

---

# **CV32E40P User Manual**

**OpenHW Group**

**Nov 16, 2022**



## CONTENTS:

<b>1</b>	<b>Changelog</b>	<b>1</b>
1.1	cv32e40p_v1.1.0 . . . . .	1
1.2	cv32e40p_v1.0.0: . . . . .	1
1.3	pulpassimo-v1.0.0: . . . . .	1
1.4	pulpino-v1.0.0: . . . . .	1
<b>2</b>	<b>Introduction</b>	<b>3</b>
2.1	License . . . . .	3
2.2	Bus Interfaces . . . . .	4
2.3	Standards Compliance . . . . .	4
2.4	Synthesis guidelines . . . . .	5
2.4.1	ASIC Synthesis . . . . .	5
2.4.2	FPGA Synthesis . . . . .	6
2.5	Verification . . . . .	6
2.6	Contents . . . . .	7
2.7	History . . . . .	7
2.7.1	Memory-Protocol . . . . .	7
2.7.2	RV32F Extensions . . . . .	8
2.7.3	RV32A Extensions, Security and Memory Protection . . . . .	8
2.7.4	CSR Address Re-Mapping . . . . .	8
2.7.5	Interrupts . . . . .	8
2.7.6	PULP HWLoop Spec . . . . .	8
2.7.7	Compliance, bug fixing, code clean-up, and documentation . . . . .	8
2.8	References . . . . .	8
2.9	Contributors . . . . .	9
<b>3</b>	<b>Getting Started with CV32E40P</b>	<b>11</b>
3.1	Register File . . . . .	11
3.2	Clock Gating Cell . . . . .	11
<b>4</b>	<b>Core Integration</b>	<b>13</b>
4.1	Instantiation Template . . . . .	13
4.2	Parameters . . . . .	14
4.3	Interfaces . . . . .	15
<b>5</b>	<b>Pipeline Details</b>	<b>17</b>
5.1	Multi- and Single-Cycle Instructions . . . . .	18
5.2	Hazards . . . . .	19
<b>6</b>	<b>Instruction Fetch</b>	<b>21</b>
6.1	Misaligned Accesses . . . . .	21

6.2	Protocol . . . . .	22
<b>7</b>	<b>Load-Store-Unit (LSU)</b>	<b>25</b>
7.1	Misaligned Accesses . . . . .	25
7.2	Protocol . . . . .	26
7.3	Post-Incrementing Load and Store Instructions . . . . .	30
<b>8</b>	<b>Register File</b>	<b>31</b>
8.1	Flip-Flop-Based Register File . . . . .	31
8.2	Latch-based Register File . . . . .	31
8.3	FPU Register File . . . . .	32
<b>9</b>	<b>Auxiliary Processing Unit (APU)</b>	<b>33</b>
9.1	Auxiliary Processing Unit Interface . . . . .	33
9.2	Protocol . . . . .	33
9.3	Connection with the FPU . . . . .	33
9.4	APU Tracer . . . . .	34
9.5	Output file . . . . .	34
9.6	Trace output format . . . . .	34
<b>10</b>	<b>Floating Point Unit (FPU)</b>	<b>35</b>
10.1	FP CSR . . . . .	35
<b>11</b>	<b>Sleep Unit</b>	<b>37</b>
11.1	Startup behavior . . . . .	38
11.2	WFI . . . . .	38
11.3	PULP Cluster Extension . . . . .	39
<b>12</b>	<b>CORE-V Hardware Loop Extensions</b>	<b>41</b>
12.1	Hardware Loop constraints . . . . .	41
<b>13</b>	<b>Control and Status Registers</b>	<b>43</b>
13.1	CSR Map . . . . .	43
13.2	CSR Descriptions . . . . .	45
13.2.1	Floating-point accrued exceptions (fflags) . . . . .	45
13.2.2	Floating-point dynamic rounding mode (frm) . . . . .	45
13.2.3	Floating-point control and status register (fcsr) . . . . .	46
13.2.4	HWLoop Start Address 0/1 (lpstart0/1) . . . . .	46
13.2.5	HWLoop End Address 0/1 (lpend0/1) . . . . .	46
13.2.6	HWLoop Count Address 0/1 (lpcount0/1) . . . . .	46
13.2.7	Privilege Level (privlv) . . . . .	47
13.2.8	User Hardware Thread ID (uhartid) . . . . .	47
13.2.9	Machine Status (mstatus) . . . . .	47
13.2.10	Machine ISA (misa) . . . . .	48
13.2.11	Machine Interrupt Enable Register (mie) . . . . .	49
13.2.12	Machine Trap-Vector Base Address (mtvec) . . . . .	49
13.2.13	Machine Counter-Inhibit Register (mcountinhibit) . . . . .	49
13.2.14	Machine Performance Monitoring Event Selector (mhpmevent3 .. mhpmevent31) . . . . .	50
13.2.15	Machine Scratch (mscratch) . . . . .	50
13.2.16	Machine Exception PC (mepc) . . . . .	50
13.2.17	Machine Cause (mcause) . . . . .	50
13.2.18	Machine Trap Value (mtval) . . . . .	51
13.2.19	Machine Interrupt Pending Register (mip) . . . . .	51
13.2.20	Trigger Select Register (tselect) . . . . .	51
13.2.21	Trigger Data Register 1 (tdata1) . . . . .	51

13.2.22	Trigger Data Register 2 (tdata2)	52
13.2.23	Trigger Data Register 3 (tdata3)	52
13.2.24	Trigger Info (tinfo)	53
13.2.25	Machine Context Register (mcontext)	53
13.2.26	Supervisor Context Register (scontext)	53
13.2.27	Debug Control and Status (dcsr)	54
13.2.28	Debug PC (dpc)	54
13.2.29	Debug Scratch Register 0/1 (dscratch0/1)	55
13.2.30	Machine Cycle Counter (mcycle)	55
13.2.31	Machine Instructions-Retired Counter (minstret)	55
13.2.32	Machine Performance Monitoring Counter (mhpmpcounter3 .. mhpmpcounter31)	55
13.2.33	Upper 32 Machine Cycle Counter (mcycleh)	56
13.2.34	Upper 32 Machine Instructions-Retired Counter (minstreth)	56
13.2.35	Upper 32 Machine Performance Monitoring Counter (mhpmpcounter3h .. mhpmpcounter31h)	56
13.2.36	Machine Vendor ID (mvendorid)	56
13.2.37	Machine Architecture ID (marchid)	57
13.2.38	Machine Implementation ID (mimpid)	57
13.2.39	Hardware Thread ID (mhartid)	57
13.3	Cycle Counter (cycle)	57
13.4	Instructions-Retired Counter (instret)	58
13.5	Performance Monitoring Counter (hpmcounter3 .. hpmcounter31)	58
13.6	Upper 32 Cycle Counter (cycleh)	58
13.7	Upper 32 Instructions-Retired Counter (instreth)	59
13.8	Upper 32 Performance Monitoring Counter (hpmcounter3h .. hpmcounter31h)	59
<b>14</b>	<b>Performance Counters</b>	<b>61</b>
14.1	Event Selector	61
14.2	Controlling the counters from software	62
14.3	Parametrization at synthesis time	62
14.4	Time Registers (time(h))	63
<b>15</b>	<b>Exceptions and Interrupts</b>	<b>65</b>
15.1	Interrupt Interface	65
15.2	Interrupts	66
15.3	Exceptions	66
15.4	Nested Interrupt/Exception Handling	66
<b>16</b>	<b>Debug &amp; Trigger</b>	<b>69</b>
16.1	Interface	70
16.2	Core Debug Registers	70
16.3	Debug state	71
16.4	EBREAK Behavior	73
16.4.1	Scenario 1 : Enter Exception	73
16.4.2	Scenario 2 : Enter Debug Mode	73
16.4.3	Scenario 3 : Exit Program Buffer & Restart Debug Code	74
16.5	Interrupts during Single-Step Behavior	74
<b>17</b>	<b>Tracer</b>	<b>75</b>
17.1	Output file	75
17.2	Trace output format	75
<b>18</b>	<b>CORE-V Instruction Set Extensions</b>	<b>77</b>
18.1	Post-Incrementing Load & Store Instructions and Register-Register Load & Store Instructions	77
18.1.1	Load Operations	78
18.1.2	Store Operations	79

18.2	Event Load Instructions . . . . .	80
18.2.1	Load Operations . . . . .	80
18.3	Hardware Loops . . . . .	81
18.3.1	Operations . . . . .	81
18.4	ALU . . . . .	82
18.4.1	Bit Reverse Instruction . . . . .	82
18.4.2	Bit Manipulation Operations . . . . .	84
18.4.3	Bit Manipulation Encoding . . . . .	85
18.4.4	General ALU Operations . . . . .	85
18.4.5	General ALU Encoding . . . . .	87
18.4.6	Immediate Branching Operations . . . . .	88
18.4.7	Immediate Branching Encoding . . . . .	88
18.5	Multiply-Accumulate . . . . .	89
18.5.1	MAC Operations . . . . .	89
18.5.2	MAC Encoding . . . . .	90
18.6	SIMD . . . . .	91
18.6.1	SIMD ALU Operations . . . . .	92
18.6.2	SIMD Bit Manipulation Operations . . . . .	92
18.6.3	SIMD ALU Encoding . . . . .	95
18.6.4	SIMD Comparison Operations . . . . .	98
18.6.5	SIMD Comparison Encoding . . . . .	98
18.6.6	SIMD Complex-number Operations . . . . .	100
18.6.7	SIMD Complex-numbers Encoding . . . . .	100
<b>19</b>	<b>Core Versions and RTL Freeze Rules</b>	<b>101</b>
19.1	What happens after RTL Freeze? . . . . .	101
19.1.1	RTL changes on verified parameters . . . . .	101
19.1.2	A bug is found . . . . .	101
19.1.3	RTL changes on non-verified yet parameters . . . . .	101
19.1.4	PPA optimizations and new features . . . . .	102
19.2	Released core versions . . . . .	102
19.3	mimpid=0 . . . . .	102
<b>20</b>	<b>Glossary</b>	<b>105</b>

