handling exceptions and interrupts.
- *Debug & Trigger* gives a brief overview on the debug infrastructure.
- *Tracer* gives a brief overview of the tracer module.
- *CORE-V Instruction Set Extensions* describes the custom instruction set extensions.
- *Glossary* provides definitions of used terminology.

CHAPTER

TWO

GETTING STARTED WITH CV32E41P

This page discusses initial steps and requirements to start using CV32E41P in your design.

2.1 Register File

CV32E41P comes with two different register file implementations. Depending on the target technology, either the implementation in `cv32e41p_register_file_ff.sv` or the one in `cv32e41p_register_file_latch.sv` should be selected in the manifest file. For more information about the two register file implementations and their trade-offs, check out *Register File*.

2.2 Clock Gating Cell

CV32E41P requires clock gating cells. These cells are usually specific to the selected target technology and thus not provided as part of the RTL design. A simulation-only version of the clock gating cell is provided in `cv32e41p_sim_clock_gate.sv`. This file contains a module called `cv32e41p_clock_gate` that has the following ports:

- `clk_i`: Clock Input
- `en_i`: Clock Enable Input
- `scan_cg_en_i`: Scan Clock Gate Enable Input (activates the clock even though `en_i` is not set)
- `clk_o`: Gated Clock Output

Inside CV32E41P, clock gating cells are used both in `cv32e41p_sleep_unit.sv` and `cv32e41p_register_file_latch.sv`. For more information on the expected behavior of the clock gating cell when using the latch-based register file check out *Register File*.

The `cv32e41p_sim_clock_gate.sv` file is not intended for synthesis. For ASIC synthesis and FPGA synthesis the manifest should be adapted to use a customer specific file that implements the `cv32e41p_clock_gate` module using design primitives that are appropriate for the intended synthesis target technology.

CHAPTER

THREE

CORE INTEGRATION

The main module is named `cv32e41p_core` and can be found in `cv32e41p_core.sv`. Below, the instantiation template is given and the parameters and interfaces are described.

3.1 Instantiation Template

```
cv32e41p_core #(
    .FPU                      ( 0 ),
    .NUM_MHPMCOUNTERS         ( 1 ),
    .PULP_CLUSTER             ( 0 ),
    .PULP_XPULP               ( 0 ),
    .ZFINX                    ( 0 )
    .Zcea                     ( 0 ) //FIXME these will change names
    .Zceb                     ( 0 ) //when moving to v0.70
    .Zcec                     ( 0 )
    .Zcee                     ( 0 )
) u_core (
    // Clock and reset
    .clk_i                    (),
    .rst_ni                   (),
    .scan_cg_en_i             (),

    // Configuration
    .boot_addr_i              (),
    .mtvec_addr_i             (),
    .dm_halt_addr_i           (),
    .dm_exception_addr_i      (),
    .hart_id_i                (),

    // Instruction memory interface
    .instr_req_o              (),
    .instr_gnt_i              (),
    .instr_rvalid_i           (),
    .instr_addr_o             (),
    .instr_rdata_i            (),

    // Data memory interface
    .data_req_o               (),
    .data_gnt_i               (),
```

(continues on next page)

(continued from previous page)

```
    .data_rvalid_i            (),
    .data_addr_o              (),
    .data_be_o                (),
    .data_wdata_o             (),
    .data_we_o                (),
    .data_rdata_i             (),

    // Auxiliary Processing Unit (APU) interface
    .apu_req_o                (),
    .apu_gnt_i                (),
    .apu_operands_o           (),
    .apu_op_o                 (),
    .apu_flags_o              (),
    .apu_rvalid_i             (),
    .apu_result_i             (),
    .apu_flags_i              (),

     // Interrupt interface
    .irq_i                    (),
    .irq_ack_o                (),
    .irq_id_o                 (),

    // Debug interface
    .debug_req_i              (),
    .debug_havereset_o        (),
    .debug_running_o          (),
    .debug_halted_o           (),

    // Special control signals
    .fetch_enable_i           (),
    .core_sleep_o             (),
    .pulp_clock_en_i          ()
);
```

3.2 Parameters

Note: The non-default (i.e. non-zero) settings of `FPU`, `PULP_CLUSTER`, `PULP_XPULP` and `ZFINX` have not been verified yet. The default parameter value for `PULP_XPULP` will be changed to 1 once it has been verified. The default configuration reflected below is currently under verification and this verification effort will be completed first.

Note: The instruction encodings for the PULP instructions is expected to change in a non-backward-compatible manner, see https://github.com/openhwgroup/cv32e41p/issues/452.

Name	Type/Range	Default	Description
FPU	bit	0	Enable Floating Point Unit (FPU) support, see *Floating Point Unit (FPU)*
NUM_MHPMCOUNTERS	MHPMCOUNTERS (0..29)	MHPMCOUNTERS	Number of MHPMCOUNTER performance counters, see *Performance Counters*
PULP_CLUSTER	bit	0	Enable PULP Cluster support, see *PULP Cluster Extension*
PULP_XPULP	bit	0	Enable all of the custom PULP ISA extensions (except **cv.elw**) (see *CORE-V Instruction Set Extensions*) and all custom CSRs (see *Control and Status Registers*). Examples of PULP ISA extensions are post-incrementing load and stores (see *Post-Incrementing Load & Store Instructions and Register-Register Load & Store Instructions*) and hardware loops (see *Hardware Loops*).
ZFINX	bit	0	Enable Floating Point instructions to use the General Purpose register file instead of requiring a dedicated Floating Point register file, see *Floating Point Unit (FPU)*. Only allowed to be set to 1 if FPU = 1
Zcea	bit	0	Enable all Zcea instruction from Zce v0.50.1
Zceb	bit	0	Enable all Zceb instruction from Zce v0.50.1
Zcec	bit	0	Enable all Zcec instruction from Zce v0.50.1
Zcee	bit	0	Enable all Zcee instruction from Zce v0.50.1

3.3 Interfaces

Signal(s)	Width	Dir	Description
clk_i	1	in	Clock signal
rst_ni	1	in	Active-low asynchronous reset
scan_cg_en_i	1	in	Scan clock gate enable. Design for test (DfT) related signal. Can be used during scan testing operation to force instantiated clock gate(s) to be enabled. This signal should be 0 during normal / functional operation.
boot_addr_i	32	in	Boot address. First program counter after reset = boot_addr_i. Must be half-word aligned. Do not change after enabling core via fetch_enable_i
mtvec_addr_i	32	in	mtvec address. Initial value for the address part of *Machine Trap-Vector Base Address (mtvec)*. Do not change after enabling core via fetch_enable_i
dm_halt_addr_i	32	in	Address to jump to when entering Debug Mode, see *Debug & Trigger*. Must be word-aligned. Do not change after enabling core via fetch_enable_i
dm_exception_addr_i	32	in	Address to jump to when an exception occurs when executing code during Debug Mode, see *Debug & Trigger*. Must be word-aligned. Do not change after enabling core via fetch_enable_i
hart_id_i	32	in	Hart ID, usually static, can be read from *Hardware Thread ID (mhartid)* and *User Hardware Thread ID (uhartid)* CSRs
instr_*	Instruction fetch interface, see *Instruction Fetch*		
data_*	Load-store unit interface, see *Load-Store-Unit (LSU)*		
apu_*	Auxiliary Processing Unit (APU) interface, see *Auxiliary Processing Unit (APU)*		
irq_*	Interrupt inputs, see *Exceptions and Interrupts*		
debug_*	Debug interface, see *Debug & Trigger*		
fetch_enable_i	1	in	Enable the instruction fetch of CV32E41P. The first instruction fetch after reset deassertion will not happen as long as this signal is 0. fetch_enable_i needs to be set to 1 for at least one cycle while not in reset to enable fetching. Once fetching has been enabled the value fetch_enable_i is ignored.
core_sleep_o	1	out	Core is sleeping, see *Sleep Unit*.
pulp_clock_en_i	1	in	PULP clock enable (only used when PULP_CLUSTER = 1, tie to 0 otherwise), see *Sleep Unit*

Figure 3.1: CV32E41P Pipeline

CHAPTER

FOUR

PIPELINE DETAILS

CV32E41P has a 4-stage in-order completion pipeline, the 4 stages are:

Instruction Fetch (IF) Fetches instructions from memory via an aligning prefetch buffer, capable of fetching 1 instruction per cycle if the instruction side memory system allows. See *Instruction Fetch* for details.

Instruction Decode (ID) Decodes fetched instruction and performs required register file reads. Jumps are taken from the ID stage.

Execute (EX) Executes the instructions. The EX stage contains the ALU, Multiplier and Divider. Branches (with their condition met) are taken from the EX stage. Multi-cycle instructions will stall this stage until they are complete. The ALU, Multiplier and Divider instructions write back their result to the register file from the EX stage. The address generation part of the load-store-unit (LSU) is contained in EX as well.

Writeback (WB) Writes the result of Load instructions back to the register file.

4.1 Multi- and Single-Cycle Instructions

Table 4.1 shows the cycle count per instruction type. Some instructions have a variable time, this is indicated as a range e.g. 3..35 means that the instruction takes a minimum of 3 cycles and a maximum of 35 cycles. The cycle counts assume zero stall on the instruction-side interface and zero stall on the data-side memory interface.

Table 4.1: Cycle counts per instruction type

Instruction Type	Cycles	Description
Integer Computational	1	Integer Computational Instructions are defined in the RISCV-V RV32I Base Integer Instruction Set.
CSR Access	4 (mstatus, mepc, mtvec, mcause, mcycle, minstret, mhpmcounter*, mcycleh, minstreth, mhpmcounter*h, mcountinhibit, mhpmevent*, dscr, dpc, dscratch0, dscratch1, privlv) 1 (all the other CSRs)	CSR Access Instruction are defined in 'Zicsr' of the RISC-V specification.
Load/Store	1 2 (non-word aligned word transfer) 2 (halfword transfer crossing word boundary) 4 (cv.elw)	Load/Store is handled in 1 bus transaction using both EX and WB stages for 1 cycle each. For misaligned word transfers and for halfword transfers that cross a word boundary 2 bus transactions are performed using EX and WB stages for 2 cycles each. A **cv.elw** takes 4 cycles.
Multiplication	1 (mul) 5 (mulh, mulhsu, mulhu)	CV32E41P uses a single-cycle 32-bit x 32-bit multiplier with a 32-bit result. The multiplications with upper-word result take 5 cycles to compute.
Division Remainder	3 - 35 3 - 35	The number of cycles depends on the divider operand value (operand b), i.e. in the number of leading bits at 0. The minimum number of cycles is 3 when the divider has zero leading bits at 0 (e.g., 0x8000000). The maximum number of cycles is 35 when the divider is 0
Jump	2 3 (target is a non-word-aligned non-RVC instruction)	Jumps are performed in the ID stage. Upon a jump the IF stage (including prefetch buffer) is flushed. The new PC request will appear on the instruction-side memory interface the same cycle the jump instruction is in the ID stage.
Branch (Not-Taken)	1	Any branch where the condition is not met will not stall.
Branch (Taken)	3 4 (target is a non-word-aligned non-RVC instruction)	The EX stage is used to compute the branch decision. Any branch where the condition is met will be taken from the EX stage and will cause a flush of the IF stage (including prefetch buffer) and ID stage.
Instruction Fence	2 3 (target is a non-word-aligned non-RVC instruction)	The FENCE.I instruction as defined in 'Zifencei' of the RISC-V specification. Internally it is implemented as a jump to the instruction following the fence. The jump performs the required flushing as described above.

4.2 Hazards

The CV32E41P experiences a 1 cycle penalty on the following hazards.

- Load data hazard (in case the instruction immediately following a load uses the result of that load)
- Jump register (jalr) data hazard (in case that a jalr depends on the result of an immediately preceding instruction)

CHAPTER

FIVE

INSTRUCTION FETCH

The Instruction Fetch (IF) stage of the CV32E41P is able to supply one instruction to the Instruction Decode (ID) stage per cycle if the external bus interface is able to serve one instruction per cycle. In case of executing compressed instructions, on average less than one 32-bit instruction fetch will we needed per instruction in the ID stage.

For optimal performance and timing closure reasons, a prefetcher is used which fetches instructions via the external bus interface from for example an externally connected instruction memory or instruction cache.

The prefetch unit performs word-aligned 32-bit prefetches and stores the fetched words in a FIFO with four entries. As a result of this (speculative) prefetch, CV32E41P can fetch up to four words outside of the code region and care should therefore be taken that no unwanted read side effects occur for such prefetches outside of the actual code region.

Table 5.1 describes the signals that are used to fetch instructions. This interface is a simplified version of the interface that is used by the LSU, which is described in *Load-Store-Unit (LSU)*. The difference is that no writes are possible and thus it needs fewer signals.

Table 5.1: Instruction Fetch interface signals

Signal	Direction	Description
instr_req_o	output	Request valid, will stay high until instr_gnt_i is high for one cycle
instr_addr_o[31:0]	output	Address, word aligned
instr_rdata_i[31:0]	input	Data read from memory
instr_rvalid_i	input	instr_rdata_i holds valid data when instr_rvalid_i is high. This signal will be high for exactly one cycle per request.
instr_gnt_i	input	The other side accepted the request. instr_addr_o may change in the next cycle.

5.1 Misaligned Accesses

Externally, the IF interface performs word-aligned instruction fetches only. Misaligned instruction fetches are handled by performing two separate word-aligned instruction fetches. Internally, the core can deal with both word- and half-word-aligned instruction addresses to support compressed instructions. The LSB of the instruction address is ignored internally.

5.2 Protocol

The instruction bus interface is compliant to the OBI (Open Bus Interface) protocol. See https://github.com/openhwgroup/core-v-docs/blob/master/cores/obi/OBI-v1.2.pdf for details about the protocol. The CV32E41P instruction fetch interface does not implement the following optional OBI signals: we, be, wdata, auser, wuser, aid, rready, err, ruser, rid. These signals can be thought of as being tied off as specified in the OBI specification. The CV32E41P instruction fetch interface can cause up to two outstanding transactions.

Figure 5.1 and Figure 5.2 show example timing diagrams of the protocol.

Figure 5.1: Back-to-back Memory Transactions

Figure 5.2: Multiple Outstanding Memory Transactions

CHAPTER

SIX

LOAD-STORE-UNIT (LSU)

The Load-Store Unit (LSU) of the core takes care of accessing the data memory. Load and stores on words (32 bit), half words (16 bit) and bytes (8 bit) are supported.

Table 6.1 describes the signals that are used by the LSU.

Table 6.1: LSU interface signals

Signal	Direction	Description
data_req_o	output	Request valid, will stay high until data_gnt_i is high for one cycle
data_addr_o[31:0]	output	Address
data_we_o	output	Write Enable, high for writes, low for reads. Sent together with data_req_o
data_be_o[3:0]	output	Byte Enable. Is set for the bytes to write/read, sent together with data_req_o
data_wdata_o[31:0]	output	Data to be written to memory, sent together with data_req_o
data_rdata_i[31:0]	input	Data read from memory
data_rvalid_i	input	data_rvalid_i will be high for exactly one cycle to signal the end of the response phase of for both read and write transactions. For a read transaction data_rdata_i holds valid data when data_rvalid_i is high.
data_gnt_i	input	The other side accepted the request. data_addr_o may change in the next cycle.

6.1 Misaligned Accesses

The LSU never raises address-misaligned exceptions. For loads and stores where the effective address is not naturally aligned to the referenced datatype (i.e., on a four-byte boundary for word accesses, and a two-byte boundary for halfword accesses) the load/store is performed as two bus transactions in case that the data item crosses a word boundary. A single load/store instruction is therefore performed as two bus transactions for the following scenarios:

- Load/store of a word for a non-word-aligned address
- Load/store of a halfword crossing a word address boundary

In both cases the transfer corresponding to the lowest address is performed first. All other scenarios can be handled with a single bus transaction.

6.2 Protocol

The data bus interface is compliant to the OBI (Open Bus Interface) protocol. See https://github.com/openhwgroup/core-v-docs/blob/master/cores/obi/OBI-v1.2.pdf for details about the protocol. The CV32E41P data interface does not implement the following optional OBI signals: auser, wuser, aid, rready, err, ruser, rid. These signals can be thought of as being tied off as specified in the OBI specification. The CV32E41P data interface can cause up to two outstanding transactions.

The OBI protocol that is used by the LSU to communicate with a memory works as follows.

The LSU provides a valid address on `data_addr_o`, control information on `data_we_o`, `data_be_o` (as well as write data on `data_wdata_o` in case of a store) and sets `data_req_o` high. The memory sets `data_gnt_i` high as soon as it is ready to serve the request. This may happen at any time, even before the request was sent. After a request has been granted the address phase signals (`data_addr_o`, `data_we_o`, `data_be_o` and `data_wdata_o`) may be changed in the next cycle by the LSU as the memory is assumed to already have processed and stored that information. After granting a request, the memory answers with a `data_rvalid_i` set high if `data_rdata_i` is valid. This may happen one or more cycles after the request has been granted. Note that `data_rvalid_i` must also be set high to signal the end of the response phase for a write transaction (although the `data_rdata_i` has no meaning in that case). When multiple granted requests are outstanding, it is assumed that the memory requests will be kept in-order and one `data_rvalid_i` will be signalled for each of them, in the order they were issued.

Figure 6.1, Figure 6.2, Figure 6.3 and Figure 6.4 show example timing diagrams of the protocol.

Figure 6.1: Basic Memory Transaction

Figure 6.2: Back-to-back Memory Transactions

Figure 6.3: Slow Response Memory Transaction

6.3 Post-Incrementing Load and Store Instructions

This section is only valid if `PULP_XPULP=1`

Post-incrementing load and store instructions perform a load/store operation from/to the data memory while at the same time increasing the base address by the specified offset. For the memory access, the base address without offset is used.

Post-incrementing load and stores reduce the number of required instructions to execute code with regular data access patterns, which can typically be found in loops. These post-incrementing load/store instructions allow the address increment to be embedded in the memory access instructions and get rid of separate instructions to handle pointers. Coupled with hardware loop extension, these instructions allow to reduce the loop overhead significantly.

Figure 6.4: Multiple Outstanding Memory Transactions

CHAPTER

SEVEN

REGISTER FILE

Source files: `rtl/cv32e41p_register_file_ff.sv` `rtl/cv32e41p_register_file_latch.sv`

CV32E41P has 31 32-bit wide registers which form registers `x1` to `x31`. Register `x0` is statically bound to 0 and can only be read, it does not contain any sequential logic.

The register file has three read ports and two write ports. Register file reads are performed in the ID stage. Register file writes are performed in the WB stage.

There are two flavors of register file available.