## CHANGELOG

### 1.1 cv32e40p\_v1.1.0

*Released on 2022-11-14 - [GitHub](#)*

### 1.2 cv32e40p\_v1.0.0:

*Released on 2020-12-10 - [GitHub](#)*

### 1.3 pulpissimo-v1.0.0:

*Released on 2018-01-23 - [GitHub](#)*

### 1.4 pulpino-v1.0.0:

*Released on 2018-01-23 - [GitHub](#)*





The diagram illustrates the CV32E40P processor architecture, which is a 32-bit RISC-V processor. The main components and their interconnections are as follows:

- Instruction Interface:** The external interface for instructions, connected to the **prefetch buffer**.
- Instruction Path:**
  - prefetch buffer** feeds into the **aligner**.
  - aligner** feeds into the **compress decoder**.
  - compress decoder** feeds into the **IF ID** stage.
- IF ID Stage:** A vertical purple bar representing the Instruction Fetch and Instruction Decode stages.
- Decoder:** Receives input from the IF ID stage and feeds into the **register file**.
- Register File:** A central component with multiple ports (rA, rB, rC, rD, rE, rF, rG, rH, rI, rJ, rK, rL, rM, rN, rO, rP, rQ, rR, rS, rT, rU, rV, rW, rX, rY, rZ, r31) and control signals (DA, DB, DC, DE, DF, DG, DH, DI, DJ, DK, DL, DM, DN, DO, DP, DQ, DR, DS, DT, DU, DV, DW, DX, DY, DZ, D31).
- hwloop regs:** Hardware loop registers connected to the register file and the **controller**.
- PC RF, EX WB, IM:** A vertical purple bar representing the Program Counter, Register File, Execution, Write Back, and Instruction Memory stages.
- Execution Path:**
  - OpA, OpB, OpC:** Operation codes for the **LSU**, **CSR**, **ALU**, and **MULT** units.
  - RD:** Register destination for the **CSR**, **ALU**, and **MULT** units.
  - OpA, OpB, OpC:** Operation codes for the **CSR**, **ALU**, and **MULT** units.
  - RD:** Register destination for the **CSR**, **ALU**, and **MULT** units.
  - OpA, OpB, OpC:** Operation codes for the **CSR**, **ALU**, and **MULT** units.
  - RD:** Register destination for the **CSR**, **ALU**, and **MULT** units.
- EX WB Stage:** A vertical purple bar representing the Execution and Write Back stages.
- Data Interface:** The external interface for data, connected to the **EX WB** stage.
- Controller:** A red horizontal bar at the bottom, connected to the **IF ID** stage, **PC RF, EX WB, IM** stage, **hwloop regs**, **register file**, **LSU**, **CSR**, **ALU**, **MULT**, and **sleep unit**.
- Sleep Unit:** A red rounded rectangle at the bottom right, connected to the **controller**.
- Debug Interface:** A blue rounded rectangle at the bottom left, connected to the **controller**.
- Interrupt Interface:** A blue rounded rectangle at the bottom right, connected to the **controller**.

## 2.2 Bus Interfaces

The Instruction Fetch and Load/Store data bus interfaces are 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. Additional information can be found in the *Instruction Fetch* and *Load-Store-Unit (LSU)* chapters of this document.

The Auxiliary Processing Unit bus interface is derived from to the OBI (Open Bus Interface) protocol, see the *Auxiliary Processing Unit (APU)* chapter of this document.

## 2.3 Standards Compliance

CV32E40P 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). CV32E40P implements the Machine ISA version 1.11.
- RISC-V External Debug Support, draft version 0.13.2 (commit 4e0bb0fc2d843473db2356623792c6b7603b94d4 of <https://github.com/riscv/riscv-debug-spec/>)

Many features in the RISC-V specification are optional, and CV32E40P can be parameterized to enable or disable some of them.

CV32E40P 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 2.1: CV32E40P Standard Instruction Set Extensions

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

The following custom instruction set extensions are available.

Table 2.2: CV32E40P Custom Instruction Set Extensions

Custom Extension	Ver- sion	Configurability
<b>Xcorev</b> : CORE-V ISA Extensions (excluding <b>cv.elw</b> )	1.0	optionally enabled with the PULP_XPULP parameter
<b>Xpulpcluster</b> : PULP Cluster Extension	1.0	optionally enabled with the PULP_CLUSTER parameter
<b>Xpulpzfinx</b> : PULP Share Integer (X) Registers with Floating Point (F) Register Extension	1.0	optionally enabled with the PULP_ZFINX parameter

Most content of the RISC-V privileged specification is optional. CV32E40P 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*

## 2.4 Synthesis guidelines

The CV32E40P 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*). However, the use of the flip-flop-based register-file is the one suggested and used by default as it has been verified. 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/cv32e40p_sim_clock_gate.sv` (see *Clock Gating Cell*).

The `constraints/cv32e40p_core.sdc` file provides an example of synthesis constraints.

### 2.4.1 ASIC Synthesis

ASIC synthesis is supported for CV32E40P. 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.

## 2.4.2 FPGA Synthesis

FPGA synthesis is only supported for CV32E40P 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*.

## 2.5 Verification

The verification environment (testbenches, testcases, etc.) for the CV32E40P core can be found at [core-v-verif](#). It is recommended that you start by reviewing the [CORE-V Verification Strategy](#).

In early 2021 the CV32E40P achieved Functional RTL Freeze, meaning that it has been fully verified as per its [Verification Plan](#). The top-level [README](#) in [core-v-verif](#) has a link to the final functional, code and test coverage reports.

The unofficial start date for the CV32E40P verification effort is 2020-02-27, which is the date the core-v-verif environment “went live”. Between then and RTL Freeze, a total of 47 RTL issues and 38 User Manual issues were identified and resolved<sup>1</sup>. A breakdown of the RTL issues is as follows:

Table 2.3: How RTL Issues Were Found

“Found By”	Count	Note
Simulation	18	See classification below
Inspection	13	Human review of the RTL
Formal Verification	13	This includes both Designer and Verifier use of FV
Lint	2	
Unknown	1	

A classification of the simulation issues by method used to identify them is informative:

Table 2.4: Breakdown of Issues found by Simulation

Simulation Method	Count	Note
Directed, self-checking test	10	Many test supplied by Design team and a couple from the Open Source Community at large
Step & Compare	6	Issues directly attributed to S&C against ISS
Constrained-Random	2	Test generated by corev-dv (extension of riscv-dv)

A classification of the issues themselves:

Table 2.5: Issue Classification

Issue Type	Count	Note
RTL Functional	40	A bug!
RTL coding style	4	Linter issues, removing TODOs, removing `ifdefs, etc.
Non-RTL functional	1	Issue related to behavioral tracer (not part of the core)
Unreproducible	1	
Invalid	1	

Additional details are available as part of the [CV32E40P v1.0.0 Report](#).

<sup>1</sup> It is a testament on the quality of the work done by the PULP platform team that it took a team of professional verification engineers more than 9 months to find all these issues.

## 2.6 Contents

- *Getting Started with CV32E40P* discusses the requirements and initial steps to start using CV32E40P.
- *Core Integration* provides the instantiation template and gives descriptions of the design parameters as well as the input and output ports.
- *Pipeline Details* described the overall pipeline structure.
- The instruction and data interfaces of CV32E40P 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 CV32E40P.
- *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.

## 2.7 History

CV32E40P started its life as a fork of the OR10N CPU core based on the OpenRISC ISA. Then, under the name of RI5CY, it became a RISC-V core (2016), and it has been maintained by the PULP platform <<https://pulp-platform.org>> team until February 2020, when it has been contributed to OpenHW Group <https://www.openhwgroup.org>.

As RI5CY has been used in several projects, a list of all the changes made by OpenHW Group since February 2020 follows:

### 2.7.1 Memory-Protocol

The Instruction and Data memory interfaces are now compliant with the OBI protocol (see <https://github.com/openhwgroup/core-v-docs/blob/master/cores/obi/OBI-v1.2.pdf>). Such memory interface is slightly different from the one used by RI5CY as: the grant signal can now be kept high by the bus even without the core raising a request; and the request signal does not depend anymore on the rvalid signal (no combinatorial dependency). The OBI is easier to be interfaced to the AMBA AXI and AHB protocols and improves timing as it removes rvalid->req dependency. Also, the protocol forces the address stability. Thus, the core can not retract memory requests once issued, nor can it change the issued address (as was the case for the RI5CY instruction memory interface).

## 2.7.2 RV32F Extensions

The FPU is not instantiated in the core EX stage anymore, and it must be attached to the APU interface. Previously, RI5CY could select with a parameter whether the FPU was instantiated inside the EX stage or via the APU interface.

## 2.7.3 RV32A Extensions, Security and Memory Protection

CV32E40P core does not support the RV32A (atomic) extensions, the U-mode, and the PMP anymore. Most of the previous RTL descriptions of these features have been kept but not maintained. The RTL code has been partially kept to allow previous users of these features to develop their own by reusing previously developed RI5CY modules.

## 2.7.4 CSR Address Re-Mapping

CV32E40P is fully compliant with RISC-V. RI5CY used to have custom performance counters 32b wide (not compliant with RISC-V) in the CSR address space {0x7A0, 0x7A1, 0x780-0x79F}. CV32E40P is fully compliant with the RISC-V spec. The custom PULP HWLoop CSRs moved from the 0x7C\* to RISC-V user custom space 0x80\* address space.

## 2.7.5 Interrupts

RI5CY used to have a req plus a 5bits ID interrupt interface, supporting up to 32 interrupt requests (only one active at a time), with the priority defined outside in an interrupt controller. CV32E40P is now compliant with the CLINT RISC-V spec, extended with 16 custom interrupts lines called fast, for a total of 19 interrupt lines. They can be all active simultaneously, and priority and per-request interrupt enable bit is controlled by the core CLINT definition.

## 2.7.6 PULP HWLoop Spec

RI5CY supported two nested HWLoops. Every loop had a minimum of two instructions. The start and end of the loop addresses could be misaligned, and the instructions in the loop body could be of any kind. CV32E40P has a more restricted spec for the HWLoop (see *CORE-V Hardware Loop Extensions*).

## 2.7.7 Compliancy, bug fixing, code clean-up, and documentation

The CV32E40P has been verified. It is fully compliant with RISC-V (RI5CY was partially compliant). Many bugs have been fixed, and the RTL code cleaned-up. The documentation has been formatted with reStructuredText and has been developed following at industrial quality level.

## 2.8 References

1. Gautschi, Michael, et al. “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices.” in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 10, pp. 2700-2713, Oct. 2017
2. Schiavone, Pasquale Davide, et al. “Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications.” 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS 2017)

## 2.9 Contributors

Andreas Traber (\*[atraber@iis.ee.ethz.ch](mailto:atraber@iis.ee.ethz.ch)\*)  
Michael Gautschi (\*[gautschi@iis.ee.ethz.ch](mailto:gautschi@iis.ee.ethz.ch)\*)  
Pasquale Davide Schiavone (\*[pschiavo@iis.ee.ethz.ch](mailto:pschiavo@iis.ee.ethz.ch)\*)

Arjan Bink (\*[arjan.bink@silabs.com](mailto:arjan.bink@silabs.com)\*)  
Paul Zavalney (\*[paul.zavalney@silabs.com](mailto:paul.zavalney@silabs.com)\*)

Pascal Gouédo (\*[pascal.gouedo@dolphin.fr](mailto:pascal.gouedo@dolphin.fr)\*)

Micrel Lab and Multitherman Lab  
University of Bologna, Italy

Integrated Systems Lab  
ETH Zürich, Switzerland





## GETTING STARTED WITH CV32E40P

This page discusses initial steps and requirements to start using CV32E40P in your design.

### 3.1 Register File

CV32E40P comes with two different register file implementations. Depending on the target technology, either the implementation in `cv32e40p_register_file_ff.sv` or the one in `cv32e40p_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](#).

### 3.2 Clock Gating Cell

CV32E40P 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 `cv32e40p_sim_clock_gate.sv`. This file contains a module called `cv32e40p_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 CV32E40P, clock gating cells are used both in `cv32e40p_sleep_unit.sv` and `cv32e40p_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 `cv32e40p_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 `cv32e40p_clock_gate` module using design primitives that are appropriate for the intended synthesis target technology.



## CORE INTEGRATION

The main module is named `cv32e40p_core` and can be found in `cv32e40p_core.sv`. Below, the instantiation template is given and the parameters and interfaces are described.

### 4.1 Instantiation Template

```
cv32e40p_core #(
    .FPU                      ( 0 ),
    .NUM_MHPMCOUNTERS        ( 1 ),
    .PULP_CLUSTER             ( 0 ),
    .PULP_XPULP               ( 0 ),
    .PULP_ZFINX               ( 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                 (),
    .data_rvalid_i              (),
    .data_addr_o                (),
    .data_be_o                  (),
    .data_wdata_o               (),
```

(continues on next page)

(continued from previous page)

```

.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    ()
);

```

## 4.2 Parameters

---

**Note:** The non-default (i.e. non-zero) settings of FPU, PULP\_CLUSTER, PULP\_XPULP and PULP\_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/cv32e40p/issues/452>.

---

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

## 4.3 Interfaces

Sig-nal(s)	Width	Dir	Description
clk_i	1	in	Clock signal
rst_ni	1	in	Active-low asynchronous reset
scan_cg_en_i		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 <i>fetch_enable_i</i>
mtvec_addr_i	32	in	mtvec address. Initial value for the address part of <i>Machine Trap-Vector Base Address (mtvec)</i> . Do not change after enabling core via <i>fetch_enable_i</i>
dm_halt_addr_i	32	in	Address to jump to when entering Debug Mode, see <i>Debug &amp; Trigger</i> . Must be word-aligned. Do not change after enabling core via <i>fetch_enable_i</i>
dm_exception_addr_i	32	in	Address to jump to when an exception occurs when executing code during Debug Mode, see <i>Debug &amp; Trigger</i> . Must be word-aligned. Do not change after enabling core via <i>fetch_enable_i</i>
hart_id_i	32	in	Hart ID, usually static, can be read from <i>Hardware Thread ID (mhartid)</i> and <i>User Hardware Thread ID (uhartid)</i> CSRs
instr_*	Instruction fetch interface, see <i>Instruction Fetch</i>		
data_*	Load-store unit interface, see <i>Load-Store-Unit (LSU)</i>		
apu_*	Auxiliary Processing Unit (APU) interface, see <i>Auxiliary Processing Unit (APU)</i>		
irq_*	Interrupt inputs, see <i>Exceptions and Interrupts</i>		
debug_*	Debug interface, see <i>Debug &amp; Trigger</i>		
fetch_enable_i		in	Enable the instruction fetch of CV32E40P. The first instruction fetch after reset de-assertion will not happen as long as this signal is 0. <i>fetch_enable_i</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 <i>fetch_enable_i</i> is ignored.
core_sleep_o		out	Core is sleeping, see <i>Sleep Unit</i> .
pulp_clock_en_i		in	PULP clock enable (only used when PULP_CLUSTER = 1, tie to 0 otherwise), see <i>Sleep Unit</i>



## PIPELINE DETAILS

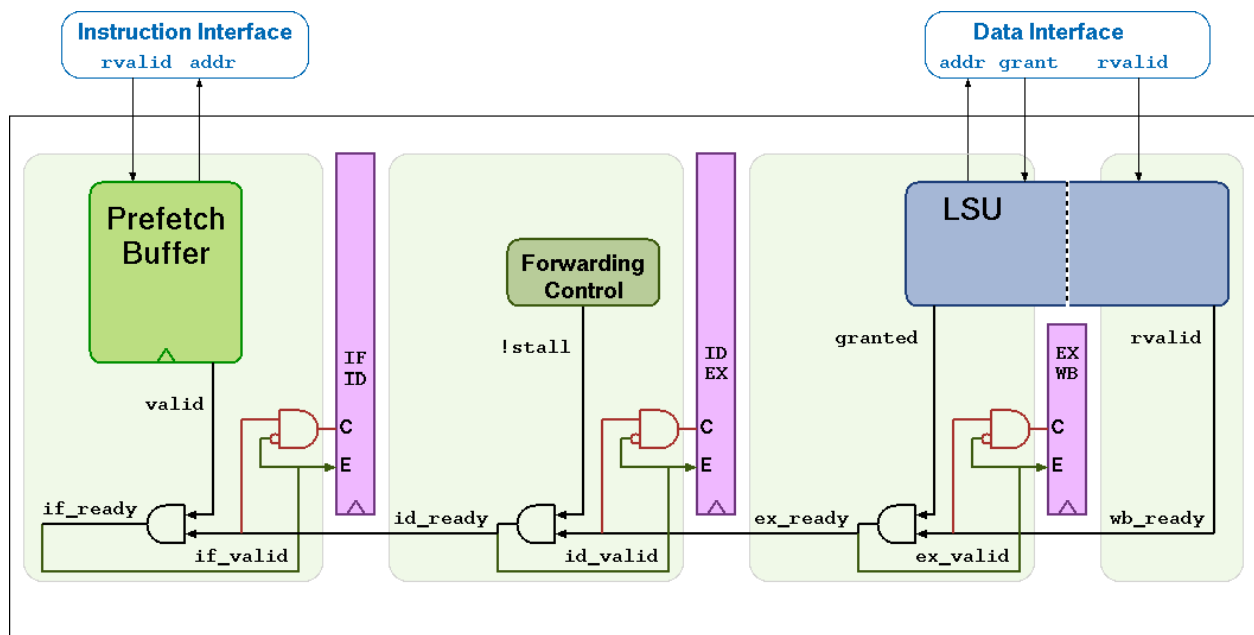


Figure 5.1: CV32E40P Pipeline

CV32E40P 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. The IF stage also pre-decodes RVC instructions into RV32I base instructions. 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.

## 5.1 Multi- and Single-Cycle Instructions

Table 5.1 shows the cycle count per instruction type. Some instructions have a variable time, this is indicated as a range e.g. 1..32 means that the instruction takes a minimum of 1 cycle and a maximum of 32 cycles. The cycle counts assume zero stall on the instruction-side interface and zero stall on the data-side memory interface.

Table 5.1: Cycle counts per instruction type

In-struction Type	Cycles	Description
Integer Computational	1	Integer Computational Instructions are defined in the RISC-V RV32I Base Integer Instruction Set.
CSR Access	4 (mstatus, mepc, mtvec, mcause, mcycle, minstret, mhpcounter*, mcycleh, minstreth, mhpcounter*h, mcountinhibit, mhpmevent*, dscr, dpc, dscratch0, dscratch1, privlv) 1 (all the other CSRs)	CSR Access Instructions are defined in ‘Zicsr’ of the RISC-V specification.
Load/Store	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 <b>cv.elw</b> takes 4 cycles.
Multiplication	1 (mul) 5 (mulh, mulhsu, mulhu)	CV32E40P 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.



## 5.2 Hazards

The CV32E40P 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)



## INSTRUCTION FETCH

The Instruction Fetch (IF) stage of the CV32E40P 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 be 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, CV32E40P 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 6.1 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 6.1: 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.