- Flip-flop based (`rtl/cv32e41p_register_file_ff.sv`)
- Latch-based (`rtl/cv32e41p_register_file_latch.sv`)

Both flavors have their own benefits and trade-offs. While the latch-based register file is recommended for ASICs, the flip-flop based register file is recommended for FPGA synthesis, although both are compatible with either synthesis target. Note the flip-flop based register file is significantly larger than the latch-based register-file for an ASIC implementation.

7.1 Flip-Flop-Based Register File

The flip-flop-based register file uses regular, positive-edge-triggered flip-flops to implement the registers. This makes it the **first choice when simulating the design using Verilator**. To select the flip-flop-based register file, make sure to use the source file `cv32e41p_register_file_ff.sv` in your project.

7.2 Latch-based Register File

The latch-based register file uses level-sensitive latches to implement the registers.

This allows for significant area savings compared to an implementation using regular flip-flops and thus makes the latch-based register file the **first choice for ASIC implementations**. Simulation of the latch-based register file is possible using commercial tools.

Note: The latch-based register file cannot be simulated using Verilator.

The latch-based register file can also be used for FPGA synthesis, but this is not recommended as FPGAs may not support latches.

To select the latch-based register file, make sure to use the source file `cv32e41p_register_file_latch.sv` in your project. In addition, a technology-specific clock gating cell must be provided to keep the clock inactive when the

latches are not written. This cell must be wrapped in a module called `cv32e41p_clock_gate`. For more information regarding the clock gating cell, checkout *Getting Started with CV32E41P*.

7.3 FPU Register File

If the optional FPU is instantiated, unless ZFINX is configured, the register file is extended with an additional register bank of 32 registers `f0-f31`. These registers are stacked on top of the existing register file and can be accessed concurrently with the limitation that a maximum of three operands per cycle can be read. Each of the three operands addresses is extended with a register file select signal which is generated in the instruction decoder when a FP instruction is decoded. This additional signals determines if the operand is located in the integer or the floating point register file.

Forwarding paths, and write-back logic are shared for the integer and floating point operations and are not replicated.

CHAPTER

EIGHT

AUXILIARY PROCESSING UNIT (APU)

8.1 Auxiliary Processing Unit Interface

Table 8.1 describes the signals of the Auxiliary Processing Unit interface.

Table 8.1: Auxiliary Processing Unit interface signals

Signal	Direction	Description
apu_req_o	output	Request valid, will stay high until apu_gnt_i is high for one cycle
apu_gnt_i	input	The other side accepted the request. apu_operands_o, apu_op_o, apu_flags_o may change in the next cycle.
apu_operands_o[2:0][31:0]	output	APU's operands
apu_op_o[5:0]	output	APU's operation
apu_flags_o[14:0]	output	APU's flags
apu_rvalid_i	input	apu_result_i holds valid data when apu_valid_i is high. This signal will be high for exactly one cycle per request
apu_result_i[31:0]	input	APU's result
apu_flags_i[4:0]	input	APU's flag result

8.2 Protocol

The apu bus interface is derived from to the OBI (Open Bus Interface) protocol. See https://github.com/openhwgroup/core-v-docs/blob/master/cores/obi/OBI-v1.2.pdf for details about the protocol. The CV32E41P apu interface uses the apu_operands_o, apu_op_o, and apu_flags_o as the address signal during the Address phase, indicating its validity with the apu_req_o signal. It uses the apu_result_i and apu_flags_i as the rdata of the response phase. It does not implement the OBI signals: we, be, wdata, auser, wuser, aid, rready, err, ruser, rid. These signals can be thought of as being tied off as specified in the OBI specification. The CV32E41P apu interface can cause up to two outstanding transactions.

8.3 Connection with the FPU

The CV32E41P sends FP operands over the `apu_operands_o` bus; the decoded RV32F operation as ADD, SUB, MUL, etc through the `apu_op_o` bus; the cast, destination and source formats as well as rounding mode through the `apu_flags_o` bus. The response is the FPU result and relative output flags as Overflow, Underflow, etc.

8.4 APU Tracer

The module `cv32e41p_apu_tracer` can be used to create a log of the APU interface. It is a behavioral, non-synthesizable, module instantiated in the example testbench that is provided for the `cv32e41p_core`. It can be enabled during simulation by defining **CV32E41P_APU_TRACE**.

8.5 Output file

The APU trace is written to a log file which is named `apu_trace_core_<HARTID>.log`, with `<HARTID>` being the 32 digit hart ID of the core being traced.

8.6 Trace output format

The trace output is in tab-separated columns.

1. **Time**: The current simulation time.
2. **Register**: The register file write address.
3. **Result**: The register file write data.

CHAPTER

NINE

FLOATING POINT UNIT (FPU)

The RV32F ISA extension for floating-point support in the form of IEEE-754 single precision can be enabled by setting the parameter **FPU** of the toplevel file `cv32e41p_core.sv` to 1. This will extend the CV32E41P decoder accordingly. The actual Floating Point Unit (FPU) is instantiated outside the CV32E41P and is accessed via the APU interface (see *Auxiliary Processing Unit (APU)*). The FPU repository used by the CV32E41P core is available at https://github.com/pulp-platform/fpnew. In the core repository, a wrapper showing how the FPU is connected to the core is available at `example_tb/core/cv32e41p_fp_wrapper.sv`. By default a dedicated register file consisting of 32 floating-point registers, `f0-f31`, is instantiated. This default behavior can be overruled by setting the parameter **ZFINX** of the toplevel file `cv32e41p_core.sv` to 1, in which case the dedicated register file is not included and the general purpose register file is used instead to host the floating-point operands.

The latency of the individual instructions are set by means of parameters in the FPU repository (see https://github.com/pulp-platform/fpnew/tree/develop/docs).

9.1 FP CSR

When using floating-point extensions the standard specifies a floating-point status and control register (*Floating-point control and status register (fcsr)*) which contains the exceptions that occurred since it was last reset and the rounding mode. *Floating-point accrued exceptions (fflags)* and *Floating-point dynamic rounding mode (frm)* can be accessed directly or via *Floating-point control and status register (fcsr)* which is mapped to those two registers.

CHAPTER

TEN

SLEEP UNIT

Source File: `rtl/cv32e41p_sleep_unit.sv`

The Sleep Unit contains and controls the instantiated clock gate, see *Clock Gating Cell*, that gates `clk_i` and produces a gated clock for use by the other modules inside CV32E41P. The Sleep Unit is the only place in which `clk_i` itself is used; all other modules use the gated version of `clk_i`.

The clock gating in the Sleep Unit is impacted by the following:

- `rst_ni`
- `fetch_enable_i`
- **wfi** instruction (only when `PULP_CLUSTER` = 0)
- **cv.elw** instruction (only when `PULP_CLUSTER` = 1)
- `pulp_clock_en_i` (only when `PULP_CLUSTER` = 1)

Table 10.1 describes the Sleep Unit interface.

Table 10.1: Sleep Unit interface signals

Signal	Direction	Description
`pulp_clock_en_i`	Input	`PULP_CLUSTER` = 0: `pulp_clock_en_i` is not used. Tie to 0.
		`PULP_CLUSTER` = 1: `pulp_clock_en_i` can be used to gate `clk_i` internal to the core when `core_sleep_o` = 1. See *PULP Cluster Extension* for details.
`core_sleep_o`	Output	`PULP_CLUSTER` = 0: Core is sleeping because of a **wfi** instruction. If `core_sleep_o` = 1, then `clk_i` is gated off internally and it is allowed to gate off `clk_i` externally as well. See *WFI* for details.
		`PULP_CLUSTER` = 1: Core is sleeping because of a **cv.elw** instruction. If `core_sleep_o` = 1, then the `pulp_clock_en_i` directly controls the internally instantiated clock gate and therefore `pulp_clock_en_i` can be set to 0 to internally gate off `clk_i`. If `core_sleep_o` = 0, then it is not allowed to set `pulp_clock_en_i` to 0. See *PULP Cluster Extension* for details.

Note: The semantics of `pulp_clock_en_i` and `core_sleep_o` depend on the `PULP_CLUSTER` parameter.

10.1 Startup behavior

clk_i is internally gated off (while signaling core_sleep_o = 0) during CV32E41P startup:

- clk_i is internally gated off during rst_ni assertion
- clk_i is internally gated off from rst_ni deassertion until fetch_enable_i = 1

After initial assertion of fetch_enable_i, the fetch_enable_i signal is ignored until after a next reset assertion.

10.2 WFI

The **wfi** instruction can under certain conditions be used to enter sleep mode awaiting a locally enabled interrupt to become pending. The operation of **wfi** is unaffected by the global interrupt bits in **mstatus**.

A **wfi** will not enter sleep mode, but will be executed as a regular **nop**, if any of the following conditions apply:

- debug_req_i = 1 or a debug request is pending
- The core is in debug mode
- The core is performing single stepping (debug)
- The core has a trigger match (debug)
- PULP_CLUSTER = 1

If a **wfi** causes sleep mode entry, then core_sleep_o is set to 1 and clk_i is gated off internally. clk_i is allowed to be gated off externally as well in this scenario. A wake-up can be triggered by any of the following:

- A locally enabled interrupt is pending
- A debug request is pending
- Core is in debug mode

Upon wake-up core_sleep_o is set to 0, clk_i will no longer be gated internally, must not be gated off externally, and instruction execution resumes.

If one of the above wake-up conditions coincides with the **wfi** instruction, then sleep mode is not entered and core_sleep_o will not become 1.

Figure 10.1 shows an example waveform for sleep mode entry because of a **wfi** instruction.

Figure 10.1: **wfi** example

10.3 PULP Cluster Extension

CV32E41P has an optional extension to enable its usage in a PULP Cluster in the PULP (Parallel Ultra Low Power) platform. This extension is enabled by setting the PULP_CLUSTER parameter to 1. The PULP platform is organized as clusters of multiple (typically 4 or 8) CV32E41P cores that share a tightly-coupled data memory, aimed at running digital signal processing applications efficiently.

The mechanism via which CV32E41P cores in a PULP Cluster synchronize with each other is implemented via the custom **cv.elw** instruction that performs a read transaction on an external Event Unit (which for example implements barriers and semaphores). This read transaction to the Event Unit together with the core_sleep_o signal inform the Event Unit that the CV32E41P is not busy and ready to go to sleep. Only in that case the Event Unit is allowed to set

pulp_clock_en_i to 0, thereby gating off clk_i internal to the core. Once the CV32E41P core is ready to start again (e.g. when the last core meets the barrier), pulp_clock_en_i is set to 1 thereby enabling the CV32E41P to run again.

If the PULP Cluster extension is not used (PULP_CLUSTER = 0), the pulp_clock_en_i signal is not used and should be tied to 0.

Execution of a **cv.elw** instructions causes core_sleep_o = 1 only if all of the following conditions are met:

- The **cv.elw** did not yet complete (which can be achieved by witholding data_gnt_i and/or data_rvalid_i)
- No debug request is pending
- The core is not in debug mode
- The core is not single stepping (debug)
- The core does not have a trigger match (debug)

As pulp_clock_en_i can directly impact the internal clock gate, certain requirements are imposed on the environment of CV32E41P in case PULP_CLUSTER = 1:

- If core_sleep_o = 0, then pulp_clock_en_i must be 1
- If pulp_clock_en_i = 0, then irq_i[] must be 0
- If pulp_clock_en_i = 0, then debug_req_i must be 0
- If pulp_clock_en_i = 0, then instr_rvalid_i must be 0
- If pulp_clock_en_i = 0, then instr_gnt_i must be 0
- If pulp_clock_en_i = 0, then data_rvalid_i must be 0
- If pulp_clock_en_i = 0, then data_gnt_i must be 0

Figure 10.2 shows an example waveform for sleep mode entry because of a **cv.elw** instruction.

Figure 10.2: **cv.elw** example

CHAPTER

ELEVEN

CORE-V HARDWARE LOOP EXTENSIONS

To increase the efficiency of small loops, CV32E41P supports hardware loops (HWLoop) optionally. They can be enabled by setting the `PULP_XPULP` parameter. Hardware loops make executing a piece of code multiple times possible, without the overhead of branches or updating a counter. Hardware loops involve zero stall cycles for jumping to the first instruction of a loop.

A hardware loop is defined by its start address (pointing to the first instruction in the loop), its end address (pointing to the instruction that will be executed last in the loop) and a counter that is decremented every time the loop body is executed. CV32E41P contains two hardware loop register sets to support nested hardware loops, each of them can store these three values in separate flip flops which are mapped in the CSR address space. Loop number 0 has higher priority than loop number 1 in a nested loop configuration, meaning that loop 0 represents the inner loop.

11.1 Hardware Loop constraints

The HWLoop constraints are:

- Start and End address of an HWLoop must be word aligned
- HWLoop body must contain at least 3 instructions. An illegal exception is raised otherwise.
- No Compressed instructions (RVC) allowed in the HWLoop body. An illegal exception is raised otherwise.
- No uncoditional jump instructions allowed in the HWLoop body. An illegal exception is raised otherwise.
- No coditional branch instructions allowed in the HWLoop body. An illegal exception is raised otherwise.
- No privileged instructions (mret, dret, ecall, wfi) allowed in the HWLoop body, except for ebreak. An illegal exception is raised otherwise.
- No memory ordering instructions (fence, fence.i) allowed in the HWLoop body. An illegal exception is raised otherwise.
- The End address of the outermost HWLoop (#1) must be at least 2 instructions further than the End address innermost HWLoop (#0), i.e. HWLoop[1].endaddress >= HWLoop[0].endaddress + 8 An illegal exception is raised otherwise.

In order to use hardware loops, the compiler needs to setup the loop beforehand with the following instructions. Note that the minimum loop size is 3 instructions and the last instruction cannot be any jump or branch instruction.

For debugging and context switches, the hardware loop registers are mapped into the CSR address space and thus it is possible to read and write them via csrr and csrw instructions. Since hardware loop registers could be overwritten in when processing interrupts, the registers have to be saved in the interrupt routine together with the general purpose registers. The CS HWLoop registers are described in the *Control and Status Registers* section.

The CORE-V GCC compiler uses HWLoop automatically without the need of assembly. The mainline GCC does not generate any CORE-V instructions as for the other custom extensions.

Below an assembly code example of an nested HWLoop that computes a matrix addition.

```
asm volatile (
    ".option norvc;"
    "add %[j],x0, x0;"
    "add %[j],x0, x0;"
    "cv.count  x1, %[N];"
    "cv.endi   x1, endO;"
    "cv.starti x1, startO;"
        "startO:   cv.count  x0, %[N];"
        "cv.endi   x0, endZ;"
        "cv.starti x0, startZ;"
            "startZ: addi %[i], x0, 1;"
            "        addi %[i], x0, 1;"
            "endZ:   addi %[i], x0, 1;"
        "addi %[j],x0, 2;"
        "endO:   addi %[j], x0, 2;"
    : [i] "+r" (i), [j] "+r" (j)
    : [N] "r" (10)
);
```

At the beginning of the HWLoop, the registers %[i] and %[j] are 0. The innermost loop, from start0 to end0, adds to %[i] three times 1 and it is executed 10x10 times. Whereas the outermost loop, from startO to endO, executes 10 times the innermost loop and adds two times 2 to the register %[j]. At the end of the loop, the register %[i] contains 300 and the register %[j] contains 40.

CHAPTER

TWELVE

CONTROL AND STATUS REGISTERS

CV32E41P does not implement all control and status registers specified in the RISC-V privileged specifications, but is limited to the registers that were needed for the PULP system. The reason for this is that we wanted to keep the footprint of the core as low as possible and avoid any overhead that we do not explicitly need.

12.1 CSR Map

Table 12.1 lists all implemented CSRs. To columns in Table 12.1 may require additional explanation:

The **Parameter** column identifies those CSRs that are dependent on the value of specific compile/synthesis parameters. If these parameters are not set as indicated in Table 12.1 then the associated CSR is not implemented. If the parameter column is empty then the associated CSR is always implemented.

The **Privilege** column indicates the access mode of a CSR. The first letter indicates the lowest privilege level required to access the CSR. Attempts to access a CSR with a higher privilege level than the core is currently running in will throw an illegal instruction exception. This is largely a moot point for the CV32E41P as it only supports machine and debug modes. The remaining letters indicate the read and/or write behavior of the CSR when accessed by the indicated or higher privilge level:

- **RW**: CSR is **read-write**. That is, CSR instructions (e.g. csrrw) may write any value and that value will be returned on a subsequent read (unless a side-effect causes the core to change the CSR value).
- **RO**: CSR is **read-only**. Writes by CSR instructions raise an illegal instruction exception.

Writes of a non-supported value to **WLRL** bitfields of a **RW** CSR do not result in an illegal instruction exception. The exact bitfield access types, e.g. **WLRL** or **WARL**, can be found in the RISC-V privileged specification.

Reads or writes to a CSR that is not implemented will result in an illegal instruction exception.