### 6.1 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.

## 6.2 Protocol

The CV32E40P 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 CV32E40P instruction fetch interface can cause up to two outstanding transactions.

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

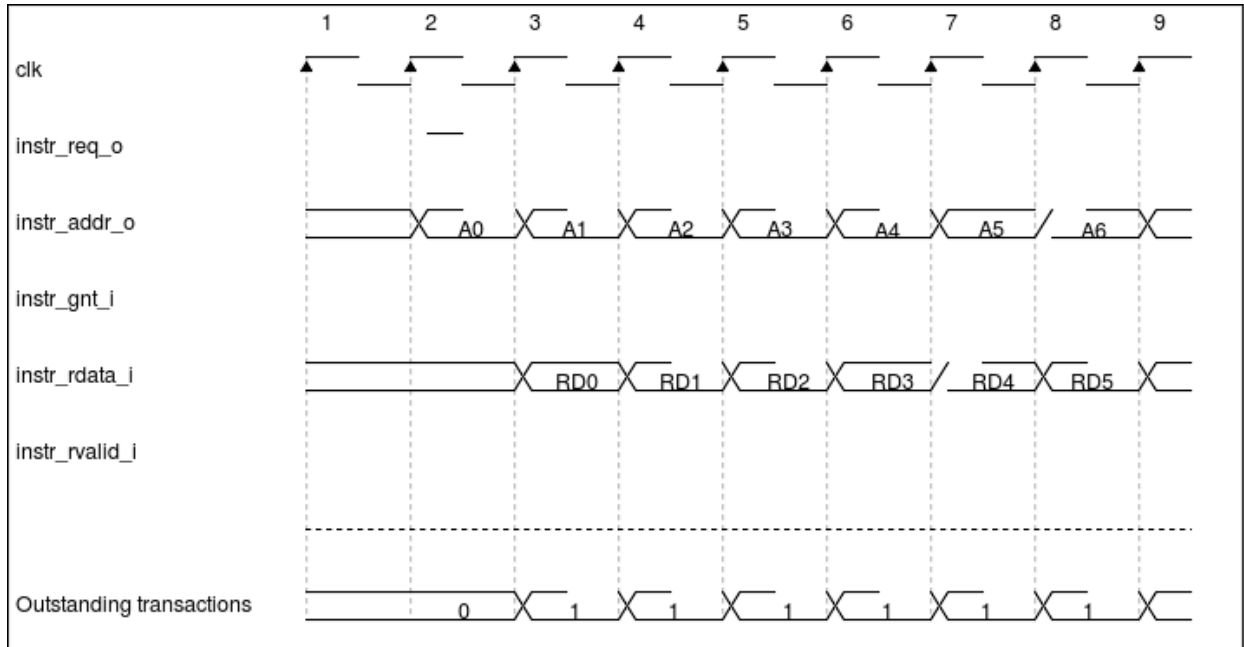


Figure 6.1: Back-to-back Memory Transactions

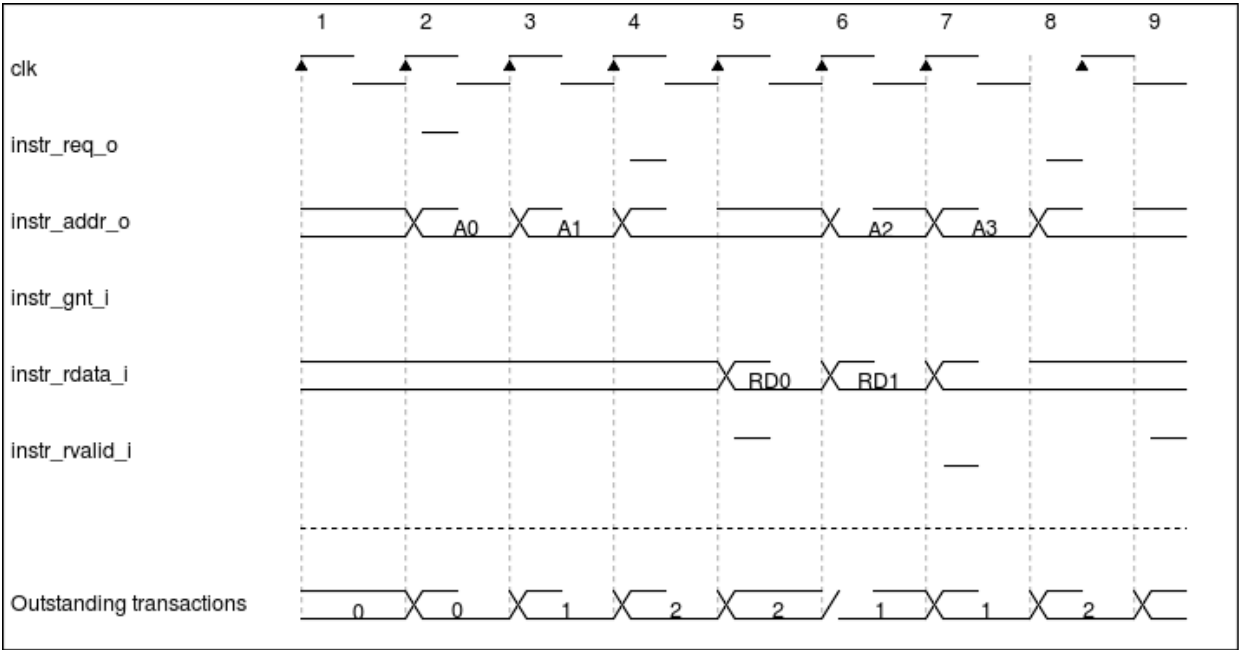


Figure 6.2: Multiple Outstanding Memory Transactions



## 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 7.1 describes the signals that are used by the LSU.

Table 7.1: LSU interface signals

Signal	Di- rec- tion	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.

### 7.1 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.

## 7.2 Protocol

The CV32E40P 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 CV32E40P 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 7.1, Figure 7.2, Figure 7.3 and Figure 7.4 show example timing diagrams of the protocol.