Table 12.1: Control and Status Register Map

CSR Address	Name	Privilege	Parameter	Description
User CSRs				
0x001	`fflags`	URW	`FPU` = 1	Floating-point accrued exceptions.
0x002	`frm`	URW	`FPU` = 1	Floating-point dynamic rounding mode.
0x003	`fcsr`	URW	`FPU` = 1	Floating-point control and status register
0xC00	`cycle`	URO		(HPM) Cycle Counter
0xC02	`instret`	URO		(HPM) Instructions-Retired Counter
0xC03	`hpmcounter3`	URO		(HPM) Performance-Monitoring Counte
. . . . .				
0xC1F	`hpmcounter31`	URO		(HPM) Performance-Monitoring Counte
0xC80	`cycleh`	URO		(HPM) Upper 32 Cycle Counter

CO

Table 12.1 – continued from previous page

CSR Address	Name	Privilege	Parameter	Description
0xC82	instreth	URO		(HPM) Upper 32 Instructions-Retired C
0xC83	hpmcounterh3	URO		(HPM) Upper 32 Performance-Monitori
. . . . .				
0xC9F	hpmcounterh31	URO		(HPM) Upper 32 Performance-Monitori
User Custom CSRs				
0x800	lpstart0	URW	PULP_XPULP = 1	Hardware Loop 0 Start.
0x801	lpend0	URW	PULP_XPULP = 1	Hardware Loop 0 End.
0x802	lpcount0	URW	PULP_XPULP = 1	Hardware Loop 0 Counter.
0x804	lpstart1	URW	PULP_XPULP = 1	Hardware Loop 1 Start.
0x805	lpend1	URW	PULP_XPULP = 1	Hardware Loop 1 End.
0x806	lpcount1	URW	PULP_XPULP = 1	Hardware Loop 1 Counter.
0xCC0	uhartid	URO	PULP_XPULP = 1	Hardware Thread ID
0xCC1	privlv	URO	PULP_XPULP = 1	Privilege Level
Machine CSRs				
0x300	mstatus	MRW		Machine Status
0x301	misa	MRW		Machine ISA
0x304	mie	MRW		Machine Interrupt Enable Register
0x305	mtvec	MRW		Machine Trap-Handler Base Address
0x320	mcountinhibit	MRW		(HPM) Machine Counter-Inhibit Registe
0x323	mhpmevent3	MRW		(HPM) Machine Performance-Monitorir
. . . .				
0x33F	mhpmevent31	MRW		(HPM) Machine Performance-Monitorir
0x340	mscratch	MRW		Machine Scratch
0x341	mepc	MRW		Machine Exception Program Counter
0x342	mcause	MRW		Machine Trap Cause
0x343	mtval	MRW		Machine Trap Value
0x344	mip	MRW		Machine Interrupt Pending Register
0x7A0	tselect	MRW		Trigger Select Register
0x7A1	tdata1	MRW		Trigger Data Register 1
0x7A2	tdata2	MRW		Trigger Data Register 2
0x7A3	tdata3	MRW		Trigger Data Register 3
0x7A4	tinfo	MRO		Trigger Info
0x7A8	mcontext	MRW		Machine Context Register
0x7AA	scontext	MRW		Machine Context Register
0x7B0	dcsr	DRW		Debug Control and Status
0x7B1	dpc	DRW		Debug PC
0x7B2	dscratch0	DRW		Debug Scratch Register 0
0x7B3	dscratch1	DRW		Debug Scratch Register 1
0xB00	mcycle	MRW		(HPM) Machine Cycle Counter
0xB02	minstret	MRW		(HPM) Machine Instructions-Retired Co
0xB03	mhpmcounter3	MRW		(HPM) Machine Performance-Monitorir
. . . .				
0xB1F	mhpmcounter31	MRW		(HPM) Machine Performance-Monitorir
0xB80	mcycleh	MRW		(HPM) Upper 32 Machine Cycle Counte
0xB82	minstreth	MRW		(HPM) Upper 32 Machine Instructions-I
0xB83	mhpmcounterh3	MRW		(HPM) Upper 32 Machine Performance-
. . . .				
0xB9F	mhpmcounterh31	MRW		(HPM) Upper 32 Machine Performance-
0xF11	mvendorid	MRO		Machine Vendor ID

CO

Table 12.1 – continued from previous page

CSR Address	Name	Privilege	Parameter	Description
0xF12	`marchid`	MRO		Machine Architecture ID
0xF13	`mimpid`	MRO		Machine Implementation ID
0xF14	`mhartid`	MRO		Hardware Thread ID

12.2 CSR Descriptions

What follows is a detailed definition of each of the CSRs listed above. The **Mode** column defines the access mode behavior of each bit field when accessed by the privilege level specified in Table 12.1 (or a higher privilege level):

- **RO**: **read-only** fields are not affect by CSR write instructions. Such fields either return a fixed value, or a value determined by the operation of the core.
- **RW**: **read/write** fields store the value written by CSR writes. Subsequent reads return either the previously written value or a value determined by the operation of the core.

12.2.1 Floating-point accrued exceptions (`fflags`)

CSR Address: 0x001 (only present if `FPU` = 1)

Reset Value: 0x0000_0000

Bit #	Mode	Description
31:5	RO	Writes are ignored; reads return 0.
4	RW	NV- Invalid Operation
3	RW	DZ - Divide by Zero
2	RW	OF - Overflow
1	RW	UF - Underflow
0	RW	NX - Inexact

12.2.2 Floating-point dynamic rounding mode (`frm`)

CSR Address: 0x002 (only present if `FPU` = 1)

Reset Value: 0x0000_0000

Bit #	Mode	Description
31:3	RO	Writes are ignored; reads return 0.
2:0	RW	Rounding mode. 000 = RNE, 001 = RTZ, 010 = RDN, 011 = RUP, 100 = RMM 101 = Invalid, 110 = Invalid, 111 = DYN.

12.2.3 Floating-point control and status register (fcsr)

CSR Address: 0x003 (only present if FPU = 1)

Reset Value: 0x0000_0000

Bit #	Mode	Description
31:8	RO	Reserved. Writes are ignored; reads return 0.
7:5	RW	Rounding Mode (frm)
4:0	RW	Accrued Exceptions (fflags)

12.2.4 HWLoop Start Address 0/1 (lpstart0/1)

CSR Address: 0x800/0x804 (only present if PULP_XPULP = 1)

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RW	Start Address of the HWLoop 0/1.

12.2.5 HWLoop End Address 0/1 (lpend0/1)

CSR Address: 0x801/0x805 (only present if PULP_XPULP = 1)

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RW	End Address of the HWLoop 0/1.

12.2.6 HWLoop Count Address 0/1 (lpcount0/1)

CSR Address: 0x802/0x806 (only present if PULP_XPULP = 1)

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RW	Number of iteration of HWLoop 0/1.

12.2.7 Privilege Level (`privlv`)

CSR Address: 0xCC1 (only present if `PULP_XPULP` = 1)

Reset Value: 0x0000_0003

Table 12.2: PRIVLV

Bit #	Mode	Description
31:2	RO	Reads as 0.
1:0	RO	Current Privilege Level. 11 = Machine, 10 = Hypervisor, 01 = Supervisor, 00 = User. CV32E41P only supports Machine mode.

12.2.8 User Hardware Thread ID (`uhartid`)

CSR Address: 0xCC0 (only present if `PULP_XPULP` = 1)

Reset Value: Defined

Table 12.3: UHARTID

Bit #	Mode	Description
31:0	RO	Hardware Thread ID **hart_id_i**, see *Core Integration*

Similar to `mhartid` the `uhartid` provides the Hardware Thread ID. It differs from `mhartid` only in the required privilege level. On CV32E41P, as it is a machine mode only implementation, this difference is not noticeable.

12.2.9 Machine Status (`mstatus`)

CSR Address: 0x300

Reset Value: 0x0000_1800

Bit #	Mode	Description
31:18	RO	Reserved, hardwired to 0
17	RO	**MPRV:** hardwired to 0
16:13	RO	Unimplemented, hardwired to 0
12:11	RO	**MPP:** Machine Previous Priviledge mode, hardwired to 11 when the user mode is not enabled.
10:8	RO	Unimplemented, hardwired to 0
7	RO	**Previous Machine Interrupt Enable:** When an exception is encountered, MPIE will be set to MIE. When the mret instruction is executed, the value of MPIE will be stored to MIE.
6:5	RO	Unimplemented, hardwired to 0
4	RO	**Previous User Interrupt Enable:** If user mode is enabled, when an exception is encountered, UPIE will be set to UIE. When the uret instruction is executed, the value of UPIE will be stored to UIE.
3	RW	**Machine Interrupt Enable:** If you want to enable interrupt handling in your exception handler, set the Interrupt Enable MIE to 1 inside your handler code.
2:1	RO	Unimplemented, hardwired to 0
0	RO	**User Interrupt Enable:** If you want to enable user level interrupt handling in your exception handler, set the Interrupt Enable UIE to 1 inside your handler code.

12.2.10 Machine ISA (misa)

CSR Address: 0x301

Reset Value: defined

Detailed:

Bit #	Mode	Description
31:30	RO (0x1)	**MXL** (Machine XLEN).
23	RO	**X** (Non-standard extensions present).
12	RO (0x1)	**M** (Integer Multiply/Divide extension).
8	RO (0x1)	**I** (RV32I/64I/128I base ISA).
5	RO	**F** (Single-precision floating-point extension).
2	RO (0x1)	**C** (Compressed extension).
others	RO (0x0)	All other fields read as zero

All bitfields in the `misa` CSR read as 0 except for the following:

- **C** = 1
- **F** = 1 if `FPU` = 1 and `ZFINX` = 0
- **I** = 1
- **M** = 1
- **X** = 1 if `PULP_XPULP` = 1 or `PULP_CLUSTER` = 1
- **MXL** = 1 (i.e. XLEN = 32)

The bit positions are shown in the table above.

12.2.11 Machine Interrupt Enable Register (mie)

CSR Address: 0x304

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:16	RW	Machine Fast Interrupt Enables: Set bit x to enable interrupt irq_i[x].
11	RW	**Machine External Interrupt Enable (MEIE)**: If set, irq_i[11] is enabled.
7	RW	**Machine Timer Interrupt Enable (MTIE)**: If set, irq_i[7] is enabled.
3	RW	**Machine Software Interrupt Enable (MSIE)**: if set, irq_i[3] is enabled.

12.2.12 Machine Trap-Vector Base Address (mtvec)

CSR Address: 0x305

Reset Value: Defined

Detailed:

Bit #	Mode	Description
31 : 8	RW	BASE[31:8]: The trap-handler base address, always aligned to 256 bytes.
7 : 2	RO	BASE[7:2]: The trap-handler base address, always aligned to 256 bytes, i.e., mtvec[7:2] is always set to 0.
1	RO	MODE[1]: always 0
0	RW	MODE[0]: 0 = direct mode, 1 = vectored mode.

The initial value of `mtvec` is equal to {**mtvec_addr_i[31:8]**, 6'b0, 2'b01}.

When an exception or an interrupt is encountered, the core jumps to the corresponding handler using the content of the MTVEC[31:8] as base address. Only 8-byte aligned addresses are allowed. Both direct mode and vectored mode are supported.

12.2.13 Machine Counter-Inhibit Register (mcountinhibit)

CSR Address: 0x320

Reset Value: 0x0000_000D

The performance counter inhibit control register. The default value is to inihibit counters out of reset. The bit returns a read value of 0 for non implemented counters. This reset value shows the result using the default number of performance counters to be 1.

Detailed:

Bit#	Mode	Description
31:4	RW	Dependent on number of counters implemented in design parameter
3	RW	**selectors:** mhpmcounter3 inhibit
2	RW	minstret inhibit
1	RO	0
0	RW	mcycle inhibit

12.2.14 Machine Performance Monitoring Event Selector (mhpmevent3 .. mhpmevent31)

CSR Address: 0x323 - 0x33F

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:16	RO	0
15:0	RW	**selectors:** Each bit represent a unique event to count

The event selector fields are further described in Performance Counters section. Non implemented counters always return a read value of 0.

12.2.15 Machine Scratch (mscratch)

CSR Address: 0x340

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RW	Scratch value

12.2.16 Machine Exception PC (mepc)

CSR Address: 0x341

Reset Value: 0x0000_0000

Bit #	Mode	Description
31:1	RW	Machine Expection Program Counter 31:1
0	R0	Always 0

When an exception is encountered, the current program counter is saved in MEPC, and the core jumps to the exception address. When a mret instruction is executed, the value from MEPC replaces the current program counter.

12.2.17 Machine Cause (mcause)

CSR Address: 0x342

Reset Value: 0x0000_0000

Bit #	Mode	Description
31	RW	**Interrupt:** This bit is set when the exception was triggered by an interrupt.
30:5	RO (0)	Always 0
4:0	RW	**Exception Code** (See note below)

NOTE: software accesses to *mcause[4:0]* must be sensitive to the WLRL field specification of this CSR. For example, when *mcause[31]* is set, writing 0x1 to *mcause[1]* (Supervisor software interrupt) will result in UNDEFINED behavior.

12.2.18 Machine Trap Value (mtval)

CSR Address: 0x343

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RO (0)	Writes are ignored; reads return 0.

12.2.19 Machine Interrupt Pending Register (mip)

CSR Address: 0x344

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:16	RO	Machine Fast Interrupts Pending: If bit x is set, interrupt irq_i[x] is pending.
11	RO	**Machine External Interrupt Pending (MEIP)**: If set, irq_i[11] is pending.
7	RO	**Machine Timer Interrupt Pending (MTIP)**: If set, irq_i[7] is pending.
3	RO	**Machine Software Interrupt Pending (MSIP)**: if set, irq_i[3] is pending.

12.2.20 Trigger Select Register (tselect)

CSR Address: 0x7A0

Reset Value: 0x0000_0000

Accessible in Debug Mode or M-Mode.

Bit #	Mode	Description
31:0	RO	CV32E41P implements a single trigger, therefore this register will always read as zero

12.2.21 Trigger Data Register 1 (tdata1)

CSR Address: 0x7A1

Reset Value: 0x2800_1040

Accessible in Debug Mode or M-Mode. Since native triggers are not supported, writes to this register from M-Mode will be ignored.

Note: CV32E41P only implements one type of trigger, Match Control. Most fields of this register will read as a fixed value to reflect the single mode that is supported, in particular, instruction address match as described in the Debug Specification 0.13.2 section 5.2.2 & 5.2.9. The **type**, **dmode**, **hit**, **select**, **timing**, **sizelo**, **action**, **chain**, **match**, **m**, **s**, **u**, **store** and **load** bitfields of this CSR, which are marked as R/W in Debug Specification 0.13.2, are therefore implemented as WARL bitfields (corresponding to how these bitfields will be specified in the forthcoming Debug Specification 0.14.0).

Bit#	Mode	Description
31:28	RO (0x2)	**type:** 2 = Address/Data match trigger type.
27	RO (0x1)	**dmode:** 1 = Only debug mode can write tdata registers
26:21	RO (0x0)	**maskmax:** 0 = Only exact matching supported.
20	RO (0x0)	**hit:** 0 = Hit indication not supported.
19	RO (0x0)	**select:** 0 = Only address matching is supported.
18	RO (0x0)	**timing:** 0 = Break before the instruction at the specified address.
17:16	RO (0x0)	**sizelo:** 0 = Match accesses of any size.
15:12	RO (0x1)	**action:** 1 = Enter debug mode on match.
11	RO (0x0)	**chain:** 0 = Chaining not supported.
10:7	RO (0x0)	**match:** 0 = Match the whole address.
6	RO (0x1)	**m:** 1 = Match in M-Mode.
5	RO (0x0)	zero.
4	RO (0x0)	**s:** 0 = S-Mode not supported.
3	RO (0x0)	**u:** 0 = U-Mode not supported.
2	RW	**execute:** Enable matching on instruction address.
1	RO (0x0)	**store:** 0 = Store address / data matching not supported.
0	RO (0x0)	**load:** 0 = Load address / data matching not supported.

12.2.22 Trigger Data Register 2 (tdata2)

CSR Address: 0x7A2

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:0	RW	**data**

Accessible in Debug Mode or M-Mode. Since native triggers are not supported, writes to this register from M-Mode will be ignored. This register stores the instruction address to match against for a breakpoint trigger.

12.2.23 Trigger Data Register 3 (tdata3)

CSR Address: 0x7A3

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:0	RO	0

Accessible in Debug Mode or M-Mode. CV32E41P does not support the features requiring this register. Writes are ignored and reads will always return zero.

12.2.24 Trigger Info (tinfo)

CSR Address: 0x7A4

Reset Value: 0x0000_0004

Detailed:

Bit#	Mode	Description
31:16	RO (0x0)	0
15:0	RO (0x4)	**info**. Only type 2 is supported.

The **info** field contains one bit for each possible *type* enumerated in *tdata1*. Bit N corresponds to type N. If the bit is set, then that type is supported by the currently selected trigger. If the currently selected trigger does not exist, this field contains 1.

Accessible in Debug Mode or M-Mode.

12.2.25 Machine Context Register (mcontext)

CSR Address: 0x7A8

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:0	RO	0

Accessible in Debug Mode or M-Mode. CV32E41P does not support the features requiring this register. Writes are ignored and reads will always return zero.

12.2.26 Supervisor Context Register (scontext)

CSR Address: 0x7AA

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:0	RO	0

Accessible in Debug Mode or M-Mode. CV32E41P does not support the features requiring this register. Writes are ignored and reads will always return zero.

12.2.27 Debug Control and Status (dcsr)

CSR Address: 0x7B0

Reset Value: 0x4000_0003

Note: The **ebreaks**, **ebreaku** and **prv** bitfields of this CSR are marked as R/W in Debug Specification 0.13.2. However, as CV32E41P only supports machine mode, these bitfields are implemented as WARL bitfields (corresponding to how these bitfields will be specified in the forthcoming Debug Specification 0.14.0).

Detailed:

Bit #	Mode	Description
31:28	RO (0x4)	**xdebugver:** returns 4 - External debug support exists as it is described in this document.
27:16	RO (0x0)	Reserved
15	RW	**ebreakm**
14	RO (0x0)	Reserved
13	RO (0x0)	**ebreaks**. Always 0.
12	RO (0x0)	**ebreaku**. Always 0.
11	RW	**stepie**
10	RO (0x0)	**stopcount**. Always 0.
9	RO (0x0)	**stoptime**. Always 0.
8:6	RO	**cause**
5	RO (0x0)	Reserved
4	RO (0x0)	**mprven**. Always 0.
3	RO (0x0)	**nmip**. Always 0.
2	RW	**step**
1:0	RO (0x3)	**prv:** returns the current priviledge mode

12.2.28 Debug PC (dpc)

CSR Address: 0x7B1

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:1	RO	zero
0	RO	DPC

When the core enters in Debug Mode, DPC contains the virtual address of the next instruction to be executed.

12.2.29 Debug Scratch Register 0/1 (dscratch0/1)

CSR Address: 0x7B2/0x7B3

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RW	DSCRATCH0/1

12.2.30 Machine Cycle Counter (mcycle)

CSR Address: 0xB00

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:0	RW	The lower 32 bits of the 64 bit machine mode cycle counter.

12.2.31 Machine Instructions-Retired Counter (minstret)

CSR Address: 0xB02

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:0	RW	The lower 32 bits of the 64 bit machine mode instruction retired counter.

12.2.32 Machine Performance Monitoring Counter (mhpmcounter3 .. mhpmcounter31)

CSR Address: 0xB03 - 0xB1F

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:0	RW	Machine performance-monitoring counter

The lower 32 bits of the 64 bit machine performance-monitoring counter(s). The number of machine performance-monitoring counters is determined by the parameter NUM_MHPMCOUNTERS with a range from 0 to 29 (default value of 1). Non implemented counters always return a read value of 0.

12.2.33 Upper 32 Machine Cycle Counter (mcycleh)

CSR Address: 0xB80

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:0	RW	The upper 32 bits of the 64 bit machine mode cycle counter.

12.2.34 Upper 32 Machine Instructions-Retired Counter (minstreth)

CSR Address: 0xB82

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:0	RW	The upper 32 bits of the 64 bit machine mode instruction retired counter.

12.2.35 Upper 32 Machine Performance Monitoring Counter (mhpmcounter3h .. mhpmcounter31h)

CSR Address: 0xB83 - 0xB9F

Reset Value: 0x0000_0000

Detailed:

Bit#	Mode	Description
31:0	RW	Machine performance-monitoring counter

The upper 32 bits of the 64 bit machine performance-monitoring counter(s). The number of machine performance-monitoring counters is determined by the parameter `NUM_MHPMCOUNTERS` with a range from 0 to 29 (default value of 1). Non implemented counters always return a read value of 0.

12.2.36 Machine Vendor ID (mvendorid)

CSR Address: 0xF11

Reset Value: 0x0000_0602

Detailed:

Bit #	Mode	Description
31:7	RO	0xC. Number of continuation codes in JEDEC manufacturer ID.
6:0	RO	0x2. Final byte of JEDEC manufacturer ID, discarding the parity bit.

The `mvendorid` encodes the OpenHW JEDEC Manufacturer ID, which is 2 decimal (bank 13).

12.2.37 Machine Architecture ID (marchid)

CSR Address: 0xF12

Reset Value: 0x0000_0004

Detailed:

Bit #	Mode	Description
31:0	RO	Machine Architecture ID of CV32E41P is 4

12.2.38 Machine Implementation ID (mimpid)

CSR Address: 0xF13

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RO	Reads return 0.

12.2.39 Hardware Thread ID (mhartid)

CSR Address: 0xF14

Reset Value: Defined

Bit #	Mode	Description
31:0	RO	Hardware Thread ID **hart_id_i**, see *Core Integration*

NOTE: software accesses to *ucause[4:0]* must be sensitive to the WLRL field specification of this CSR. For example, when *ucause[31]* is set, writing 0x1 to *ucause[1]* (Supervisor software interrupt) will result in UNDEFINED behavior.

12.3 Cycle Counter (cycle)

CSR Address: 0xC00

Reset Value: 0x0000_0000

Detailed:

Bit#	R/W	Description
31:0	R	0

Read-only unprivileged shadow of the lower 32 bits of the 64 bit machine mode cycle counter.

12.4 Instructions-Retired Counter (instret)

CSR Address: 0xC02

Reset Value: 0x0000_0000

Detailed:

Bit#	R/W	Description
31:0	R	0

Read-only unprivileged shadow of the lower 32 bits of the 64 bit machine mode instruction retired counter.

12.5 Performance Monitoring Counter (hpmcounter3 .. hpmcounter31)

CSR Address: 0xC03 - 0xC1F

Reset Value: 0x0000_0000

Detailed:

Bit#	R/W	Description
31:0	R	0

Read-only unprivileged shadow of the lower 32 bits of the 64 bit machine mode performance counter. Non implemented counters always return a read value of 0.

12.6 Upper 32 Cycle Counter (cycleh)

CSR Address: 0xC80

Reset Value: 0x0000_0000

Detailed:

Bit#	R/W	Description
31:0	R	0

Read-only unprivileged shadow of the upper 32 bits of the 64 bit machine mode cycle counter.

12.7 Upper 32 Instructions-Retired Counter (instreth)

CSR Address: 0xC82

Reset Value: 0x0000_0000

Detailed:

Bit#	R/W	Description
31:0	R	0

Read-only unprivileged shadow of the upper 32 bits of the 64 bit machine mode instruction retired counter.

12.8 Upper 32 Performance Monitoring Counter (hpmcounter3h .. hpmcounter31h)

CSR Address: 0xC83 - 0xC9F

Reset Value: 0x0000_0000

Detailed:

Bit#	R/W	Description
31:0	R	0

Read-only unprivileged shadow of the upper 32 bits of the 64 bit machine mode performance counter. Non implemented counters always return a read value of 0.

CHAPTER

THIRTEEN

PERFORMANCE COUNTERS

CV32E41P implements performance counters according to the RISC-V Privileged Specification, version 1.11 (see Hardware Performance Monitor, Section 3.1.11). The performance counters are placed inside the Control and Status Registers (CSRs) and can be accessed with the `CSRRW(I)` and `CSRRS/C(I)` instructions.

CV32E41P implements the clock cycle counter `mcycle(h)`, the retired instruction counter `minstret(h)`, as well as the parameterizable number of event counters `mhpmcounter3(h)` - `mhpmcounter31(h)` and the corresponding event selector CSRs `mhpmevent3` - `mhpmevent31`, and the `mcountinhibit` CSR to individually enable/disable the counters. `mcycle(h)` and `minstret(h)` are always available.

All counters are 64 bit wide.

The number of event counters is determined by the parameter `NUM_MHPMCOUNTERS` with a range from 0 to 29 (default value of 1).

Unimplemented counters always read 0.

Note: All performance counters are using the gated version of `clk_i`. The **wfi** instruction, the **cv.elw** instruction, and `pulp_clock_en_i` impact the gating of `clk_i` as explained in *Sleep Unit* and can therefore affect the counters.

13.1 Event Selector

The following events can be monitored using the performance counters of CV32E41P.

Bit #	Event Name	
0	CYCLES	Number of cycles
1	INSTR	Number of instructions retired
2	LD_STALL	Number of load use hazards
3	JMP_STALL	Number of jump register hazards
4	IMISS	Cycles waiting for instruction fethces, excluding jumps and branches
5	LD	Number of load instructions
6	ST	Number of store instructions
7	JUMP	Number of jumps (unconditional)
8	BRANCH	Number of branches (conditional)
9	BRANCH_TAKEN	Number of branches taken (conditional)
10	COMP_INSTR	Number of compressed instructions retired
11	PIPE_STALL	Cycles from stalled pipeline
12	APU_TYPE	Numbe of type conflicts on APU/FP
13	APU_CONT	Number of contentions on APU/FP
14	APU_DEP	Number of dependency stall on APU/FP
15	APU_WB	Number of write backs on APUB/FP

The event selector CSRs `mhpmevent3` - `mhpmevent31` define which of these events are counted by the event counters `mhpmcounter3(h)` - `mhpmcounter31(h)`. If a specific bit in an event selector CSR is set to 1, this means that events with this ID are being counted by the counter associated with that selector CSR. If an event selector CSR is 0, this means that the corresponding counter is not counting any event.

Note: At most 1 bit should be set in an event selector. If multiple bits are set in an event selector, then the operation of the associated counter is undefined.

13.2 Controlling the counters from software

By default, all available counters are disabled after reset in order to provide the lowest power consumption.

They can be individually enabled/disabled by overwriting the corresponding bit in the `mcountinhibit` CSR at address `0x320` as described in the RISC-V Privileged Specification, version 1.11 (see Machine Counter-Inhibit CSR, Section 3.1.13). In particular, to enable/disable `mcycle(h)`, bit 0 must be written. For `minstret(h)`, it is bit 2. For event counter `mhpmcounterX(h)`, it is bit X.

The lower 32 bits of all counters can be accessed through the base register, whereas the upper 32 bits are accessed through the `h`-register. Reads of all these registers are non-destructive.

13.3 Parametrization at synthesis time

The `mcycle(h)` and `minstret(h)` counters are always available and 64 bit wide.

The number of available event counters `mhpmcounterX(h)` can be controlled via the `NUM_MHPMCOUNTERS` parameter. By default `NUM_MHPCOUNTERS` set to 1.

An increment of 1 to the NUM_MHPCOUNTERS results in the addition of the following:

- 64 flops for `mhpmcounterX`
- 15 flops for *mhpmeventX*

- 1 flop for *mcountinhibit[X]*
- Adder and event enablement logic

13.4 Time Registers (`time(h)`)

The user mode `time(h)` registers are not implemented. Any access to these registers will cause an illegal instruction trap. It is recommended that a software trap handler is implemented to detect access of these CSRs and convert that into access of the platform-defined `mtime` register (if implemented in the platform).

CHAPTER

FOURTEEN

EXCEPTIONS AND INTERRUPTS

CV32E41P implements trap handling for interrupts and exceptions according to the RISC-V Privileged Specification, version 1.11. The `irq_i[31:16]` interrupts are a custom extension.

When entering an interrupt/exception handler, the core sets the `mepc` CSR to the current program counter and saves `mstatus`.MIE to `mstatus`.MPIE. All exceptions cause the core to jump to the base address of the vector table in the `mtvec` CSR. Interrupts are handled in either direct mode or vectored mode depending on the value of `mtvec`.MODE. In direct mode the core jumps to the base address of the vector table in the `mtvec` CSR. In vectored mode the core jumps to the base address plus four times the interrupt ID. Upon executing an MRET instruction, the core jumps to the program counter previously saved in the `mepc` CSR and restores `mstatus`.MPIE to `mstatus`.MIE.

The base address of the vector table must be aligned to 256 bytes (i.e., its least significant byte must be 0x00) and can be programmed by writing to the `mtvec` CSR. For more information, see the *Control and Status Registers* documentation.

The core starts fetching at the address defined by `boot_addr_i`. It is assumed that the boot address is supplied via a register to avoid long paths to the instruction fetch unit.

14.1 Interrupt Interface

Table 14.1 describes the interrupt interface.

Table 14.1: Interrupt interface signals

Signal	Direction	Description
`irq_i[31:0]`	Input	Active high, level sensistive interrupt inputs. Not all interrupt inputs can be used on CV32E41P. Specifically irq_i[15:12], irq_i[10:8], irq_i[6:4] and irq_i[2:0] shall be tied to 0 externally as they are reserved for future standard use (or for cores which are not Machine mode only) in the RISC-V Privileged specification. irq_i[11], irq_i[7], and irq_i[3] correspond to the Machine External Interrupt (MEI), Machine Timer Interrupt (MTI), and Machine Software Interrupt (MSI) respectively. The irq_i[31:16] interrupts are a CV32E41P specific extension to the RISC-V Basic (a.k.a. CLINT) interrupt scheme.
`irq_ack_o`	Output	Interrupt acknowledge. Set to 1 for one cycle when the interrupt with ID `irq_id_o[4:0]` is taken.
`irq_id_o[4:0]`	Output	Interrupt index for taken interrupt. Only valid when `irq_ack_o` = 1.

14.2 Interrupts

The `irq_i[31:0]` interrupts are controlled via the `mstatus`, `mie` and `mip` CSRs. CV32E41P uses the upper 16 bits of `mie` and `mip` for custom interrupts (`irq_i[31:16]`), which reflects an intended custom extension in the RISC-V Basic (a.k.a. CLINT) interrupt architecture. After reset, all interrupts are disabled. To enable interrupts, both the global interrupt enable (MIE) bit in the `mstatus` CSR and the corresponding individual interrupt enable bit in the `mie` CSR need to be set. For more information, see the *Control and Status Registers* documentation.

If multiple interrupts are pending, they are handled in the fixed priority order defined by the RISC-V Privileged Specification, version 1.11 (see Machine Interrupt Registers, Section 3.1.9). The highest priority is given to the interrupt with the highest ID, except for the Machine Timer Interrupt, which has the lowest priority. So from high to low priority the interrupts are ordered as follows: `irq_i[31]`, `irq_i[30]`, ..., `irq_i[16]`, `irq_i[11]`, `irq_i[3]`, `irq_i[7]`.

All interrupt lines are level-sensitive. There are two supported mechanisms by which interrupts can be cleared at the external source.

- A software-based mechanism in which the interrupt handler signals completion of the handling routine to the interrupt source, e.g., through a memory-mapped register, which then deasserts the corresponding interrupt line.
- A hardware-based mechanism in which the `irq_ack_o` and `irq_id_o[4:0]` signals are used to clear the interrupt sourcee, e.g. by an external interrupt controller. `irq_ack_o` is a 1 `clk_i` cycle pulse during which `irq_id_o[4:0]` reflects the index in `irq_id[]` of the taken interrupt.

In Debug Mode, all interrupts are ignored independent of `mstatus`.MIE and the content of the `mie` CSR.

14.3 Exceptions

CV32E41P can trigger an exception due to the following exception causes:

Exception Code	Description
2	Illegal instruction
3	Breakpoint
11	Environment call from M-Mode (ECALL)

The illegal instruction exception and M-Mode ECALL instruction exceptions cannot be disabled and are always active. The core raises an illegal instruction exception for any instruction in the RISC-V privileged and unprivileged specifications that is explicitly defined as being illegal according to the ISA implemented by the core, as well as for any instruction that is left undefined in these specifications unless the instruction encoding is configured as a custom CV32E41P instruction for specific parameter settings as defined in (see :ref:custom-isa-extensions). For example, in case the parameter FPU is set to 0, the CV32E41P raises an illegal instruction exception for any RVF instruction. The same concerns for XPULP extensions everytime the parameter PULP_XPULP is set to 0 (see :ref:core-integration).

14.4 Nested Interrupt/Exception Handling

CV32E41P does support nested interrupt/exception handling in software. The hardware automatically disables interrupts upon entering an interrupt/exception handler. Otherwise, interrupts/exceptions during the critical part of the handler, i.e. before software has saved the `mepc` and `mstatus` CSRs, would cause those CSRs to be overwritten. If desired, software can explicitly enable interrupts by setting `mstatus`.MIE to 1 from within the handler. However, software should only do this after saving `mepc` and `mstatus`. There is no limit on the maximum number of nested interrupts. Note that, after enabling interrupts by setting `mstatus`.MIE to 1, the current handler will be interrupted also by lower priority interrupts. To allow higher priority interrupts only, the handler must configure `mie` accordingly.

The following pseudo-code snippet visualizes how to perform nested interrupt handling in software.

```
isr_handle_nested_interrupts(id) {
  // Save mpec and mstatus to stack
  mepc_bak = mepc;
  mstatus_bak = mstatus;

  // Save mie to stack (optional)
  mie_bak = mie;

  // Keep lower-priority interrupts disabled (optional)
  mie = mie & ~((1 << (id + 1)) - 1);

  // Re-enable interrupts
  mstatus.MIE = 1;

  // Handle interrupt
  // This code block can be interrupted by other interrupts.
  // ...

  // Restore mstatus (this disables interrupts) and mepc
  mstatus = mstatus_bak;
  mepc = mepc_bak;

  // Restore mie (optional)
  mie = mie_bak;
}
```

Nesting of interrupts/exceptions in hardware is not supported.

CHAPTER

FIFTEEN

DEBUG & TRIGGER

CV32E41P offers support for execution-based debug according to the RISC-V Debug Specification, version 0.13.2. The main requirements for the core are described in Chapter 4: RISC-V Debug, Chapter 5: Trigger Module, and Appendix A.2: Execution Based.

The following list shows the simplified overview of events that occur in the core when debug is requested:

1. Enters Debug Mode
2. Saves the PC to DPC
3. Updates the cause in the DCSR
4. Points the PC to the location determined by the input port dm_haltaddr_i
5. Begins executing debug control code.

Debug Mode can be entered by one of the following conditions:

- External debug event using the debug_req_i signal
- Trigger Module match event
- ebreak instruction when not in Debug Mode and when DCSR.EBREAKM == 1 (see *EBREAK Behavior* below)

A user wishing to perform an abstract access, whereby the user can observe or control a core's GPR or CSR register from the hart, is done by invoking debug control code to move values to and from internal registers to an externally addressable Debug Module (DM). Using this execution-based debug allows for the reduction of the overall number of debug interface signals.

Note: Debug support in CV32E41P is only one of the components needed to build a System on Chip design with run-control debug support (think "the ability to attach GDB to a core over JTAG"). Additionally, a Debug Module and a Debug Transport Module, compliant with the RISC-V Debug Specification, are needed.

A supported open source implementation of these building blocks can be found in the RISC-V Debug Support for PULP Cores IP block.

The CV3240P also supports a Trigger Module to enable entry into Debug Mode on a trigger event with the following features:

- Number of trigger register(s) : 1
- Supported trigger types: instruction address match (Match Control)

The CV32E41P will not support the optional debug features 10, 11, & 12 listed in Section 4.1 of the RISC-V Debug Specification. Specifically, a control transfer instruction's destination location being in or out of the Program Buffer and instructions depending on PC value shall **not** cause an illegal instruction.

15.1 Interface

Signal	Direction	Description
debug_req_i	input	Request to enter Debug Mode
debug_havereset_o	output	Debug status: Core has been reset
debug_running_o	output	Debug status: Core is running
debug_halted_o	output	Debug status: Core is halted
dm_halt_addr_i[31:0]	input	Address for debugger entry
dm_exception_addr_i[31:0]	input	Address for debugger exception entry

debug_req_i is the "debug interrupt", issued by the debug module when the core should enter Debug Mode. The debug_req_i is synchronous to clk_i and requires a minimum assertion of one clock period to enter Debug Mode. The instruction being decoded during the same cycle that debug_req_i is first asserted shall not be executed before entering Debug Mode.

debug_havereset_o, debug_running_o, and debug_mode_o signals provide the operational status of the core to the debug module. The assertion of these signals is mutually exclusive.

debug_havereset_o is used to signal that the CV32E41P has been reset. debug_havereset_o is set high during the assertion of rst_ni. It will be cleared low a few (unspecified) cycles after rst_ni has been deasserted **and** fetch_enable_i has been sampled high.

debug_running_o is used to signal that the CV32E41P is running normally.

debug_halted_o is used to signal that the CV32E41P is in debug mode.

dm_halt_addr_i is the address where the PC jumps to for a debug entry event. When in Debug Mode, an ebreak instruction will also cause the PC to jump back to this address without affecting status registers. (see *EBREAK Behavior* below)

dm_exception_addr_i is the address where the PC jumps to when an exception occurs during Debug Mode. When in Debug Mode, the mret or uret instruction will also cause the PC to jump back to this address without affecting status registers.

Both dm_halt_addr_i and dm_exception_addr_i must be word aligned.

15.2 Core Debug Registers

CV32E41P implements four core debug registers, namely *Debug Control and Status (dcsr)*, *Debug PC (dpc)*, and two debug scratch registers. Access to these registers in non Debug Mode results in an illegal instruction.

Several trigger registers are required to adhere to specification. The following are the most relevant: *Trigger Select Register (tselect)*, *Trigger Data Register 1 (tdata1)*, *Trigger Data Register 2 (tdata2)* and *Trigger Info (tinfo)*

The TDATA1.DMODE is hardwired to a value of 1. In non Debug Mode, writes to Trigger registers are ignored and reads reflect CSR values.

15.3 Debug state

As specified in RISC-V Debug Specification every hart that can be selected by the Debug Module is in exactly one of four states: `nonexistent`, `unavailable`, `running` or `halted`.

The remainder of this section assumes that the CV32E41P will not be classified as `nonexistent` by the integrator.

The CV32E41P signals to the Debug Module whether it is `running` or `halted` via its `debug_running_o` and `debug_halted_o` pins respectively. Therefore, assuming that this core will not be integrated as a `nonexistent` core, the CV32E41P is classified as `unavailable` when neither `debug_running_o` or `debug_halted_o` is asserted. Upon `rst_ni` assertion the debug state will be `unavailable` until some cycle(s) after `rst_ni` has been deasserted and `fetch_enable_i` has been sampled high. After this point (until a next reset assertion) the core will transition between having its `debug_halted_o` or `debug_running_o` pin asserted depending whether the core is in debug mode or not. Exactly one of the `debug_havereset_o`, `debug_running_o`, `debug_halted_o` is asserted at all times.