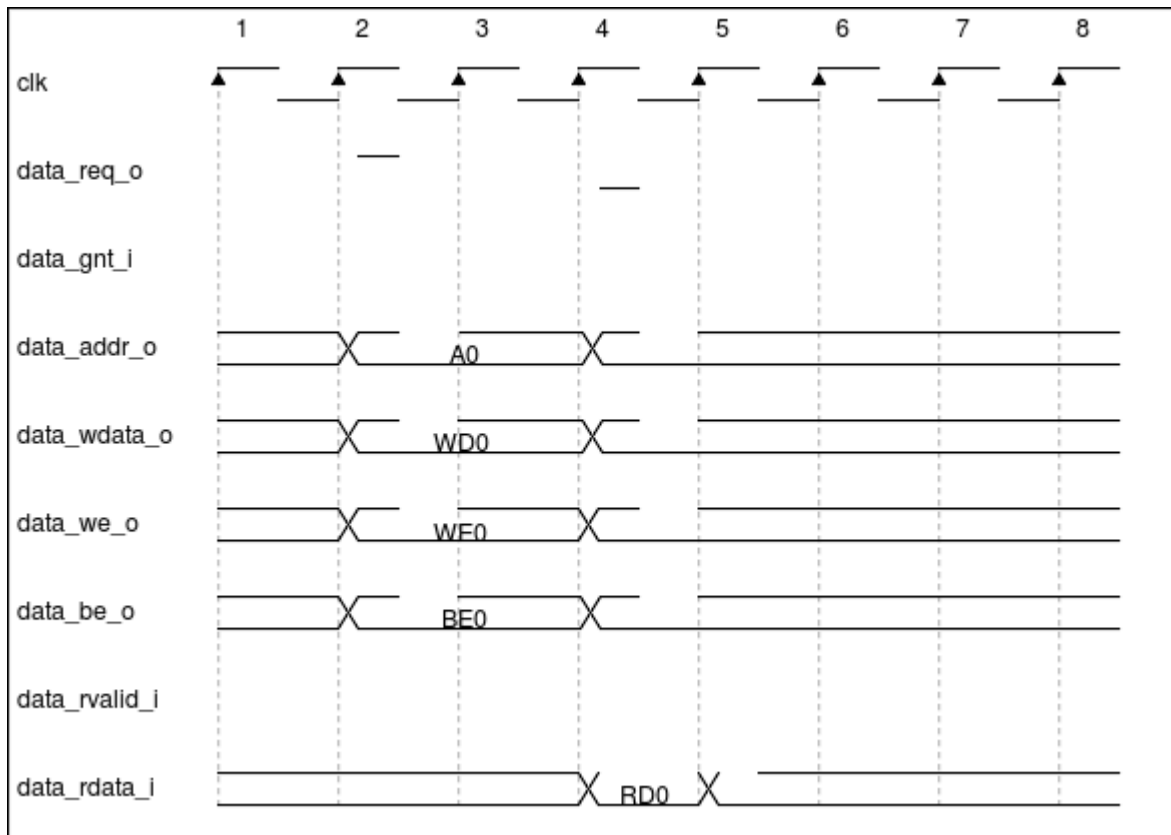


Figure 7.1: Basic Memory Transaction



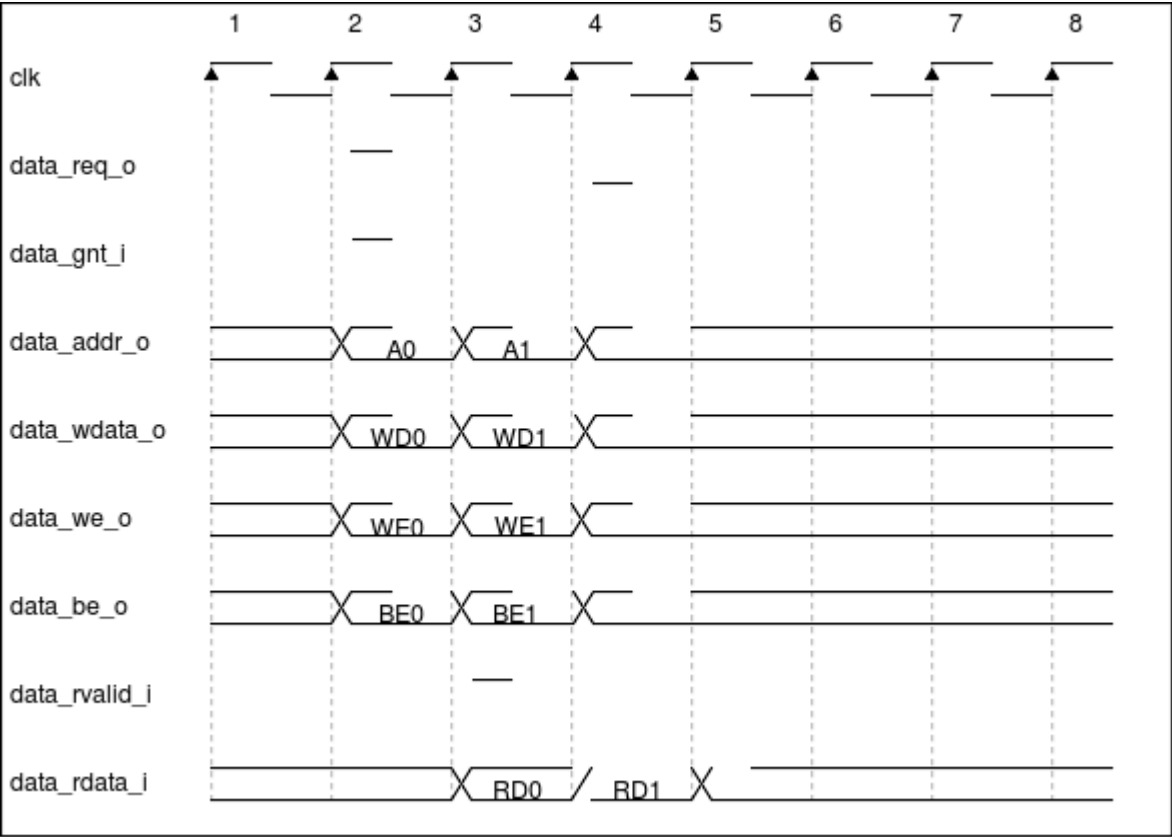


Figure 7.2: Back-to-back Memory Transactions

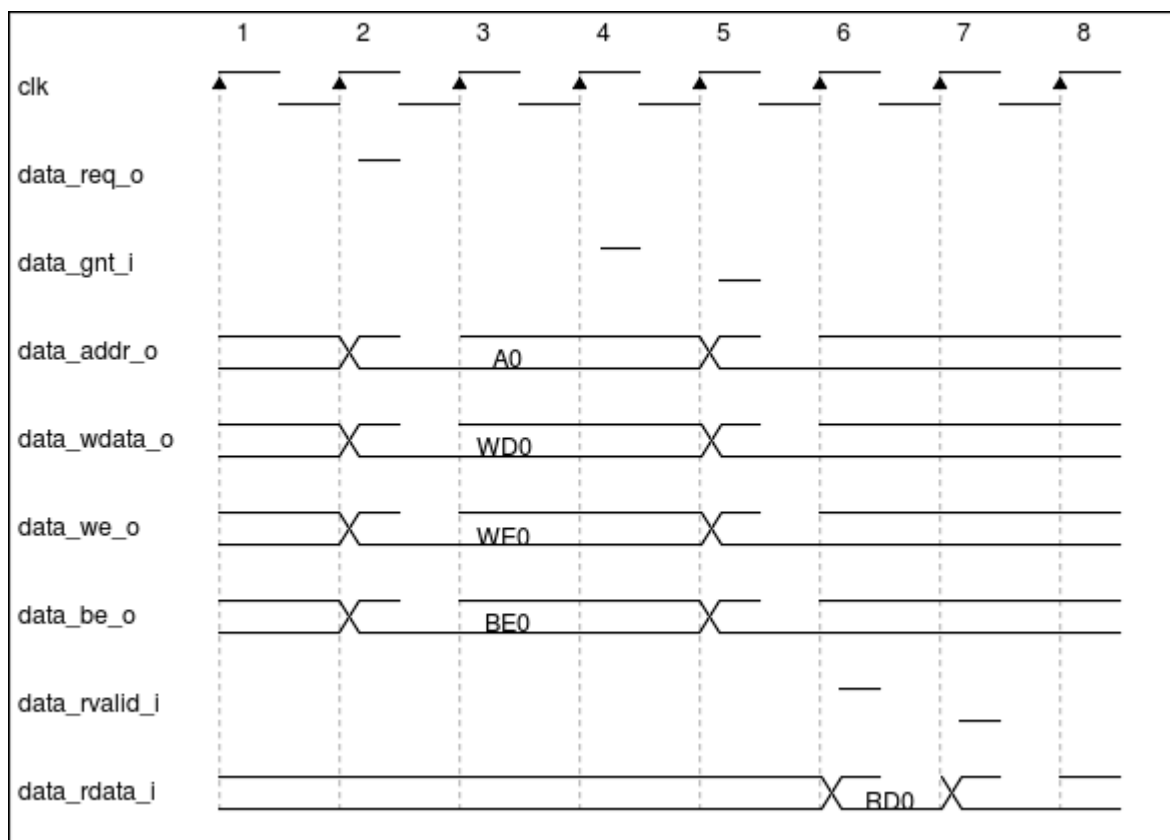


Figure 7.3: Slow Response Memory Transaction

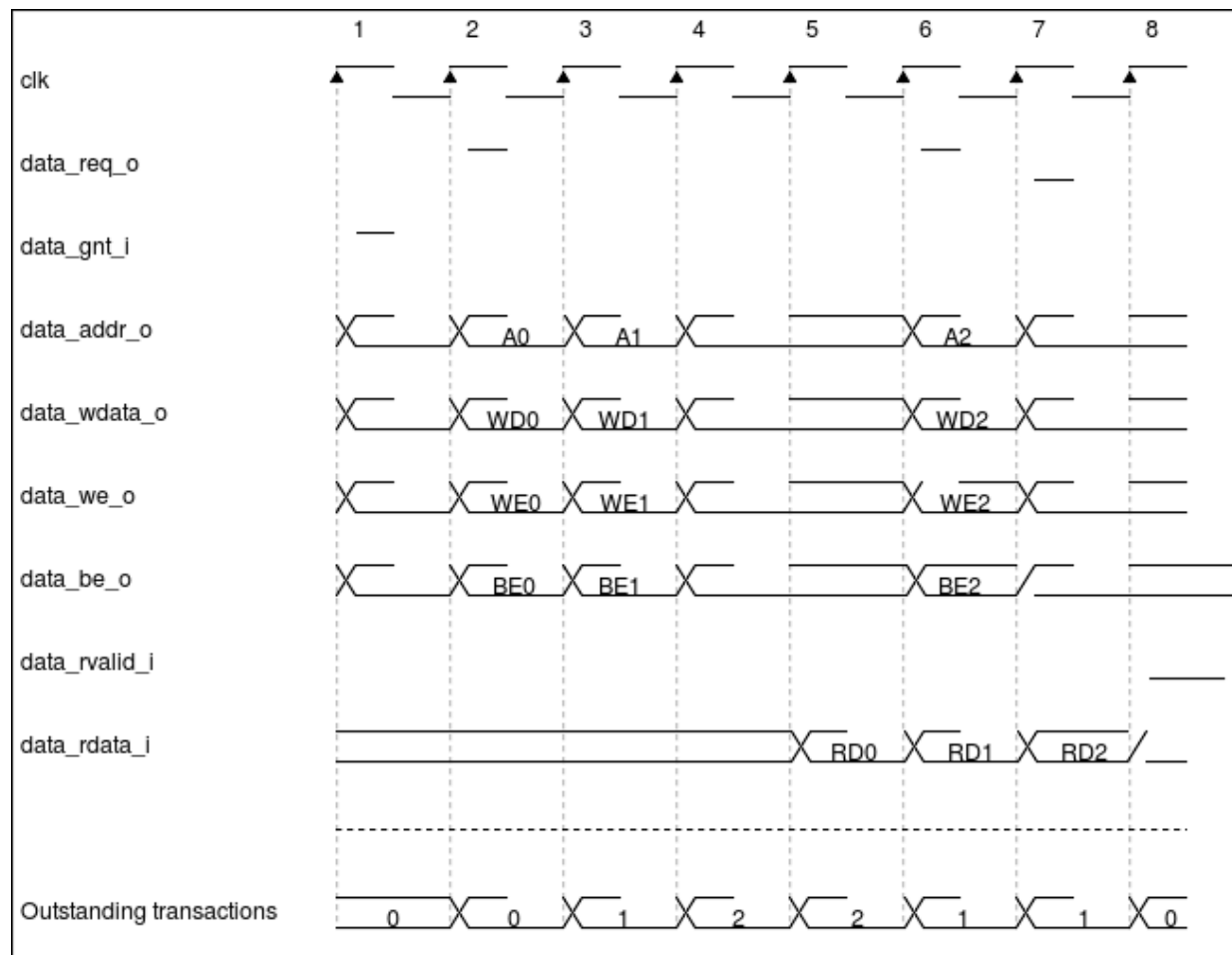


Figure 7.4: Multiple Outstanding Memory Transactions

## 7.3 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.

## REGISTER FILE

Source files: `rtl/cv32e40p_register_file_ff.sv` `rtl/cv32e40p_register_file_latch.sv`

CV32E40P 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/cv32e40p_register_file_ff.sv`)
- Latch-based (`rtl/cv32e40p_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.