Figure 15.1 and show Figure 15.2 show typical examples of transitioning into the `running` and `halted` states.

Figure 15.1: Transition into debug `running` state

Figure 15.2: Transition into debug `halted` state

The key properties of the debug states are:

- The CV32E41P can remain in its `unavailable` state for an arbitrarily long time (depending on `rst_ni` and `fetch_enable_i`).
- If `debug_req_i` is asserted after `rst_ni` deassertion and before or coincident with the assertion of `fetch_enable_i`, then the CV32E41P is guaranteed to transition straight from its `unavailable` state into its `halted` state. If `debug_req_i` is asserted at a later point in time, then the CV32E41P might transition through the `running` state on its ways to the `halted` state.
- If `debug_req_i` is asserted during the `running` state, the core will eventually transition into the `halted` state (typically after a couple of cycles).

15.4 EBREAK Behavior

The EBREAK instruction description is distributed across several RISC-V specifications: RISC-V Debug Specification, RISC-V Priveleged Specification, RISC-V ISA. The following is a summary of the behavior for three common scenarios.

15.4.1 Scenario 1 : Enter Exception

Executing the EBREAK instruction when the core is **not** in Debug Mode and the DCSR.EBREAKM == 0 shall result in the following actions:

- The core enters the exception handler routine located at MTVEC (Debug Mode is not entered)
- MEPC & MCAUSE are updated

To properly return from the exception, the ebreak handler will need to increment the MEPC to the next instruction. This requires querying the size of the ebreak instruction that was used to enter the exception (16 bit c.ebreak or 32 bit ebreak).

Note: The CV32E41P does not support MTVAL CSR register which would have saved the value of the instruction for exceptions. This may be supported on a future core.

15.4.2 Scenario 2 : Enter Debug Mode

Executing the EBREAK instruction when the core is **not** in Debug Mode and the DCSR.EBREAKM == 1 shall result in the following actions:

- The core enters Debug Mode and starts executing debug code located at `dm_halt_addr_i` (exception routine not called)
- DPC & DCSR are updated

Similar to the exception scenario above, the debugger will need to increment the DPC to the next instruction before returning from Debug Mode.

Note: The default value of DCSR.EBREAKM is 0 and the DCSR is only accessible in Debug Mode. To enter Debug Mode from EBREAK, the user will first need to enter Debug Mode through some other means, such as from the external ``debug_req_i``, and set DCSR.EBREAKM.

15.4.3 Scenario 3 : Exit Program Buffer & Restart Debug Code

Execuitng the EBREAK instruction when the core is in Debug Mode shall result in the following actions:

- The core remains in Debug Mode and execution jumps back to the beginning of the debug code located at `dm_halt_addr_i`
- none of the CSRs are modified

CHAPTER

SIXTEEN

TRACER

The module `cv32e41p_tracer` can be used to create a log of the executed instructions. It is a behavioral, non-synthesizable, module instantiated in the example testbench that is provided for the `cv32e41p_core`. It can be enabled during simulation by defining **CV32E41P_TRACE_EXECUTION**.

16.1 Output file

All traced instructions are written to a log file. The log file is named `trace_core_<HARTID>.log`, with `<HARTID>` being the 32 digit hart ID of the core being traced.

16.2 Trace output format

The trace output is in tab-separated columns.

1. **Time**: The current simulation time.
2. **Cycle**: The number of cycles since the last reset.
3. **PC**: The program counter
4. **Instr**: The executed instruction (base 16). 32 bit wide instructions (8 hex digits) are uncompressed instructions, 16 bit wide instructions (4 hex digits) are compressed instructions.
5. **Decoded instruction**: The decoded (disassembled) instruction in a format equal to what objdump produces when calling it like `objdump -Mnumeric -Mno-aliases -D`. - Unsigned numbers are given in hex (prefixed with `0x`), signed numbers are given as decimal numbers. - Numeric register names are used (e.g. `x1`). - Symbolic CSR names are used. - Jump/branch targets are given as absolute address if possible (PC + immediate).
6. **Register and memory contents**: For all accessed registers, the value before and after the instruction execution is given. Writes to registers are indicated as `registername=value`, reads as `registername:value`. For memory accesses, the address and the loaded and stored data are given.

```
Time          Cycle      PC       Instr    Decoded instruction Register and memory␣
↪contents
          130         61 00000150 4481     c.li    x9,0        x9=0x00000000
          132         62 00000152 00008437 lui     x8,0x8      x8=0x00008000
          134         63 00000156 fff40413 addi    x8,x8,-1    x8:0x00008000 ␣
↪x8=0x00007fff
          136         64 0000015a 8c65     c.and   x8,x9       x8:0x00007fff ␣
↪x9:0x00000000  x8=0x00000000
          142         67 0000015c c622     c.swsp  x8,12(x2)   x2:0x00002000 ␣
↪x8:0x00000000 PA:0x0000200c store:0x00000000  load:0xffffffff
```

(continues on next page)

(continued from previous page)

CHAPTER

SEVENTEEN

CORE-V INSTRUCTION SET EXTENSIONS

CV32E41P supports the following CORE-V ISA Extensions, which are part of **Xcorev** and can be enabled by setting `PULP_XPULP` == 1.

- Post-Incrementing load and stores, see *Post-Incrementing Load & Store Instructions and Register-Register Load & Store Instructions*.
- Hardware Loop extension, see *Hardware Loops*.
- ALU extensions, see *ALU*.
- Multiply-Accumulate extensions, see *Multiply-Accumulate*.
- Optional support for Hardware Loops, see *SIMD*.

Additionally the event load instruction (**cv.elw**) is supported by setting `PULP_CLUSTER` == 1.

To use such instructions, you need to compile your SW with the CORE-V GCC compiler.

If not specified, all the operands are signed and immediate values are sign-extended.

17.1 Post-Incrementing Load & Store Instructions and Register-Register Load & Store Instructions

Post-Incrementing load and store instructions perform a load, or a store, respectively, while at the same time incrementing the address that was used for the memory access. Since it is a post-incrementing scheme, the base address is used for the access and the modified address is written back to the register-file. There are versions of those instructions that use immediates and those that use registers as offsets. The base address always comes from a register.

The custom post-incrementing load & store instructions and register-register load & store instructions are only supported if `PULP_XPULP` == 1.

17.1.1 Load Operations

Mnemonic	Description
Register-Immediate Loads with Post-Increment	
cv.lb rD, Imm(rs1!)	rD = Sext(Mem8(rs1)) rs1 += Imm[11:0]
cv.lbu rD, Imm(rs1!)	rD = Zext(Mem8(rs1)) rs1 += Imm[11:0]
cv.lh rD, Imm(rs1!)	rD = Sext(Mem16(rs1)) rs1 += Imm[11:0]
cv.lhu rD, Imm(rs1!)	rD = Zext(Mem16(rs1)) rs1 += Imm[11:0]
cv.lw rD, Imm(rs1!)	rD = Mem32(rs1) rs1 += Imm[11:0]
Register-Register Loads with Post-Increment	
cv.lb rD, rs2(rs1!)	rD = Sext(Mem8(rs1)) rs1 += rs2
cv.lbu rD, rs2(rs1!)	rD = Zext(Mem8(rs1)) rs1 += rs2
cv.lh rD, rs2(rs1!)	rD = Sext(Mem16(rs1)) rs1 += rs2
cv.lhu rD, rs2(rs1!)	rD = Zext(Mem16(rs1)) rs1 += rs2
cv.lw rD, rs2(rs1!)	rD = Mem32(rs1) rs1 += rs2
Register-Register Loads	
cv.lb rD, rs2(rs1)	rD = Sext(Mem8(rs1 + rs2))
cv.lbu rD, rs2(rs1)	rD = Zext(Mem8(rs1 + rs2))
cv.lh rD, rs2(rs1)	rD = Sext(Mem16(rs1 + rs2))
cv.lhu rD, rs2(rs1)	rD = Zext(Mem16(rs1 + rs2))
cv.lw rD, rs2(rs1)	rD = Mem32(rs1 + rs2)

17.1.2 Store Operations

Mnemonic	Description
Register-Immediate Stores with Post-Increment	
cv.sb rs2, Imm(rs1!)	Mem8(rs1) = rs2 rs1 += Imm[11:0]
cv.sh rs2, Imm(rs1!)	Mem16(rs1) = rs2 rs1 += Imm[11:0]
cv.sw rs2, Imm(rs1!)	Mem32(rs1) = rs2 rs1 += Imm[11:0]
Register-Register Stores with Post-Increment	
cv.sb rs2, rs3(rs1!)	Mem8(rs1) = rs2 rs1 += rs3
cv.sh rs2, rs3(rs1!)	Mem16(rs1) = rs2 rs1 += rs3
cv.sw rs2, rs3(rs1!)	Mem32(rs1) = rs2 rs1 += rs3
Register-Register Stores	
cv.sb rs2, rs3(rs1)	Mem8(rs1 + rs3) = rs2
cv.sh rs2 rs3(rs1)	Mem16(rs1 + rs3) = rs2
cv.sw rs2, rs3(rs1)	Mem32(rs1 + rs3) = rs2

Encoding

31 : 20	19 :15	14 : 12	11 :07	06 : 00	
imm[11:0]	rs1	funct3	rd	opcode	Mnemonic
offset	base	000	dest	000 1011	**cv.lb rD, Imm(rs1!)**
offset	base	100	dest	000 1011	**cv.lbu rD, Imm(rs1!)**
offset	base	001	dest	000 1011	**cv.lh rD, Imm(rs1!)**
offset	base	101	dest	000 1011	**cv.lhu rD, Imm(rs1!)**
offset	base	010	dest	000 1011	**cv.lw rD, Imm(rs1!)**

31 : 25	24 : 20	19 :15	14 : 12	11 :07	06 : 00	
funct7	rs2	rs1	funct3	rd	opcode	Mnemonic
000 0000	offset	base	111	dest	000 1011	**cv.lb rD, rs2(rs1!)**
010 0000	offset	base	111	dest	000 1011	**cv.lbu rD, rs2(rs1!)**
000 1000	offset	base	111	dest	000 1011	**cv.lh rD, rs2(rs1!)**
010 1000	offset	base	111	dest	000 1011	**cv.lhu rD, rs2(rs1!)**
001 0000	offset	base	111	dest	000 1011	**cv.lw rD, rs2(rs1!)**

31 : 25	24 : 20	19 :15	14 : 12	11 :07	06 : 00	
funct7	rs2	rs1	funct3	rd	opcode	Mnemonic
000 0000	offset	base	111	dest	000 0011	**cv.lb rD, rs2(rs1)**
010 0000	offset	base	111	dest	000 0011	**cv.lbu rD, rs2(rs1)**
000 1000	offset	base	111	dest	000 0011	**cv.lh rD, rs2(rs1)**
010 1000	offset	base	111	dest	000 0011	**cv.lhu rD, rs2(rs1)**
001 0000	offset	base	111	dest	000 0011	**cv.lw rD, rs2(rs1)**

31 : 25	24:20	19 :15	14 : 12	11 : 07	06 : 00	
imm[11:5]	rs2	rs1	funct3	rd	opcode	Mnemonic
offset[11:5]	src	base	000	offset[4:0]	010 1011	**cv.sb rs2, Imm(rs1!)**
offset[11:5]	src	base	001	offset[4:0]	010 1011	**cv.sh rs2, Imm(rs1!)**
offset[11:5]	src	base	010	offset[4:0]	010 1011	**cv.sw rs2, Imm(rs1!)**

31 : 25	24 : 20	19 :15	14 : 12	11 :07	06 : 00	
funct7	rs2	rs1	funct3	rd	opcode	Mnemonic
000 0000	src	base	100	offset	010 1011	**cv.sb rs2, rs3(rs1!)**
000 0000	src	base	101	offset	010 1011	**cv.sh rs2, rs3(rs1!)**
000 0000	src	base	110	offset	010 1011	**cv.sw rs2, rs3(rs1!)**

31 : 25	24 : 20	19 :15	14 : 12	11 :07	06 : 00	
funct7	rs2	rs1	funct3	rs3	opcode	Mnemonic
000 0000	src	base	100	offset	010 0011	**cv.sb rs2, rs3(rs1)**
000 0000	src	base	101	offset	010 0011	**cv.sh rs2, rs3(rs1)**
000 0000	src	base	110	offset	010 0011	**cv.sw rs2, rs3(rs1)**

17.2 Event Load Instructions

The event load instruction **cv.elw** is only supported if the PULP_CLUSTER parameter is set to 1. The event load performs a load word and can cause the CV32E41P to enter a sleep state as explained in *PULP Cluster Extension*.

17.2.1 Load Operations

Mnemonic	Description
Event Load	
cv.elw rD, Imm(rs1)	rD = Mem32(Sext(Imm)+rs1)

Encoding

31 : 20	19 :15	14 : 12	11 :07	06 : 00	
imm[11:0]	rs1	funct3	rd	opcode	Mnemonic
offset	base	110	dest	000 0011	**cv.elw rD, Imm(rs1)**

17.3 Hardware Loops

CV32E41P supports 2 levels of nested hardware loops. The loop has to be setup before entering the loop body. For this purpose, there are two methods, either the long commands that separately set start- and end-addresses of the loop and the number of iterations, or the short command that does all of this in a single instruction. The short command has a limited range for the number of instructions contained in the loop and the loop must start in the next instruction after the setup instruction.

Hardware loop instructions and related CSRs are only supported if `PULP_XPULP` == 1.

Details about the hardware loop constraints are provided in *CORE-V Hardware Loop Extensions*.

In the following tables, the hardware loop instructions are reported. In assembly, **L** is referred by x0 or x1.

17.3.1 Operations

Long Hardware Loop Setup instructions

Mnemonic	Description	
cv.starti	**L, uimmL**	lpstart[L] = PC + (uimmL << 1)
cv.endi	**L, uimmL**	lpend[L] = PC + (uimmL << 1)
cv.count	**L, rs1**	lpcount[L] = rs1
cv.counti	**L, uimmL**	lpcount[L] = uimmL

Short Hardware Loop Setup Instructions

Mnemonic	Description	
cv.setup	**L, rs1, uimmL**	lpstart[L] = pc + 4 lpend[L] = pc + (uimmL << 1) lpcount[L] = rs1
cv.setupi	**L, uimmL, uimmS**	lpstart[L] = pc + 4 lpend[L] = pc + (uimmS << 1) lpcount[L] = uimmL

Encoding

31 : 20	19 :15	14 : 12	11 :08	07	06 : 00	
uimmL[11:0]	rs1	funct3	rd	L	opcode	Mnemonic
uimmL[11:0]	00000	000	0000	L	111 1011	**cv.starti L, uimmL**
uimmL[11:0]	00000	001	0000	L	111 1011	**cv.endi L, uimmL**
0000 0000 0000	src1	010	0000	L	111 1011	**cv.count L, rs1**
uimmL[11:0]	00000	011	0000	L	111 1011	**cv.counti L, uimmL**
uimmL[11:0]	src1	100	0000	L	111 1011	**cv.setup L, rs1, uimmL**
uimmL[11:0]	uimmS[4:0]	101	0000	L	111 1011	**cv.setupi L, uimmL, uimmS**

17.4 ALU

CV32E41P supports advanced ALU operations that allow to perform multiple instructions that are specified in the base instruction set in one single instruction and thus increases efficiency of the core. For example, those instructions include zero-/sign-extension instructions for 8-bit and 16-bit operands, simple bit manipulation/counting instructions and min/max/avg instructions. The ALU does also support saturating, clipping, and normalizing instructions which make fixed-point arithmetic more efficient.

The custom ALU extensions are only supported if `PULP_XPULP` == 1.

Bit manipulation is not supported by the compiler tool chain.

The custom extensions to the ALU are split into several subgroups that belong together.

- Bit manipulation instructions are useful to work on single bits or groups of bits within a word, see *Bit Manipulation Operations*.
- General ALU instructions try to fuse common used sequences into a single instruction and thus increase the performance of small kernels that use those sequence, see *General ALU Operations*.
- Immediate branching instructions are useful to compare a register with an immediate value before taking or not a branch, see see *Immediate Branching Operations*.

Extract, Insert, Clear and Set instructions have the following meaning:

- Extract Is3+1 or rs2[9:5]+1 bits from position Is2 or rs2[4:0] [and sign extend it]
- Insert Is3+1 or rs2[9:5]+1 bits at position Is2 or rs2[4:0]
- Clear Is3+1 or rs2[9:5]+1 bits at position Is2 or rs2[4:0]
- Set Is3+1 or rs2[9:5]+1 bits at position Is2 or rs2[4:0]

17.4.1 Bit Reverse Instruction

This section will describe the *cv.bitrev* instruction from a bit manipulation perspective without describing it's application as part of an FFT. The bit reverse instruction will reverse bits in groupings of 1, 2 or 3 bits. The number of grouped bits is described by *Is3* as follows:

- **0** - reverse single bits
- **1** - reverse groups of 2 bits
- **2** - reverse groups of 3 bits

The number of bits that are reversed can be controlled by *Is2*. This will specify the number of bits that will be removed by a left shift prior to the reverse operation resulting in the *32-Is2* least significant bits of the input value being reversed and the *Is2* most significant bits of the input value being thrown out.

What follows is a few examples.

```
cv.bitrev x18, x20, 0, 4 (groups of 1 bit; radix-2)

in:    0xC64A5933 11000110010010100101100100110011
shift: 0x64A59330 01100100101001011001001100110000
out:   0x0CC9A526 00001100110010011010010100100110

Swap pattern:
A B C D E F G H . . . . . . . . . . . . . . . . . . . . . . . .
0 1 1 0 0 1 0 0 1 0 1 0 0 1 0 1 1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 0
```

(continues on next page)

(continued from previous page)

```
. . . . . . . . . . . . . . . . . . . . . . . . H G F E D C B A
0 0 0 0 1 1 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 1 0 1 0 0 1 0 0 1 1 0
```

In this example the input value is first shifted by 4 (*Is2*). Each individual bit is reversed. For example, bits 31 and 0 are swapped, 30 and 1, etc.

```
cv.bitrev x18, x20, 1, 4 (groups of 2 bits; radix-4)

in:    0xC64A5933 11000110010010100101100100110011
shift: 0x64A59330 01100100101001011001001100110000
out:   0x0CC65A19 00001100110001100101101000011001

Swap pattern:
A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P
01 10 01 00 10 10 01 01 10 01 00 11 00 11 00 00
P  O  N  M  L  K  J  I  H  G  F  E  D  C  B  A
00 00 11 00 11 00 01 10 01 01 10 10 00 01 10 01
```

In this example the input value is first shifted by 4 (*Is2*). Each group of two bits are reversed. For example, bits 31 and 30 are swapped with 1 and 0 (retaining their position relative to each other), bits 29 and 28 are swapped with 3 and 2, etc.

```
cv.bitrev x18, x20, 2, 4 (groups of 3 bits; radix-8)

in:    0xC64A5933 11000110010010100101100100110011
shift: 0x64A59330 01100100101001011001001100110000
out:   0x216B244B 00100000101101011001001000 1001011

Swap pattern:
A   B   C   D   E   F   G   H   I   J
011 001 001 010 010 110 010 011 001 100 00
   J   I   H   G   F   E   D   C   B   A
00 100 001 011 010 110 010 010 001 001 011
```

In this last example the input value is first shifted by 4 (*Is2*). Each group of three bits are reversed. For example, bits 31, 30 and 29 are swapped with 4, 3 and 2 (retaining their position relative to each other), bits 28, 27 and 26 are swapped with 7, 6 and 5, etc. Notice in this example that bits 0 and 1 are lost and the result is shifted right by two with bits 31 and 30 being tied to zero. Also notice that when J (100) is swapped with A (011), the four most significant bits are no longer zero as in the other cases. This may not be desirable if the intention is to pack a specific number of grouped bits aligned to the least significant bit and zero extended into the result. In this case care should be taken to set *Is2* appropriately.

17.4.2 Bit Manipulation Operations

Mnemonic		Description
cv.extract	**rD, rs1, Is3, Is2**	rD = Sext(rs1[min(Is3+Is2,31):Is2])
cv.extractu	**rD, rs1, Is3, Is2**	rD = Zext(rs1[min(Is3+Is2,31):Is2])
cv.extractr	**rD, rs1, rs2**	rD = Sext(rs1[min(rs2[9:5]+rs2[4:0],31):rs2[4:0]])
cv.extractur	**rD, rs1, rs2**	rD = Zext(rs1[min(rs2[9:5]+rs2[4:0],31):rs2[4:0]])
cv.insert	**rD, rs1, Is3, Is2**	rD[min(Is3+Is2,31):Is2] = rs1[Is3:max(Is3+Is2,31)-31] the rest of the bits of rD are passed through and are not modified
cv.insertr	**rD, rs1, rs2**	rD[min(rs2[9:5]+rs2[4:0],31):rs2[4:0]] = rs1[rs2[9:5]:max(rs2[9:5]+rs2[4:0],31)-31] the rest of the bits of rD are passed through and are not modified
cv.bclr	**rD, rs1, Is3, Is2**	rD = (rs1 & ~(((1<<Is3)-1)<<Is2))
cv.bclrr	**rD, rs1, rs2**	rD = (rs1 & ~(((1<<rs2[9:5])-1)<<rs2[4:0]))
cv.bset	**rD, rs1, Is3, Is2**	rD = (rs1 | (((1<<Is3)-1)<<Is2))
cv.bsetr	**rD, rs1, rs2**	rD = (rs1 | (((1<<rs2[9:5])-1)<<rs2[4:0]))
cv.ff1	**rD, rs1**	rD = bit position of the first bit set in rs1, starting from LSB. If bit 0 is set, rD will be 0. If only bit 31 is set, rD will be 31. If rs1 is 0, rD will be 32.
cv.fl1	**rD, rs1**	rD = bit position of the last bit set in rs1, starting from MSB. If bit 31 is set, rD will be 31. If only bit 0 is set, rD will be 0. If rs1 is 0, rD will be 32.
cv.clb	**rD, rs1**	rD = count leading bits of rs1 Note: This is the number of consecutive 1's or 0's from MSB. Note: If rs1 is 0, rD will be 0.
cv.cnt	**rD, rs1**	rD = Population count of rs1, i.e. number of bits set in rs1
cv.ror	**rD, rs1, rs2**	rD = RotateRight(rs1, rs2)
cv.bitrev	**rD, rs1, Is3, Is2**	Given an input rs1 it returns a bit reversed representation assuming FFT on 2^Is2 points in Radix 2^(Is3+1) Note: Is3 can be either 0 (radix-2), 1 (radix-4) or 2 (radix-8)

Note: Sign extension is done over the extracted bit, i.e. the Is2-th bit.

17.4.3 Bit Manipulation Encoding

31:30	29 : 25	24 : 20	19 :15	14 : 12	11 :07	06 : 00	
f2	ls3[4:0]	ls2[4:0]	rs1	funct3	rd	opcode	Mnemonic
11	Luimm5[4:0]	Iuimm5[4:0]	src	000	dest	011 0011	**cv.extract rD, rs1, Is3, Is2**
11	Luimm5[4:0]	Iuimm5[4:0]	src	001	dest	011 0011	**cv.extractu rD, rs1, Is3, Is2**
11	Luimm5[4:0]	Iuimm5[4:0]	src	010	dest	011 0011	**cv.insert rD, rs1, Is3, Is2**
11	Luimm5[4:0]	Iuimm5[4:0]	src	011	dest	011 0011	**cv.bclr rD, rs1, Is3, Is2**
11	Luimm5[4:0]	Iuimm5[4:0]	src	100	dest	011 0011	**cv.bset rD, rs1, Is3, Is2**
10	5'b0_0000	src2	src1	000	dest	011 0011	**cv.extractr rD, rs1, rs2**
10	5'b0_0000	src2	src1	001	dest	011 0011	**cv.extractur rD, rs1, rs2**
10	5'b0_0000	src2	src1	010	dest	011 0011	**cv.insertr rD, rs1, rs2**
10	5'b0_0000	src2	src1	011	dest	011 0011	**cv.bclrr rD, rs1, rs2**
10	5'b0_0000	src2	scr1	100	dest	011 0011	**cv.bsetr rD, rs1, rs2**
11	{3'bXXX,Luimm2[1:0]}	Iuimm5[4:0]	src	101	dest	011 0011	**cv.bitrev rD, rs1, Is3, Is2**

31 : 25	24 : 20	19 :15	14 : 12	11 : 7	6 : 0	
funct7	rs2	rs1	funct3	rD	opcode	
000 0100	src2	src1	101	dest	011 0011	**cv.ror rD, rs1, rs2**
000 1000	00000	src1	000	dest	011 0011	**cv.ff1 rD, rs1**
000 1000	00000	src1	001	dest	011 0011	**cv.fl1 rD, rs1**
000 1000	00000	src1	010	dest	011 0011	**cv.clb rD, rs1**
000 1000	00000	src1	011	dest	011 0011	**cv.cnt rD, rs1**

17.4.4 General ALU Operations

Mnemonic		Description
cv.abs	**rD, rs1**	rD = rs1 < 0 ? –rs1 : rs1
cv.slet	**rD, rs1, rs2**	rD = rs1 <= rs2 ? 1 : 0 Note: Comparison is signed
cv.sletu	**rD, rs1, rs2**	rD = rs1 <= rs2 ? 1 : 0 Note: Comparison is unsigned
cv.min	**rD, rs1, rs2**	rD = rs1 < rs2 ? rs1 : rs2 Note: Comparison is signed
cv.minu	**rD, rs1, rs2**	rD = rs1 < rs2 ? rs1 : rs2 Note: Comparison is unsigned

continues on next page

Table 17.1 – continued from previous page

Mnemonic		Description
cv.max	**rD, rs1, rs2**	rD = rs1 < rs2 ? rs2 : rs1 Note: Comparison is signed
cv.maxu	**rD, rs1, rs2**	rD = rs1 < rs2 ? rs2 : rs1 Note: Comparison is unsigned
cv.exths	**rD, rs1**	rD = Sext(rs1[15:0])
cv.exthz	**rD, rs1**	rD = Zext(rs1[15:0])
cv.extbs	**rD, rs1**	rD = Sext(rs1[7:0])
cv.extbz	**rD, rs1**	rD = Zext(rs1[7:0])
cv.clip	**rD, rs1, Is2**	if rs1 <= -2^(Is2-1), rD = -2^(Is2-1), else if rs1 >= 2^(Is2-1)–1, rD = 2^(Is2-1)-1, else rD = rs1 Note: If ls2 is equal to 0, -2^(Is2-1)= -1 while (2^(Is2-1)-1)=0;
cv.clipr	**rD, rs1, rs2**	if rs1 <= -(rs2+1), rD = -(rs2+1), else if rs1 >=rs2, rD = rs2, else rD = rs1
cv.clipu	**rD, rs1, Is2**	if rs1 <= 0, rD = 0, else if rs1 >= 2^(Is2–1)-1, rD = 2^(Is2-1)-1, else rD = rs1 Note: If ls2 is equal to 0, (2^(Is2-1)-1)=0;
cv.clipur	**rD, rs1, rs2**	if rs1 <= 0, rD = 0, else if rs1 >= rs2, rD = rs2, else rD = rs1
cv.addN	**rD, rs1, rs2, Is3**	rD = (rs1 + rs2) >>> Is3 Note: Arithmetic shift right. Setting Is3 to 2 replaces former p.avg
cv.adduN	**rD, rs1, rs2, Is3**	rD = (rs1 + rs2) >> Is3 Note: Logical shift right. Setting Is3 to 2 replaces former p.avg
cv.addRN	**rD, rs1, rs2, Is3**	rD = (rs1 + rs2 + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right.
cv.adduRN	**rD, rs1, rs2, Is3**	rD = (rs1 + rs2 + 2^(Is3-1))) >> Is3 Note: Logical shift right.
cv.addNr	**rD, rs1, rs2**	rD = (rD + rs1) >>> rs2[4:0] Note: Arithmetic shift right.
cv.adduNr	**rD, rs1, rs2**	rD = (rD + rs1) >> rs2[4:0] Note: Logical shift right.
cv.addRNr	**rD, rs1, rs2**	rD = (rD + rs1 + 2^(rs2[4:0]-1)) >>> rs2[4:0] Note: Arithmetic shift right.
cv.adduRNr	**rD, rs1, rs2**	rD = (rD + rs1 + 2^(rs2[4:0]-1))) >> rs2[4:0] Note: Logical shift right.
cv.subN	**rD, rs1, rs2, Is3**	rD = (rs1 - rs2) >>> Is3 Note: Arithmetic shift right.
cv.subuN	**rD, rs1, rs2, Is3**	rD = (rs1 - rs2) >> Is3 Note: Logical shift right.

continues on next page

Table 17.1 – continued from previous page

Mnemonic		Description
cv.subRN	**rD, rs1, rs2, Is3**	rD = (rs1 - rs2 + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right.
cv.subuRN	**rD, rs1, rs2, Is3**	rD = (rs1 - rs2 + 2^(Is3-1))) >> Is3 Note: Logical shift right.
cv.subNr	**rD, rs1, rs2**	rD = (rD – rs1) >>> rs2[4:0] Note: Arithmetic shift right.
cv.subuNr	**rD, rs1, rs2**	rD = (rD – rs1) >> rs2[4:0] Note: Logical shift right.
cv.subRNr	**rD, rs1, rs2**	rD = (rD – rs1+ 2^(rs2[4:0]-1)) >>> rs2[4:0] Note: Arithmetic shift right.
cv.subuRNr	**rD, rs1, rs2**	rD = (rD – rs1+ 2^(rs2[4:0]-1))) >> rs2[4:0] Note: Logical shift right.

17.4.5 General ALU Encoding

31 : 25	24 : 20	19 :15	14 : 12	11 : 7	6 : 0	
funct7	rs2	rs1	funct	rD	opcode	
000 0010	00000	src1	000	dest	011 0011	**cv.abs rD, rs1**
000 0010	src2	src1	010	dest	011 0011	**cv.slet rD, rs1, rs2**
000 0010	src2	src1	011	dest	011 0011	**cv.sletu rD, rs1, rs2**
000 0010	src2	src1	100	dest	011 0011	**cv.min rD, rs1, rs2**
000 0010	src2	src1	101	dest	011 0011	**cv.minu rD, rs1, rs2**
000 0010	src2	src1	110	dest	011 0011	**cv.max rD, rs1, rs2**
000 0010	src2	src1	111	dest	011 0011	**cv.maxu rD, rs1, rs2**
000 1000	00000	src1	100	dest	011 0011	**cv.exths rD, rs1**
000 1000	00000	src1	101	dest	011 0011	**cv.exthz rD, rs1**
000 1000	00000	src1	110	dest	011 0011	**cv.extbs rD, rs1**
000 1000	00000	src1	111	dest	011 0011	**cv.extbz rD, rs1**

31 : 25	24 : 20	19 :15	14 : 12	11 : 7	6 : 0	
funct7	Is2[4:0]	rs1	funct3	rD	opcode	
000 1010	Iuimm5[4:0]	src1	001	dest	011 0011	**cv.clip rD, rs1, Is2**
000 1010	Iuimm5[4:0]	src1	010	dest	011 0011	**cv.clipu rD, rs1, Is2**
000 1010	src2	src1	101	dest	011 0011	**cv.clipr rD, rs1, rs2**
000 1010	src2	src1	110	dest	011 0011	**cv.clipur rD, rs1, rs2**

31:30	29 : 25	24 :20	19 :15	14 : 12	11 : 7	6 : 0	
f2	Is3[4:0]	rs2	rs1	funct3	rD	opcode	
00	Luimm5[4:0]	src2	src1	010	dest	101 1011	**cv.addN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	010	dest	101 1011	**cv.adduN rD, rs1, rs2, Is3**
00	Luimm5[4:0]	src2	src1	110	dest	101 1011	**cv.addRN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	110	dest	101 1011	**cv.adduRN rD, rs1, rs2, Is3**
00	Luimm5[4:0]	src2	src1	011	dest	101 1011	**cv.subN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	011	dest	101 1011	**cv.subuN rD, rs1, rs2, Is3**
00	Luimm5[4:0]	src2	src1	111	dest	101 1011	**cv.subRN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	111	dest	101 1011	**cv.subuRN rD, rs1, rs2, Is3**
01	00000	src2	src1	010	dest	101 1011	**cv.addNr rD, rs1, rs2**
11	00000	src2	src1	010	dest	101 1011	**cv.adduNr rD, rs1, rs**
01	00000	src2	src1	110	dest	101 1011	**cv.addRNr rD, rs1, rs**
11	00000	src2	src1	110	dest	101 1011	**cv.adduRNr rD, rs1, rs2**
01	00000	src2	src1	011	dest	101 1011	**cv.subNr rD, rs1, rs2**
11	00000	src2	src1	011	dest	101 1011	**cv.subuNr rD, rs1, rs2**
01	00000	src2	src1	111	dest	101 1011	**cv.subRNr rD, rs1, rs2**
11	00000	src2	src1	111	dest	101 1011	**cv.subuRNr rD, rs1, rs2**

17.4.6 Immediate Branching Operations

Mnemonic	Description
cv.beqimm rs1, Imm5, Imm12	Branch to PC + (Imm12 << 1) if rs1 is equal to Imm5. Imm5 is signed.
cv.bneimm rs1, Imm5, Imm12	Branch to PC + (Imm12 << 1) if rs1 is not equal to Imm5. Imm5 is signed.

17.4.7 Immediate Branching Encoding

31	30 : 25	24 : 20	19 : 15	14 : 12	11 : 8	7	6 : 0	
Imm12[12]	Imm12[10:5]	rs2	rs1	funct3	Imm12	Imm12	op-code	
Imm12[12]	Imm12[10:5]	Imm5	src1	010	Imm12[4:1]	Imm12[11]	110 0011	**cv.beqimm rs1, Imm5, Imm12**
Imm12[12]	Imm12[10:5]	Imm5	src1	011	Imm12[4:1]	Imm12[11]	110 0011	**cv.bneimm rs1, Imm5, Imm12**

17.5 Multiply-Accumulate

CV32E41P supports custom extensions for multiply-accumulate and half-word multiplications with an optional post-multiplication shift.

The custom multiply-accumulate extensions are only supported if `PULP_XPULP` == 1.