### 8.1 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 `cv32e40p_register_file_ff.sv` in your project.

### 8.2 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 `cv32e40p_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 `cv32e40p_clock_gate`. For more information regarding the clock gating cell, checkout *Getting Started with CV32E40P*.

## 8.3 FPU Register File

If the optional FPU is instantiated, unless PULP\_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 signal 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.

## AUXILIARY PROCESSING UNIT (APU)

### 9.1 Auxiliary Processing Unit Interface

Table 9.1 describes the signals of the Auxiliary Processing Unit interface.

Table 9.1: 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]	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

### 9.2 Protocol

The CV32E40P 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 CV32E40P apu interface can cause up to two outstanding transactions.

### 9.3 Connection with the FPU

The CV32E40P 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.

## 9.4 APU Tracer

The module `cv32e40p_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 `cv32e40p_core`. It can be enabled during simulation by defining `CV32E40P_APU_TRACE`.

## 9.5 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.

## 9.6 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.



## 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 `cv32e40p_core.sv` to 1. This will extend the CV32E40P decoder accordingly. The actual Floating Point Unit (FPU) is instantiated outside the CV32E40P and is accessed via the APU interface (see *Auxiliary Processing Unit (APU)*). The FPU repository used by the CV32E40P 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/cv32e40p_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 **PULP\_ZFINX** of the toplevel file `cv32e40p_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>).

### 10.1 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.



## SLEEP UNIT

Source File: rtl/cv32e40p\_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 CV32E40P. 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 11.1 describes the Sleep Unit interface.

Table 11.1: Sleep Unit interface signals

Sig- nal	Di- rec- tion	Description
<code>pulp_clock_en_i</code>	put	<p><code>PULP_CLUSTER = 0</code>: <code>pulp_clock_en_i</code> is not used. Tie to 0.</p> <p><code>PULP_CLUSTER = 1</code>: <code>pulp_clock_en_i</code> can be used to gate <code>clk_i</code> internal to the core when <code>core_sleep_o = 1</code>. See <a href="#">PULP Cluster Extension</a> for details.</p>
<code>core_sleep_o</code>	put	<p><code>PULP_CLUSTER = 0</code>: Core is sleeping because of a <b>wfi</b> instruction. If <code>core_sleep_o = 1</code>, then <code>clk_i</code> is gated off internally and it is allowed to gate off <code>clk_i</code> externally as well. See <a href="#">WFI</a> for details.</p> <p><code>PULP_CLUSTER = 1</code>: Core is sleeping because of a <b>cv.elw</b> instruction. If <code>core_sleep_o = 1</code>, then the <code>pulp_clock_en_i</code> directly controls the internally instantiated clock gate and therefore <code>pulp_clock_en_i</code> can be set to 0 to internally gate off <code>clk_i</code>. If <code>core_sleep_o = 0</code>, then it is not allowed to set <code>pulp_clock_en_i</code> to 0. See <a href="#">PULP Cluster Extension</a> for details.</p>

---

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

---

## 11.1 Startup behavior

`clk_i` is internally gated off (while signaling `core_sleep_o = 0`) during CV32E40P 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.

## 11.2 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 11.1 shows an example waveform for sleep mode entry because of a **wfi** instruction.

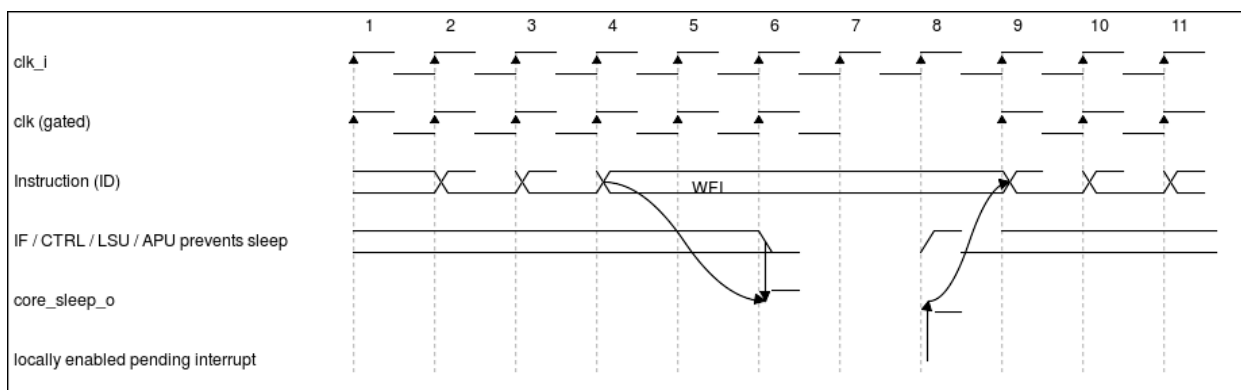


Figure 11.1: **wfi** example

## 11.3 PULP Cluster Extension

CV32E40P 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) CV32E40P cores that share a tightly-coupled data memory, aimed at running digital signal processing applications efficiently.

The mechanism via which CV32E40P 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 CV32E40P 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 CV32E40P 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 CV32E40P 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 withholding `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 CV32E40P 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 11.2 shows an example waveform for sleep mode entry because of a **cv.elw** instruction.

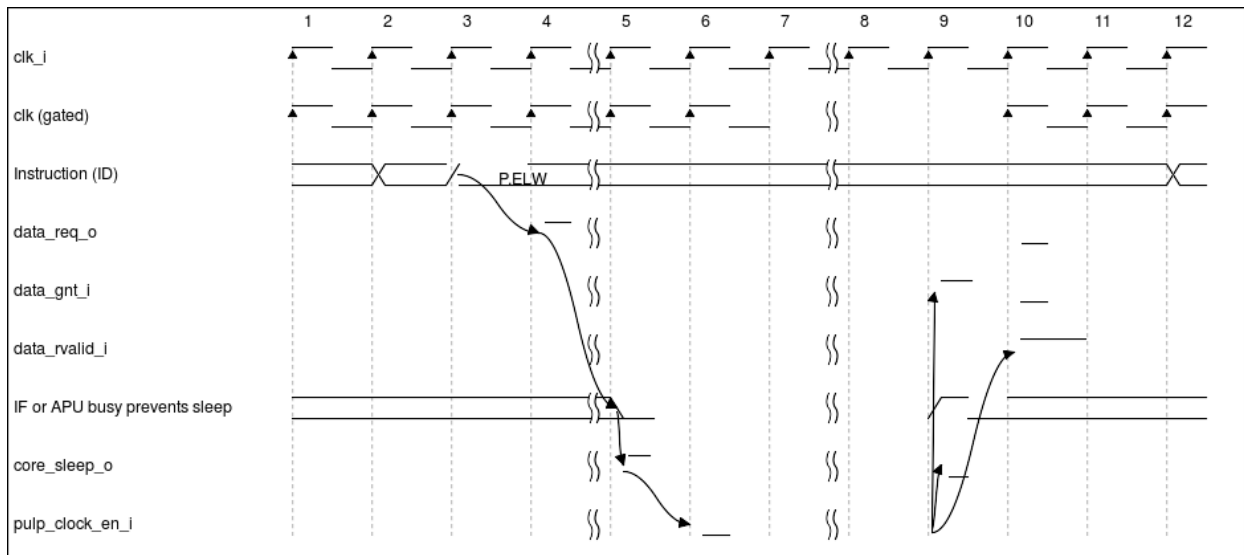


Figure 11.2: cv.elw example

## CORE-V HARDWARE LOOP EXTENSIONS

To increase the efficiency of small loops, CV32E40P 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. CV32E40P 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.

### 12.1 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 unconditional jump instructions allowed in the HWLoop body. An illegal exception is raised otherwise.
- No conditional 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.  $\text{HWLoop}[1].\text{endaddress} \geq \text{HWLoop}[0].\text{endaddress} + 8$  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.

```
1  asm volatile (  
2      ".option norvc;"  
3      "add %[i],x0, x0;"  
4      "add %[j],x0, x0;"  
5      "cv.count  x1, %[N];"  
6      "cv.endi   x1, end0;"  
7      "cv.starti x1, start0;"  
8          "start0:  cv.count  x0, %[N];"  
9          "cv.endi   x0, endZ;"  
10         "cv.starti x0, startZ;"  
11             "startZ: addi %[i], x0, 1;"  
12             "        addi %[i], x0, 1;"  
13             "endZ:   addi %[i], x0, 1;"  
14         "addi %[j],x0, 2;"  
15         "end0:   addi %[j], x0, 2;"  
16     : [i] "+r" (i), [j] "+r" (j)  
17     : [N] "r" (10)  
18 );
```

At the beginning of the HWLoop, the registers %[i] and %[j] are 0. The innermost loop, from startZ to endZ, 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.



## CONTROL AND STATUS REGISTERS

CV32E40P 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.

### 13.1 CSR Map

Table 13.1 lists all implemented CSRs. To columns in Table 13.1 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 13.1 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 CV32E40P 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 privilege 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 13.1: Control and Status Register Map

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

Table 13.1 – continued from previous page

CSR Address	Name	Privilege	Parameter	Description
0xC82	instreth	URO		(HPM) Upper 32 Instructions-Retired Count
0xC83	hpmcounterh3	URO		(HPM) Upper 32 Performance-Monitoring Counter 3
.....				
0xC9F	hpmcounterh31	URO		(HPM) Upper 32 Performance-Monitoring Counter 31
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 Register
0x323	mhpmevent3	MRW		(HPM) Machine Performance-Monitoring Counter 3
.....				
0x33F	mhpmevent31	MRW		(HPM) Machine Performance-Monitoring Counter 31
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 Count
0xB03	mhpmpcounter3	MRW		(HPM) Machine Performance-Monitoring Counter 3
.....				
0xB1F	mhpmpcounter31	MRW		(HPM) Machine Performance-Monitoring Counter 31
0xB80	mcycleh	MRW		(HPM) Upper 32 Machine Cycle Counter
0xB82	minstreth	MRW		(HPM) Upper 32 Machine Instructions-Retired Count
0xB83	mhpmpcounterh3	MRW		(HPM) Upper 32 Machine Performance-Monitoring Counter 3
.....				
0xB9F	mhpmpcounterh31	MRW		(HPM) Upper 32 Machine Performance-Monitoring Counter 31
0xF11	mvendorid	MRO		Machine Vendor ID

Table 13.1 – 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

## 13.2 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 13.1 (or a higher privilege level):

- **RO: read-only** fields are not affected 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.