17.5.1 MAC Operations

32-Bit x 32-Bit Multiplication Operations

Mnemonic	Description	
cv.mac	**rD, rs1, rs2**	rD = rD + rs1 * rs2
cv.msu	**rD, rs1, rs2**	rD = rD - rs1 * rs2

16-Bit x 16-Bit Multiplication

Mnemonic	Description	
cv.muls	**rD, rs1, rs2**	rD[31:0] = Sext(rs1[15:0]) * Sext(rs2[15:0])
cv.mulhhs	**rD, rs1, rs2**	rD[31:0] = Sext(rs1[31:16]) * Sext(rs2[31:16])
cv.mulsN	**rD, rs1, rs2, Is3**	rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0])) >>> Is3 Note: Arithmetic shift right
cv.mulhhsN	**rD, rs1, rs2, Is3**	rD[31:0] = (Sext(rs1[31:16]) * Sext(rs2[31:16])) >>> Is3 Note: Arithmetic shift right
cv.mulsRN	**rD, rs1, rs2, Is3**	rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0]) + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right
cv.mulhhsRN	**rD, rs1, rs2, Is3**	rD[31:0] = (Sext(rs1[31:16]) * Sext(rs2[31:16]) + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right
cv.mulu	**rD, rs1, rs2**	rD[31:0] = Zext(rs1[15:0]) * Zext(rs2[15:0])
cv.mulhhu	**rD, rs1, rs2**	rD[31:0] = Zext(rs1[31:16]) * Zext(rs2[31:16])
cv.muluN	**rD, rs1, rs2, Is3**	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0])) >> Is3 Note: Logical shift right
cv.mulhhuN	**rD, rs1, rs2, Is3**	rD[31:0] = (Zext(rs1[31:16]) * Zext(rs2[31:16])) >> Is3 Note: Logical shift right
cv.muluRN	**rD, rs1, rs2, Is3**	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0]) + 2^(Is3-1)) >> Is3 Note: Logical shift right
cv.mulhhuRN	**rD, rs1, rs2, Is3**	rD[31:0] = (Zext(rs1[31:16]) * Zext(rs2[31:16]) + 2^(Is3-1)) >> Is3 Note: Logical shift right

16-Bit x 16-Bit Multiply-Accumulate

Mnemonic	Description	
cv.macsN	**rD, rs1, rs2, Is3**	rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0]) + rD) >>> Is3 Note: Arithmetic shift right
cv.machhsN	**rD, rs1, rs2, Is3**	rD[31:0] = (Sext(rs1[31:16]) * Sext(rs2[31:16]) + rD) >>> Is3 Note: Arithmetic shift right
cv.macsRN	**rD, rs1, rs2, Is3**	rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0]) + rD + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right
cv.machhsRN	**, rD, rs1, rs2, Is3**	rD[31:0] = (Sext(rs1[31:16]) * Sext(rs2[31:16]) + rD + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right
cv.macuN	**rD, rs1, rs2, Is3**	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0]) + rD) >> Is3 Note: Logical shift right
cv.machhuN	**rD, rs1, rs2, Is3**	rD[31:0] = (Zext(rs1[31:16]) * Zext(rs2[31:16]) + rD) >> Is3 Note: Logical shift right
cv.macuRN	**rD, rs1, rs2, Is3**	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0]) + rD + 2^(Is3-1)) >> Is3 Note: Logical shift right
cv.machhuRN	**rD, rs1, rs2, Is3**	rD[31:0] = (Zext(rs1[31:16]) * Zext(rs2[31:16]) + rD + 2^(Is3-1)) >> Is3 Note: Logical shift right

17.5.2 MAC Encoding

31 : 25	24 :20	19 :15	14 : 12	11 : 7	6 : 0	
funct7	rs2	rs1	funct3	rD	opcode	
010 0001	src2	src1	000	dest	011 0011	**cv.mac rD, rs1, rs2**
010 0001	src2	src1	001	dest	011 0011	**cv.msu rD, rs1, rs2**

31:30	29 : 25	24 :20	19 :15	14 : 12	11 : 7	6 : 0	
f2	Is3[4:0]	rs2	rs1	funct3	rD	opcode	
10	00000	src2	src1	000	dest	101 1011	**cv.muls rD, rs1, rs2**
11	00000	src2	src1	000	dest	101 1011	**cv.mulhhs rD, rs1, rs2**
10	Luimm5[4:0]	src2	src1	000	dest	101 1011	**cv.mulsN rD, rs1, rs2, Is3**
11	Luimm5[4:0]	src2	src1	000	dest	101 1011	**cv.mulhhsN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	100	dest	101 1011	**cv.mulsRN rD, rs1, rs2, Is3**
11	Luimm5[4:0]	src2	src1	100	dest	101 1011	**cv.mulhhsRN rD, rs1, rs2, Is3**
00	00000	src2	src1	000	dest	101 1011	**cv.mulu rD, rs1, rs2**
01	00000	src2	src1	000	dest	101 1011	**cv.mulhhu rD, rs1, rs2**
00	Luimm5[4:0]	src2	src1	000	dest	101 1011	**cv.muluN rD, rs1, rs2, Is3**
01	Luimm5[4:0]	src2	src1	000	dest	101 1011	**cv.mulhhuN rD, rs1, rs2, Is3**
00	Luimm5[4:0]	src2	src1	100	dest	101 1011	**cv.muluRN rD, rs1, rs2, Is3**
01	Luimm5[4:0]	src2	src1	100	dest	101 1011	**cv.mulhhuRN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	001	dest	101 1011	**cv.macsN rD, rs1, rs2, Is3**
11	Luimm5[4:0]	src2	src1	001	dest	101 1011	**cv.machhsN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	101	dest	101 1011	**cv.macsRN rD, rs1, rs2, Is3**
11	Luimm5[4:0]	src2	src1	101	dest	101 1011	**cv.machhsRN rD, rs1, rs2, Is3**
00	Luimm5[4:0]	src2	src1	001	dest	101 1011	**cv.macuN rD, rs1, rs2, Is3**
01	Luimm5[4:0]	src2	src1	001	dest	101 1011	**cv.machhuN rD, rs1, rs2, Is3**
00	Luimm5[4:0]	src2	src1	101	dest	101 1011	**cv.macuRN rD, rs1, rs2, Is3**
01	Luimm5[4:0]	src2	src1	101	dest	101 1011	**cv.machhuRN rD, rs1, rs2, Is3**

17.6 SIMD

The SIMD instructions perform operations on multiple sub-word elements at the same time. This is done by segmenting the data path into smaller parts when 8 or 16-bit operations should be performed.

The custom SIMD extensions are only supported if `PULP_XPULP == 1`.

SIMD is not supported by the compiler tool chain.

SIMD instructions are available in two flavors:

- 8-Bit, to perform four operations on the 4 bytes inside a 32-bit word at the same time (.b)
- 16-Bit, to perform two operations on the 2 half-words inside a 32-bit word at the same time (.h)

All the operations are rounded to the specified bidwidth as for the original RISC-V arithmetic operations. This is described by the "and" operation with a MASK. No overflow or carry-out flags are generated as for the 32-bit operations.

Additionally, there are three modes that influence the second operand:

1. Normal mode, vector-vector operation. Both operands, from rs1 and rs2, are treated as vectors of bytes or half-words.

   e.g. cv.add.h x3,x2,x1 performs:

   x3[31:16] = x2[31:16] + x1[31:16]

   x3[15: 0] = x2[15: 0] + x1[15: 0]

2. Scalar replication mode (.sc), vector-scalar operation. Operand 1 is treated as a vector, while operand 2 is treated as a scalar and replicated two or four times to form a complete vector. The LSP is used for this purpose.

   e.g. cv.add.sc.h x3,x2,x1 performs:

   x3[31:16] = x2[31:16] + x1[15: 0]

   x3[15: 0] = x2[15: 0] + x1[15: 0]

3. Immediate scalar replication mode (.sci), vector-scalar operation. Operand 1 is treated as vector, while operand 2 is treated as a scalar and comes from an immediate. The immediate is either sign- or zero-extended, depending on the operation. If not specified, the immediate is sign-extended.

   e.g. cv.add.sci.h x3,x2,0xDA performs:

   x3[31:16] = x2[31:16] + 0xFFDA

   x3[15: 0] = x2[15: 0] + 0xFFDA

In the following table, the index i ranges from 0 to 1 for 16-Bit operations and from 0 to 3 for 8-Bit operations.

- The index 0 is 15:0 for 16-Bit operations, or 7:0 for 8-Bit operations.
- The index 1 is 31:16 for 16-Bit operations, or 15:8 for 8-Bit operations.
- The index 2 is 23:16 for 8-Bit operations.
- The index 3 is 31:24 for 8-Bit operations.

17.6.1 SIMD ALU Operations

Mnemonic	Description
cv.add[.sc,.sci]{.h,.b}	rD[i] = (rs1[i] + op2[i]) & 0xFFFF
cv.add{.div2,.div4, .div8}	rD[i] = ((rs1[i] + op2[i]) & 0xFFFF)>>{1,2,3}
cv.sub[.sc,.sci]{.h,.b}	rD[i] = (rs1[i] - op2[i]) & 0xFFFF
cv.sub{.div2,.div4, .div8}	rD[i] = ((rs1[i] – op2[i]) & 0xFFFF)>>{1,2,3}
cv.avg[.sc,.sci]{.h,.b}	rD[i] = ((rs1[i] + op2[i]) & {0xFFFF, 0xFF}) >> 1 Note: Arithmetic right shift
cv.avgu[.sc,.sci]{.h,.b}	rD[i] = ((rs1[i] + op2[i]) & {0xFFFF, 0xFF}) >> 1
cv.min[.sc,.sci]{.h,.b}	rD[i] = rs1[i] < op2[i] ? rs1[i] : op2[i]
cv.minu[.sc,.sci]{.h,.b}	rD[i] = rs1[i] < op2[i] ? rs1[i] : op2[i] Note: Immediate is zero-extended, comparison is unsigned
cv.max[.sc,.sci]{.h,.b}	rD[i] = rs1[i] > op2[i] ? rs1[i] : op2[i]
cv.maxu[.sc,.sci]{.h,.b}	rD[i] = rs1[i] > op2[i] ? rs1[i] : op2[i] Note: Immediate is zero-extended, comparison is unsigned
cv.srl[.sc,.sci]{.h,.b}	rD[i] = rs1[i] >> op2[i] Note: Immediate is zero-extended, shift is logical
cv.sra[.sc,.sci]{.h,.b}	rD[i] = rs1[i] >>> op2[i] Note: Immediate is zero-extended, shift is arithmetic
cv.sll[.sc,.sci]{.h,.b}	rD[i] = rs1[i] << op2[i] Note: Immediate is zero-extended, shift is logical
cv.or[.sc,.sci]{.h,.b}	rD[i] = rs1[i] | op2[i]
cv.xor[.sc,.sci]{.h,.b}	rD[i] = rs1[i] ^ op2[i]
cv.and[.sc,.sci]{.h,.b}	rD[i] = rs1[i] & op2[i]
cv.abs{.h,.b}	rD[i] = rs1 < 0 ? –rs1 : rs1
cv.extract.h	rD = Sext(rs1[((I+1)*16)-1 : I*16])
cv.extract.b	rD = Sext(rs1[((I+1)*8)-1 : I*8])
cv.extractu.h	rD = Zext(rs1[((I+1)*16)-1 : I*16])
cv.extractu.b	rD = Zext(rs1[((I+1)*8)-1 : I*8])
cv.insert.h	rD[((I+1)*16-1:I*16] = rs1[15:0] Note: The rest of the bits of rD are untouched and keep their previous value
cv.insert,b	rD[((I+1)*8-1:I*8] = rs1[7:0] Note: The rest of the bits of rD are untouched and keep their previous value

Dot Product Instructions

Mnemonic	Description
cv.dotup[.sc,.sci].h	rD = rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operations are unsigned
cv.dotup[.sc,.sci].b	rD = rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operations are unsigned
cv.dotusp[.sc,.sci].h	rD = rs1[0] * op2[0] + rs1[1] * op2[1] Note: rs1 is treated as unsigned, while rs2 is treated as signed
cv.dotusp[.sc,.sci].b	rD = rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: rs1 is treated as unsigned, while rs2 is treated as signed
cv.dotsp[.sc,.sci].h	rD = rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operations are signed
cv.dotsp[.sc,.sci].b	rD = rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operations are signed
cv.sdotup[.sc,.sci].h	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operations are unsigned
cv.sdotup[.sc,.sci].b	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operations are unsigned
cv.sdotusp[.sc,.sci].h	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] Note: rs1 is treated as unsigned, while rs2 is treated as signed
cv.sdotusp[.sc,.sci].b	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: rs1 is treated as unsigned, while rs2 is treated as signed
cv.sdotsp[.sc,.sci].h	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operations are signed
cv.sdotsp[.sc,.sci].b	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operations are signed

Shuffle and Pack Instructions

Mnemonic	Description
cv.shuffle.h	rD[31:16] = rs1[rs2[16]*16+15:rs2[16]*16] rD[15:0] = rs1[rs2[0]*16+15:rs2[0]*16]
cv.shuffle.sci.h	rD[31:16] = rs1[I1*16+15:I1*16] rD[15:0] = rs1[I0*16+15:I0*16] Note: I1 and I0 represent bits 1 and 0 of the immediate
cv.shuffle.b	rD[31:24] = rs1[rs2[25:24]*8+7:rs2[25:24]*8] rD[23:16] = rs1[rs2[17:16]*8+7:rs2[17:16]*8] rD[15:8] = rs1[rs2[9:8]*8+7:rs2[9:8]*8] rD[7:0] = rs1[rs2[1:0]*8+7:rs2[1:0]*8]
cv.shuffle0.sci.b	rD[31:24] = rs1[7:0] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
cv.shuffle1.sci.b	rD[31:24] = rs1[15:8] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
cv.shuffle2.sci.b	rD[31:24] = rs1[23:16] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
cv.shuffle3.sci.b	rD[31:24] = rs1[31:24] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
cv.shuffle2.h	rD[31:16] = ((rs2[17] == 1) ? rs1 : rD)[rs2[16]*16+15:rs2[16]*16] rD[15:0] = ((rs2[1] == 1) ? rs1 : rD)[rs2[0]*16+15:rs2[0]*16]
cv.shuffle2.b	rD[31:24] = ((rs2[26] == 1) ? rs1 : rD)[rs2[25:24]*8+7:rs2[25:24]*8] rD[23:16] = ((rs2[18] == 1) ? rs1 : rD)[rs2[17:16]*8+7:rs2[17:16]*8] rD[15:8] = ((rs2[10] == 1) ? rs1 : rD)[rs2[9:8]*8+7:rs2[9:8]*8] rD[7:0] = ((rs2[2] == 1) ? rs1 : rD)[rs2[1:0]*8+7:rs2[1:0]*8]
cv.pack	rD[31:16] = rs1[15:0] rD[15:0] = rs2[15:0]
cv.pack.h	rD[31:16] = rs1[31:16] rD[15:0] = rs2[31:16]
cv.packhi.b	rD[31:24] = rs1[7:0] rD[23:16] = rs2[7:0] Note: The rest of the bits of rD are untouched and keep their previous value
cv.packlo.b	rD[15:8] = rs1[7:0] rD[7:0] = rs2[7:0] Note: The rest of the bits of rD are untouched and keep their previous value

17.6.2 SIMD ALU Encoding

31 : 27	26	25	24 : 20	19 : 15	14 :12	11 : 7	6 : 0	
funct5	F		rs2	rs1	funct3	rD	opcode	
0 0000	0	0	src2	src1	000	dest	101 0111	**cv.a**
0 0000	0	0	src2	src1	100	dest	101 0111	**cv.a**
0 0000	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.a**
0 0000	0	0	src2	src1	001	dest	101 0111	**cv.a**
0 0000	0	0	src2	src1	101	dest	101 0111	**cv.a**
0 0000	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.a**
0 1011	1	X	src2	src1	010	dest	101 0111	**cv.a**
0 1011	1	X	src2	src1	100	dest	101 0111	**cv.a**
0 1011	1	x	src2	src1	110	dest	101 0111	**cv.a**
0 0001	0	0	src2	src1	000	dest	101 0111	**cv.s**
0 0001	0	0	src2	src1	100	dest	101 0111	**cv.s**
0 0001	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.s**
0 0001	0	0	src2	src1	001	dest	101 0111	**cv.s**
0 0001	0	0	src2	src1	101	dest	101 0111	**cv.s**
0 0001	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.s**
0 1100	1	x	src2	src1	010	dest	101 0111	**cv.s**
0 1100	1	x	src2	src1	100	dest	101 0111	**cv.s**
0 1100	1	x	src2	src1	110	dest	101 0111	**cv.s**
0 0010	0	0	src2	src1	000	dest	101 0111	**cv.a**
0 0010	0	0	src2	src1	100	dest	101 0111	**cv.a**
0 0010	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.a**
0 0010	0	0	src2	src1	001	dest	101 0111	**cv.a**
0 0010	0	0	src2	src1	101	dest	101 0111	**cv.a**
0 0010	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.a**
0 0011	0	0	src2	src1	000	dest	101 0111	**cv.a**
0 0011	0	0	src2	src1	100	dest	101 0111	**cv.a**
0 0011	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.a**
0 0011	0	0	src2	src1	001	dest	101 0111	**cv.a**
0 0011	0	0	src2	src1	101	dest	101 0111	**cv.a**
0 0011	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.a**
0 0100	0	0	src2	src1	000	dest	101 0111	**cv.m**
0 0100	0	0	src2	src1	100	dest	101 0111	**cv.m**
0 0100	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.m**
0 0100	0	0	src2	src1	001	dest	101 0111	**cv.m**
0 0100	0	0	src2	src1	101	dest	101 0111	**cv.m**
0 0100	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.m**
0 0101	0	0	src2	src1	000	dest	101 0111	**cv.m**
0 0101	0	0	src2	src1	100	dest	101 0111	**cv.m**
0 0101	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.m**
0 0101	0	0	src2	src1	001	dest	101 0111	**cv.m**
0 0101	0	0	src2	src1	101	dest	101 0111	**cv.m**
0 0101	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.m**
0 0110	0	0	src2	src1	000	dest	101 0111	**cv.m**
0 0110	0	0	src2	src1	100	dest	101 0111	**cv.m**
0 0110	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.m**
0 0110	0	0	src2	src1	001	dest	101 0111	**cv.m**

Table 17.2 – continued from previous page

31 : 27	26	25	24 : 20	19 : 15	14 :12	11 : 7	6 : 0	
funct5	F		rs2	rs1	funct3	rD	opcode	
0 0110	0	0	src2	src1	101	dest	101 0111	**cv.m**
0 0110	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.m**
0 0111	0	0	src2	src1	000	dest	101 0111	**cv.m**
0 0111	0	0	src2	src1	100	dest	101 0111	**cv.m**
0 0111	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.m**
0 0111	0	0	src2	src1	001	dest	101 0111	**cv.m**
0 0111	0	0	src2	src1	101	dest	101 0111	**cv.m**
0 0111	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.m**
0 1000	0	0	src2	src1	000	dest	101 0111	**cv.s**
0 1000	0	0	src2	src1	100	dest	101 0111	**cv.s**
0 1000	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.s**
0 1000	0	0	src2	src1	001	dest	101 0111	**cv.s**
0 1000	0	0	src2	src1	101	dest	101 0111	**cv.s**
0 1000	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.s**
0 1001	0	0	src2	src1	000	dest	101 0111	**cv.s**
0 1001	0	0	src2	src1	100	dest	101 0111	**cv.s**
0 1001	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.s**
0 1001	0	0	src2	src1	001	dest	101 0111	**cv.s**
0 1001	0	0	src2	src1	101	dest	101 0111	**cv.s**
0 1001	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.s**
0 1010	0	0	src2	src1	000	dest	101 0111	**cv.sl**
0 1010	0	0	src2	src1	100	dest	101 0111	**cv.sl**
0 1010	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.sl**
0 1010	0	0	src2	src1	001	dest	101 0111	**cv.sl**
0 1010	0	0	src2	src1	101	dest	101 0111	**cv.sl**
0 1010	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.sl**
0 1011	0	0	src2	src1	000	dest	101 0111	**cv.o**
0 1011	0	0	src2	src1	100	dest	101 0111	**cv.o**
0 1011	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.o**
0 1011	0	0	src2	src1	001	dest	101 0111	**cv.o**
0 1011	0	0	src2	src1	101	dest	101 0111	**cv.o**
0 1011	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.o**
0 1100	0	0	src2	src1	000	dest	101 0111	**cv.x**
0 1100	0	0	src2	src1	100	dest	101 0111	**cv.x**
0 1100	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.x**
0 1100	0	0	src2	src1	001	dest	101 0111	**cv.x**
0 1100	0	0	src2	src1	101	dest	101 0111	**cv.x**
0 1100	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.x**
0 1101	0	0	src2	src1	000	dest	101 0111	**cv.a**
0 1101	0	0	src2	src1	100	dest	101 0111	**cv.a**
0 1101	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.a**
0 1101	0	0	src2	src1	001	dest	101 0111	**cv.a**
0 1101	0	0	src2	src1	101	dest	101 0111	**cv.a**
0 1101	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.a**
0 1110	0	0	0	src1	000	dest	101 0111	**cv.a**
0 1110	0	0	0	src1	001	dest	101 0111	**cv.a**
0 1011	1	x	0	src1	000	dest	101 0111	**cv.c**
0 1111	0	Imm6[5:0]		src1	110	dest	101 0111	**cv.e**

Table 17.2 – continued from previous page

31 : 27	26	25	24 : 20	19 : 15	14 :12	11 : 7	6 : 0	
funct5	F		rs2	rs1	funct3	rD	opcode	
0 1111	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.ex**
1 0010	0	Imm6[5:0]		src1	110	dest	101 0111	**cv.ex**
1 0010	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.ex**
1 0110	0	Imm6[5:0]		src1	110	dest	101 0111	**cv.in**
1 0110	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.in**
1 0000	0	0	src2	src1	000	dest	101 0111	**cv.d**
1 0000	0	0	src2	src1	100	dest	101 0111	**cv.d**
1 0000	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.d**
1 0000	0	0	src2	src1	001	dest	101 0111	**cv.d**
1 0000	0	0	src2	src1	101	dest	101 0111	**cv.d**
1 0000	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.d**
1 0001	0	0	src2	src1	000	dest	101 0111	**cv.d**
1 0001	0	0	src2	src1	100	dest	101 0111	**cv.d**
1 0001	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.d**
1 0001	0	0	src2	src1	001	dest	101 0111	**cv.d**
1 0001	0	0	src2	src1	101	dest	101 0111	**cv.d**
1 0001	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.d**
1 0011	0	0	src2	src1	000	dest	101 0111	**cv.d**
1 0011	0	0	src2	src1	100	dest	101 0111	**cv.d**
1 0011	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.d**
1 0011	0	0	src2	src1	001	dest	101 0111	**cv.d**
1 0011	0	0	src2	src1	101	dest	101 0111	**cv.d**
1 0011	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.d**
1 0100	0	0	src2	src1	000	dest	101 0111	**cv.s**
1 0100	0	0	src2	src1	100	dest	101 0111	**cv.s**
1 0100	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.s**
1 0100	0	0	src2	src1	001	dest	101 0111	**cv.s**
1 0100	0	0	src2	src1	101	dest	101 0111	**cv.s**
1 0100	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.s**
1 0101	0	0	src2	src1	000	dest	101 0111	**cv.s**
1 0101	0	0	src2	src1	100	dest	101 0111	**cv.s**
1 0101	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.s**
1 0101	0	0	src2	src1	001	dest	101 0111	**cv.s**
1 0101	0	0	src2	src1	101	dest	101 0111	**cv.s**
1 0101	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.s**
1 0111	0	0	src2	src1	000	dest	101 0111	**cv.s**
1 0111	0	0	src2	src1	100	dest	101 0111	**cv.s**
1 0111	0	Imm6[5:0]s		src1	110	dest	101 0111	**cv.s**
1 0111	0	0	src2	src1	001	dest	101 0111	**cv.s**
1 0111	0	0	src2	src1	101	dest	101 0111	**cv.s**
1 0111	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.s**
1 1000	0	0	src2	src1	000	dest	101 0111	**cv.sl**
1 1000	0	Imm6[5:0]		src1	110	dest	101 0111	**cv.sl**
1 1000	0	0	src2	src1	001	dest	101 0111	**cv.sl**
1 1000	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.sl**
1 1101	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.sl**
1 1110	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.sl**
1 1111	0	Imm6[5:0]		src1	111	dest	101 0111	**cv.sl**

Table 17.2 – continued from previous page

31 : 27	26	25	24 : 20	19 : 15	14 :12	11 : 7	6 : 0	
funct5	F		rs2	rs1	funct3	rD	opcode	
1 1001	0	0	src2	src1	000	dest	101 0111	**cv.sl**
1 1001	0	0	src2	src1	001	dest	101 0111	**cv.sl**
1 1010	0	0	src2	src1	000	dest	101 0111	**cv.p**
1 1010	0	1	src2	src1	000	dest	101 0111	**cv.p**
1 1011	0	0	src2	src1	001	dest	101 0111	**cv.p**
1 1100	0	0 | src2		src1	001	dest	101 0111	**cv.p**

Note: Imm6[5:0] is encoded as { Imm6[0], Imm6[5:1] }, LSB at the 25th bit of the instruction

17.6.3 SIMD Comparison Operations

SIMD comparisons are done on individual bytes (.b) or half-words (.h), depending on the chosen mode. If the comparison result is true, all bits in the corresponding byte/half-word are set to 1. If the comparison result is false, all bits are set to 0.

The default mode (no .sc, .sci) compares the lowest byte/half-word of the first operand with the lowest byte/half-word of the second operand, and so on. If the mode is set to scalar replication (.sc), always the lowest byte/half-word of the second operand is used for comparisons, thus instead of a vector comparison a scalar comparison is performed. In the immediate scalar replication mode (.sci), the immediate given to the instruction is used for the comparison.

Mnemonic		Description
cv.cmpeq[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] == op2 ? ‘1 : ‘0
cv.cmpne[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] != op2 ? ‘1 : ‘0
cv.cmpgt[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] > op2 ? ‘1 : ‘0
cv.cmpge[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] >=op2 ? ‘1 : ‘0
cv.cmplt[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] < op2 ? ‘1 : ‘0
cv.cmple[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] <= op2 ? ‘1 : ‘0
cv.cmpgtu[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] > op2 ? ‘1 : ‘0 Note: Unsigned comparison
cv.cmpgeu[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] >= op2 ? ‘1 : ‘0 Note: Unsigned comparison
cv.cmpltu[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] < op2 ? ‘1 : ‘0 Note: Unsigned comparison
cv.cmpleu[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] <= op2 ? ‘1 : ‘0 Note: Unsigned comparison

17.6.4 SIMD Comparison Encoding

31 : 27	26	25	24 : 20	19 : 15	14 : 12	11 : 7	6 : 0	
funct5	F		rs2	rs1	funct3	rD	opcode	
0 0000	1	0	src2	src1	000	dest	101 0111	**cv.c**
0 0000	1	0	src2	src1	100	dest	101 0111	**cv.c**
0 0000	1	Imm6[5:0]		src1	110	dest	101 0111	**cv.c**
0 0000	1	0	src2	src1	001	dest	101 0111	**cv.c**
0 0000	1	0	src2	src1	101	dest	101 0111	**cv.c**
0 0000	1	Imm6[5:0]		src1	111	dest	101 0111	**cv.c**
0 0001	1	0	src2	src1	000	dest	101 0111	**cv.c**
0 0001	1	0	src2	src1	100	dest	101 0111	**cv.c**
0 0001	1	Imm6[5:0]		src1	110	dest	101 0111	**cv.c**
0 0001	1	0	src2	src1	001	dest	101 0111	**cv.c**

Table 17.3 – continued from previous page

31 : 27	26	25	24 : 20	19 : 15	14 : 12	11 : 7	6 : 0	
funct5	F		rs2	rs1	funct3	rD	opcode	
0 0001	1	0	src2	src1	101	dest	101 0111	**cv.c**
0 0001	1	Imm6[5:0]		src1	111	dest	101 0111	**cv.c**
0 0010	1	0	src2	src1	000	dest	101 0111	**cv.c**
0 0010	1	0	src2	src1	100	dest	101 0111	**cv.c**
0 0010	1	Imm6[5:0]		src1	110	dest	101 0111	**cv.c**
0 0010	1	0	src2	src1	001	dest	101 0111	**cv.c**
0 0010	1	0	src2	src1	101	dest	101 0111	**cv.c**
0 0010	1	Imm6[5:0]		src1	111	dest	101 0111	**cv.c**
0 0011	1	0	src2	src1	000	dest	101 0111	**cv.c**
0 0011	1	0	src2	src1	100	dest	101 0111	**cv.c**
0 0011	1	Imm6[5:0]		src1	110	dest	101 0111	**cv.c**
0 0011	1	0	src2	src1	001	dest	101 0111	**cv.c**
0 0011	1	0	src2	src1	101	dest	101 0111	**cv.c**
0 0011	1	Imm6[5:0]		src1	111	dest	101 0111	**cv.c**
0 0100	1	0	src2	src1	000	dest	101 0111	**cv.c**
0 0100	1	0	src2	src1	100	dest	101 0111	**cv.c**
0 0100	1	Imm6[5:0]		src1	110	dest	101 0111	**cv.c**
0 0100	1	0	src2	src1	001	dest	101 0111	**cv.c**
0 0100	1	0	src2	src1	101	dest	101 0111	**cv.c**
0 0100	1	Imm6[5:0]		src1	111	dest	101 0111	**cv.c**
0 0101	1	0	src2	src1	000	dest	101 0111	**cv.c**
0 0101	1	0	src2	src1	100	dest	101 0111	**cv.c**
0 0101	1	Imm6[5:0]		src1	110	dest	101 0111	**cv.c**
0 0101	1	0	src2	src1	001	dest	101 0111	**cv.c**
0 0101	1	0	src2	src1	101	dest	101 0111	**cv.c**
0 0101	1	Imm6[5:0]		src1	111	dest	101 0111	**cv.c**
0 0110	1	0	src2	src1	000	dest	101 0111	**cv.c**
0 0110	1	0	src2	src1	100	dest	101 0111	**cv.c**
0 0110	1	Imm6[5:0]		src1	110	dest	101 0111	**cv.c**
0 0110	1	0	src2	src1	001	dest	101 0111	**cv.c**
0 0110	1	0	src2	src1	101	dest	101 0111	**cv.c**
0 0110	1	Imm6[5:0]		src1	111	dest	101 0111	**cv.c**
0 0111	1	0	src2	src1	000	dest	101 0111	**cv.c**
0 0111	1	0	src2	src1	100	dest	101 0111	**cv.c**
0 0111	1	Imm6[5:0]		src1	110	dest	101 0111	**cv.c**
0 0111	1	0	src2	src1	001	dest	101 0111	**cv.c**
0 0111	1	0	src2	src1	101	dest	101 0111	**cv.c**
0 0111	1	Imm6[5:0]		src1	111	dest	101 0111	**cv.c**
0 1000	1	0	src2	src1	000	dest	101 0111	**cv.c**
0 1000	1	0	src2	src1	100	dest	101 0111	**cv.c**
0 1000	1	Imm6[5:0]		src1	110	dest	101 0111	**cv.c**
0 1000	1	0	src2	src1	001	dest	101 0111	**cv.c**
0 1000	1	0	src2	src1	101	dest	101 0111	**cv.c**
0 1000	1	Imm6[5:0]		src1	111	dest	101 0111	**cv.c**
0 1001	1	0	src2	src1	000	dest	101 0111	**cv.c**
0 1001	1	0	src2	src1	100	dest	101 0111	**cv.c**
0 1001	1	Imm6[5:0]		src1	110	dest	101 0111	**cv.c**
0 1001	1	0	src2	src1	001	dest	101 0111	**cv.c**

Table 17.3 – continued from previous page

31 : 27	26	25	24 : 20	19 : 15	14 : 12	11 : 7	6 : 0	
funct5	F		rs2	rs1	funct3	rD	opcode	
0 1001	1	0	src2	src1	101	dest	101 0111	**cv.c**
0 1001	1	Imm6[5:0]		src1	111	dest	101 0111	**cv.c**

Note: Imm6[5:0] is encoded as { Imm6[0], Imm6[5:1] }, LSB at the 25th bit of the instruction

17.6.5 SIMD Complex-number Operations

SIMD Complex-number operations are extra instructions that uses the packed-SIMD extentions to represent Complex-numbers. These extentions use only the half-words mode and only operand in registers. A number C = {Re, Im} is represented as a vector of two 16-Bits signed numbers. C[0] is the real part [15:0], C[1] is the imaginary part [31:16]. Such operations are subtraction of 2 complexes with post rotation by -j, the complex and conjugate, and Complex multiplications. The complex multiplications are performed in two separate instructions, one to compute the real part, and one to compute the imaginary part.

As for all the other SIMD instructions, no flags are raised and CSR register are unmodified. No carry, overflow is generated. Instructions are rounded up as the mask & 0xFFFF explicits.

Mnemonic	Description
cv.subrotmj{/,div2,div4,div8}	rD[0] = ((rs1[1] – rs2[1]) & 0xFFFF)>>{0,1,2,3} rD[1] = ((rs2[0] – rs1[0]) & 0xFFFF)>>{0,1,2,3}
cv.cplxconj	rD[0] = rs1[0] rD[1] = -rs1[1]
cv.cplxmul.r.{/,div2,div4,div8}	rD[15:0] = (rs1[0]*rs2[0] – rs1[1]*rs2[1])>>{15,16,17,18} rD[31:16] = rD[31:16]
cv.cplxmul.i.{/,div2,div4,div8}	rD[31:16] = (rs1[0]*rs2[1] + rs1[1]*rs2[0])>>{15,16,17,18} rD[15:0] = rD[15:0]

17.6.6 SIMD Complex-numbers Encoding

31 : 27	26	25	24 : 20	19 : 15	14 :12	11 : 7	6 : 0	
funct5	F		rs2	rs1	funct3	rD	opcode	
0 1101	1	x	src2	src1	000	dest	101 0111	**cv.subrotmj rD, rs1, rs2**
0 1101	1	x	src2	src1	010	dest	101 0111	**cv.subrotmj.div2 rD, rs1, rs2**
0 1101	1	x	src2	src1	100	dest	101 0111	**cv.subrotmj.div4 rD, rs1, rs2**
0 1101	1	x	src2	src1	110	dest	101 0111	**cv.subrotmj.div8 rD, rs1, rs2**
0 1011	1	x	xxxxx	src1	000	dest	101 0111	**cv.cplxconj rD, rs1**
0 1010	1	0	src2	src1	000	dest	101 0111	**cv.cplxmul.r rD, rs1, rs2**
0 1010	1	0	src2	src1	01x	dest	101 0111	**cv.cplxmul.r.div2 rD, rs1, rs2**
0 1010	1	0	src2	src1	100	dest	101 0111	**cv.cplxmul.r.div4 rD, rs1, rs2**
0 1010	1	0	src2	src1	110	dest	101 0111	**cv.cplxmul.r.div8 rD, rs1, rs2**
0 1010	1	1	src2	src1	000	dest	101 0111	**cv.cplxmul.i rD, rs1, rs2**
0 1010	1	1	src2	src1	010	dest	101 0111	**cv.cplxmul.i.div2 rD, rs1, rs2**
0 1010	1	1	src2	src1	100	dest	101 0111	**cv.cplxmul.i.div4 rD, rs1, rs2**
0 1010	1	1	src2	src1	110	dest	101 0111	**cv.cplxmul.i.div8 rD, rs1, rs2**

CHAPTER

EIGHTEEN

CORE VERSIONS AND RTL FREEZE RULES

The CV32E41P is defined by the `marchid` and `mimpid` tuple. The tuple identify which sets of parameters have been verified by OpenHW Group, and once RTL Freeze is achieved, no further non-logically equivalent changes are allowed on that set of parameters.

The core RTL is not yet frozen, but it's kept sequentially equivalent to CV32E40P for the RV32IMC subset except for don't care states.

18.1 What happens after RTL Freeze?

18.1.1 A bug is found

If a bug is found that affect the already frozen parameter set, the RTL changes required to fix such bug are non-logically equivalent by definition. Therefore, the RTL changes are applied only on a different `mimpid` value and the bug and the fix must be documented. These changes are visible by software as the `mimpid` has a different value. Every bug or set of bugs found must be followed by another RTL Freeze release and a new GitHub tag.

18.1.2 RTL changes on unverified parameters

If changes affecting the core on a non-frozen parameter set are required, as for example, to fix bugs found in the communication to the FPU (e.g., affecting the core only if `FPU=1`), or to change the ISA Extensions decoding of PULP instructions (e.g., affecting the core only if `PULP_XPULP=1`), then such changes must remain logically equivalent for the already frozen set of parameters (except for the required `mimpid` update), and they must be applied on a different `mimpid` value. They can be non-logically equivalent to a non-frozen set of parameters. These changes are visible by software as the `mimpid` has a different value. Once the new set of parameters is verified and achieved the sign-off for RTL freeze, a new GitHub tag and version of the core is released.

18.1.3 PPA optimizations and new features

Non-logically equivalent PPA optimizations and new features are not allowed on a given set of RTL frozen parameters (e.g., a faster divider). If PPA optimizations are logically-equivalent instead, they can be applied without changing the `mimpid` value (as such changes are not visible in software). However, a new GitHub tag should be release and changes documented.

18.2 Released core versions

The verified parameter sets of the core, their implementation version, GitHub tags, and dates will be reported here.

18.3 mimpid=0

The `mimpid=0` refers to the CV32E41P core verified with the following parameters:

Name	Value
NUM_MHPMCOUNTERS	1
PULP_CLUSTER	0
PULP_XPULP	0
FPU	0
ZFINX	0
Zcea	0
Zceb	0
Zcec	0
Zcee	0

CHAPTER

NINETEEN

GLOSSARY

- **ALU**: Arithmetic/Logic Unit
- **ASIC**: Application-Specific Integrated Circuit
- **Byte**: 8-bit data item
- **CPU**: Central Processing Unit, processor
- **CSR**: Control and Status Register
- **Custom extension**: Non-Standard extension to the RISC-V base instruction set (RISC-V Instruction Set Manual, Volume I: User-Level ISA)
- **EXE**: Instruction Execute
- **FPGA**: Field Programmable Gate Array
- **FPU**: Floating Point Unit
- **Halfword**: 16-bit data item
- **Halfword aligned address**: An address is halfword aligned if it is divisible by 2
- **ID**: Instruction Decode
- **IF**: Instruction Fetch (*Instruction Fetch*)
- **ISA**: Instruction Set Architecture
- **KGE**: kilo gate equivalents (NAND2)
- **LSU**: Load Store Unit (*Load-Store-Unit (LSU)*)
- **M-Mode**: Machine Mode (RISC-V Instruction Set Manual, Volume II: Privileged Architecture)
- **OBI**: Open Bus Interface
- **PC**: Program Counter
- **PULP platform**: Parallel Ultra Low Power Platform (<https://pulp-platform.org>)
- **RV32C**: RISC-V Compressed (C extension)
- **RV32F**: RISC-V Floating Point (F extension)
- **SIMD**: Single Instruction/Multiple Data
- **Standard extension**: Standard extension to the RISC-V base instruction set (RISC-V Instruction Set Manual, Volume I: User-Level ISA)
- **WARL**: Write Any Values, Reads Legal Values
- **WB**: Write Back of instruction results

- **WLRL**: Write/Read Only Legal Values
- **Word**: 32-bit data item
- **Word aligned address**: An address is word aligned if it is divisible by 4
- **WPRI**: Reserved Writes Preserve Values, Reads Ignore Values