### 13.2.1 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

### 13.2.2 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.

### 13.2.3 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 ( <i>frm</i> )
4:0	RW	Accrued Exceptions ( <i>fflags</i> )

### 13.2.4 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.

### 13.2.5 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.

### 13.2.6 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.

### 13.2.7 Privilege Level (privlv)

CSR Address: 0xCC1 (only present if PULP\_XPULP = 1)

Reset Value: 0x0000\_0003

Table 13.2: PRIVLV

Bit #	Mode	Description
31:2	RO	Reads as 0.
1:0	RO	Current Privilege Level. 11 = Machine, 10 = Hypervisor, 01 = Supervisor, 00 = User. CV32E40P only supports Machine mode.

### 13.2.8 User Hardware Thread ID (uhartid)

CSR Address: 0xCC0 (only present if PULP\_XPULP = 1)

Reset Value: Defined

Table 13.3: UHARTID

Bit #	Mode	Description
31:0	RO	Hardware Thread ID <b>hart_id_i</b> , see <i>Core Integration</i>

Similar to **mhartid** the **uhartid** provides the Hardware Thread ID. It differs from **mhartid** only in the required privilege level. On CV32E40P, as it is a machine mode only implementation, this difference is not noticeable.

### 13.2.9 Machine Status (mstatus)

CSR Address: 0x300

Reset Value: 0x0000\_1800

Bit #	Mode	Description
31:18	RO	Reserved, hardwired to 0
17	RO	<b>MPRV</b> : hardwired to 0
16:13	RO	Unimplemented, hardwired to 0
12:11	RO	<b>MPP</b> : Machine Previous Privilege mode, hardwired to 11 when the user mode is not enabled.
10:8	RO	Unimplemented, hardwired to 0
7	RO	<b>Previous Machine Interrupt Enable</b> : 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	<b>Previous User Interrupt Enable</b> : 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	<b>Machine Interrupt Enable</b> : 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	<b>User Interrupt Enable</b> : If you want to enable user level interrupt handling in your exception handler, set the Interrupt Enable UIE to 1 inside your handler code.

### 13.2.10 Machine ISA (misa)

CSR Address: 0x301

Reset Value: defined

Detailed:

Bit #	Mode	Description
31:30	RO (0x1)	<b>MXL</b> (Machine XLEN).
29:26	RO (0x0)	(Reserved).
25	RO (0x0)	<b>Z</b> (Reserved). Read-only; writes are ignored.
24	RO (0x0)	<b>Y</b> (Reserved).
23	RO	<b>X</b> (Non-standard extensions present).
22	RO (0x0)	<b>W</b> (Reserved).
21	RO (0x0)	<b>V</b> (Tentatively reserved for Vector extension).
20	RO (0x0)	<b>U</b> (User mode implemented).
19	RO (0x0)	<b>T</b> (Tentatively reserved for Transactional Memory extension).
18	RO (0x0)	<b>S</b> (Supervisor mode implemented).
17	RO (0x0)	<b>R</b> (Reserved).
16	RO (0x0)	<b>Q</b> (Quad-precision floating-point extension).
15	RO (0x0)	<b>P</b> (Tentatively reserved for Packed-SIMD extension).
14	RO (0x0)	<b>O</b> (Reserved).
13	RO (0x0)	<b>N</b> (User-level interrupts supported).
12	RO (0x1)	<b>M</b> (Integer Multiply/Divide extension).
11	RO (0x0)	<b>L</b> (Tentatively reserved for Decimal Floating-Point extension).
10	RO (0x0)	<b>K</b> (Reserved).
9	RO (0x0)	<b>J</b> (Tentatively reserved for Dynamically Translated Languages extension).
8	RO (0x1)	<b>I</b> (RV32I/64I/128I base ISA).
7	RO (0x0)	<b>H</b> (Hypervisor extension).
6	RO (0x0)	<b>G</b> (Additional standard extensions present).
5	RO	<b>F</b> (Single-precision floating-point extension).
4	RO (0x0)	<b>E</b> (RV32E base ISA).
3	RO (0x0)	<b>D</b> (Double-precision floating-point extension).
2	RO (0x1)	<b>C</b> (Compressed extension).
1	RO (0x0)	<b>B</b> (Tentatively reserved for Bit-Manipulation extension).
0	RO (0x0)	<b>A</b> (Atomic extension).

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

- **C** = 1
- **F** = 1 if FPU = 1 and PULP\_ZFINX = 0
- **I** = 1
- **M** = 1
- **X** = 1 if PULP\_XPULP = 1 or PULP\_CLUSTER = 1
- **MXL** = 1 (i.e. XLEN = 32)

### 13.2.11 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	<b>Machine External Interrupt Enable (MEIE)</b> : If set, irq_i[11] is enabled.
7	RW	<b>Machine Timer Interrupt Enable (MTIE)</b> : If set, irq_i[7] is enabled.
3	RW	<b>Machine Software Interrupt Enable (MSIE)</b> : if set, irq_i[3] is enabled.

### 13.2.12 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.

### 13.2.13 Machine Counter-Inhibit Register (mcountinhibit)

CSR Address: 0x320

Reset Value: 0x0000\_000D

The performance counter inhibit control register. The default value is to inhibit 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	<b>selectors</b> : mhpcounter3 inhibit
2	RW	minstret inhibit
1	RO	0
0	RW	mcycle inhibit

### 13.2.14 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	<b>selectors:</b> 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.

### 13.2.15 Machine Scratch (`mscratch`)

CSR Address: 0x340

Reset Value: 0x0000\_0000

Detailed:

Bit #	Mode	Description
31:0	RW	Scratch value

### 13.2.16 Machine Exception PC (`mepc`)

CSR Address: 0x341

Reset Value: 0x0000\_0000

Bit #	Mode	Description
31:1	RW	Machine Expection Program Counter 31:1
0	RO	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.

### 13.2.17 Machine Cause (`mcause`)

CSR Address: 0x342

Reset Value: 0x0000\_0000

Bit #	Mode	Description
31	RW	<b>Interrupt:</b> This bit is set when the exception was triggered by an interrupt.
30:5	RO (0)	Always 0
4:0	RW	<b>Exception Code</b> (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.

### 13.2.18 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.

### 13.2.19 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 <i>irq_i[x]</i> is pending.
11	RO	<b>Machine External Interrupt Pending (MEIP):</b> If set, <i>irq_i[11]</i> is pending.
7	RO	<b>Machine Timer Interrupt Pending (MTIP):</b> If set, <i>irq_i[7]</i> is pending.
3	RO	<b>Machine Software Interrupt Pending (MSIP):</b> if set, <i>irq_i[3]</i> is pending.

### 13.2.20 Trigger Select Register (*tselect*)

CSR Address: 0x7A0

Reset Value: 0x0000\_0000

Accessible in Debug Mode or M-Mode.

Bit #	Mode	Description
31:0	RO	CV32E40P implements a single trigger, therefore this register will always read as zero

### 13.2.21 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:** CV32E40P 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)	<b>type:</b> 2 = Address/Data match trigger type.
27	RO (0x1)	<b>dmode:</b> 1 = Only debug mode can write tdata registers
26:21	RO (0x0)	<b>maskmax:</b> 0 = Only exact matching supported.
20	RO (0x0)	<b>hit:</b> 0 = Hit indication not supported.
19	RO (0x0)	<b>select:</b> 0 = Only address matching is supported.
18	RO (0x0)	<b>timing:</b> 0 = Break before the instruction at the specified address.
17:16	RO (0x0)	<b>szelo:</b> 0 = Match accesses of any size.
15:12	RO (0x1)	<b>action:</b> 1 = Enter debug mode on match.
11	RO (0x0)	<b>chain:</b> 0 = Chaining not supported.
10:7	RO (0x0)	<b>match:</b> 0 = Match the whole address.
6	RO (0x1)	<b>m:</b> 1 = Match in M-Mode.
5	RO (0x0)	zero.
4	RO (0x0)	<b>s:</b> 0 = S-Mode not supported.
3	RO (0x0)	<b>u:</b> 0 = U-Mode not supported.
2	RW	<b>execute:</b> Enable matching on instruction address.
1	RO (0x0)	<b>store:</b> 0 = Store address / data matching not supported.
0	RO (0x0)	<b>load:</b> 0 = Load address / data matching not supported.

### 13.2.22 Trigger Data Register 2 (tdata2)

CSR Address: 0x7A2

Reset Value: 0x0000\_0000

Detailed:

Bit#	Mode	Description
31:0	RW	<b>data</b>

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.

### 13.2.23 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. CV32E40P does not support the features requiring this register. Writes are ignored and reads will always return zero.

### 13.2.24 Trigger Info (tinfo)

CSR Address: 0x7A4

Reset Value: 0x0000\_0004

Detailed:

Bit#	Mode	Description
31:16	RO (0x0)	0
15:0	RO (0x4)	<b>info</b> . 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.

### 13.2.25 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. CV32E40P does not support the features requiring this register. Writes are ignored and reads will always return zero.

### 13.2.26 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. CV32E40P does not support the features requiring this register. Writes are ignored and reads will always return zero.

### 13.2.27 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 CV32E40P 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)	<b>xdebugver:</b> returns 4 - External debug support exists as it is described in this document.
27:16	RO (0x0)	Reserved
15	RW	<b>ebreakm</b>
14	RO (0x0)	Reserved
13	RO (0x0)	<b>ebreaks.</b> Always 0.
12	RO (0x0)	<b>ebreaku.</b> Always 0.
11	RW	<b>stepie</b>
10	RO (0x0)	<b>stopcount.</b> Always 0.
9	RO (0x0)	<b>stoptime.</b> Always 0.
8:6	RO	<b>cause</b>
5	RO (0x0)	Reserved
4	RO (0x0)	<b>mprven.</b> Always 0.
3	RO (0x0)	<b>nmip.</b> Always 0.
2	RW	<b>step</b>
1:0	RO (0x3)	<b>prv:</b> returns the current privilege mode

### 13.2.28 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.

### 13.2.29 Debug Scratch Register 0/1 (dscratch0/1)

CSR Address: 0x7B2/0x7B3

Reset Value: 0x0000\_0000

Detailed:

Bit #	Mode	Description
31:0	RW	DSCRATCH0/1

### 13.2.30 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.

### 13.2.31 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.

### 13.2.32 Machine Performance Monitoring Counter (mhpmpcounter3 .. mhpmpcounter31)

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.

### 13.2.33 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.

### 13.2.34 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.

### 13.2.35 Upper 32 Machine Performance Monitoring Counter (`mhpmpcounter3h .. mhpmpcounter31h`)

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.

### 13.2.36 Machine Vendor ID (`mvvendorid`)

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

### 13.2.37 Machine Architecture ID (`marchid`)

CSR Address: 0xF12

Reset Value: 0x0000\_0004

Detailed:

Bit #	Mode	Description
31:0	RO	Machine Architecture ID of CV32E40P is 4

### 13.2.38 Machine Implementation ID (`mimpid`)

CSR Address: 0xF13

Reset Value: 0x0000\_0000

Detailed:

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

### 13.2.39 Hardware Thread ID (`mhartid`)

CSR Address: 0xF14

Reset Value: Defined

Bit #	Mode	Description
31:0	RO	Hardware Thread ID <code>hart_id_i</code> , see <i>Core Integration</i>

**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.

## 13.3 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.

## 13.4 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.

## 13.5 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.

## 13.6 Upper 32 Cycle Counter (cyc1eh)

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.



## 13.7 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.

## 13.8 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.



## PERFORMANCE COUNTERS

CV32E40P 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.

CV32E40P implements the clock cycle counter `mcycle(h)`, the retired instruction counter `minstret(h)`, as well as the parameterizable number of event counters `mhpmpcounter3(h)` - `mhpmpcounter31(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.

---

### 14.1 Event Selector

The following events can be monitored using the performance counters of CV32E40P.

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 fetches, 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	Number 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 APU/FP

The event selector CSRs `mhpmevent3` - `mhpmevent31` define which of these events are counted by the event counters `mhpmcouter3(h)` - `mhpmcouter31(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.

## 14.2 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 `mhpmcouterX(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.

## 14.3 Parametrization at synthesis time

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

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

An increment of 1 to the `NUM_MHPMCOUNTERS` results in the addition of the following:

- 64 flops for `mhpmcouterX`
- 15 flops for `mhpmeventX`

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

## 14.4 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).



## EXCEPTIONS AND INTERRUPTS

CV32E40P 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.

### 15.1 Interrupt Interface

Table 15.1 describes the interrupt interface.

Table 15.1: Interrupt interface signals

Sig- nal	Di- rec- tion	Description
<code>irq_i[31:0]</code> input		Active high, level sensitive interrupt inputs. Not all interrupt inputs can be used on CV32E40P. Specifically <code>irq_i[15:12]</code> , <code>irq_i[10:8]</code> , <code>irq_i[6:4]</code> and <code>irq_i[2:0]</code> 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. <code>irq_i[11]</code> , <code>irq_i[7]</code> , and <code>irq_i[3]</code> correspond to the Machine External Interrupt (MEI), Machine Timer Interrupt (MTI), and Machine Software Interrupt (MSI) respectively. The <code>irq_i[31:16]</code> interrupts are a CV32E40P specific extension to the RISC-V Basic (a.k.a. CLINT) interrupt scheme.
<code>irq_ack_o</code> output		Interrupt acknowledge. Set to 1 for one cycle when the interrupt with ID <code>irq_id_o[4:0]</code> is taken.
<code>irq_id_o[4:0]</code> output		Interrupt index for taken interrupt. Only valid when <code>irq_ack_o = 1</code> .

## 15.2 Interrupts

The `irq_i[31:0]` interrupts are controlled via the `mstatus`, `mie` and `mip` CSRs. CV32E40P 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 source, 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.

## 15.3 Exceptions

CV32E40P 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 CV32E40P 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 CV32E40P 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).

## 15.4 Nested Interrupt/Exception Handling

CV32E40P 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.



```
1  isr_handle_nested_interrupts(id) {  
2      // Save mpec and mstatus to stack  
3      mepc_bak = mepc;  
4      mstatus_bak = mstatus;  
5  
6      // Save mie to stack (optional)  
7      mie_bak = mie;  
8  
9      // Keep lower-priority interrupts disabled (optional)  
10     mie = mie & ~((1 << (id + 1)) - 1);  
11  
12     // Re-enable interrupts  
13     mstatus.MIE = 1;  
14  
15     // Handle interrupt  
16     // This code block can be interrupted by other interrupts.  
17     // ...  
18  
19     // Restore mstatus (this disables interrupts) and mepc  
20     mstatus = mstatus_bak;  
21     mepc = mepc_bak;  
22  
23     // Restore mie (optional)  
24     mie = mie_bak;  
25 }
```

Nesting of interrupts/exceptions in hardware is not supported.



## DEBUG & TRIGGER

CV32E40P 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 CV32E40P 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 CV32E40P 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.

## 16.1 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 CV32E40P 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 CV32E40P is running normally.

debug\_halted\_o is used to signal that the CV32E40P 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.

## 16.2 Core Debug Registers

CV32E40P 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.

## 16.3 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 CV32E40P will not be classified as nonexistent by the integrator.

The CV32E40P 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 CV32E40P 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 16.1 and show Figure 16.2 show typical examples of transitioning into the running and halted states.

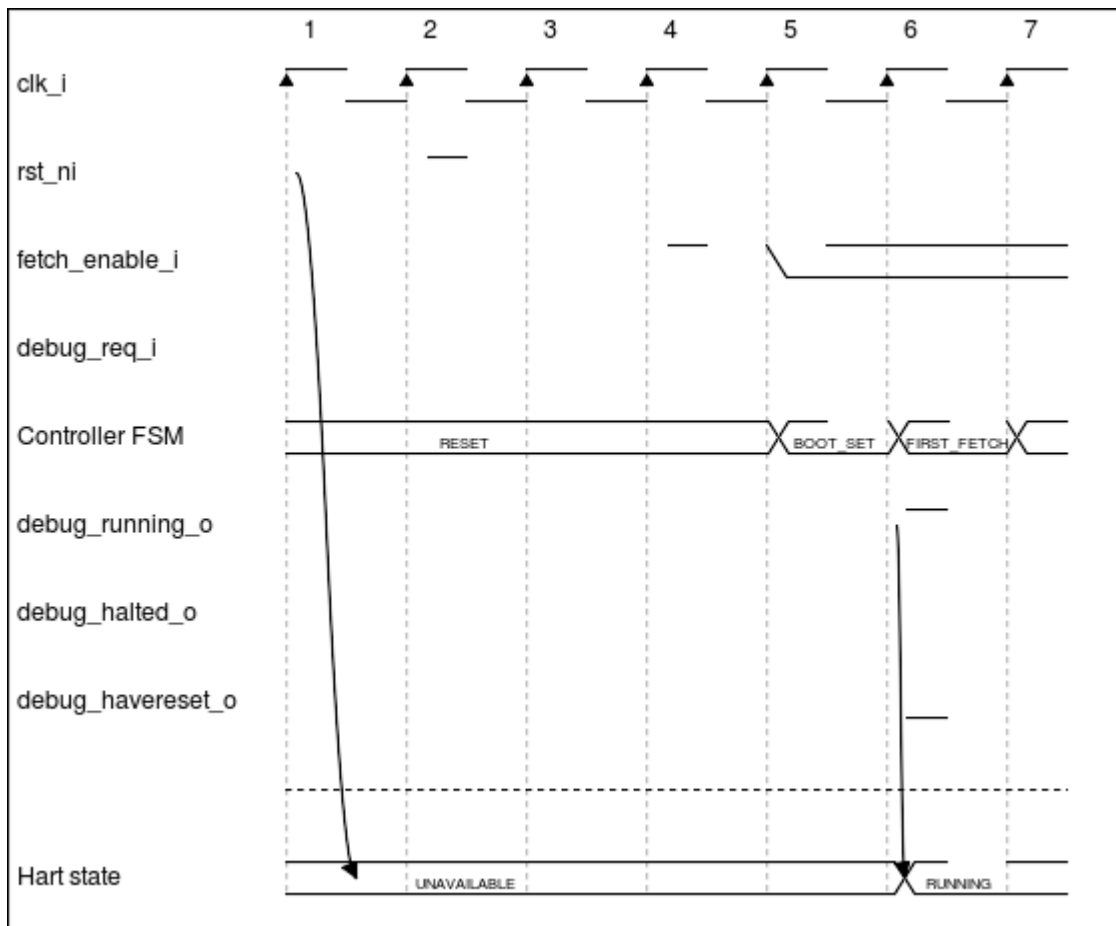


Figure 16.1: Transition into debug running state

The key properties of the debug states are:

- The CV32E40P 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 CV32E40P is guaranteed to transition straight from its unavailable state into its

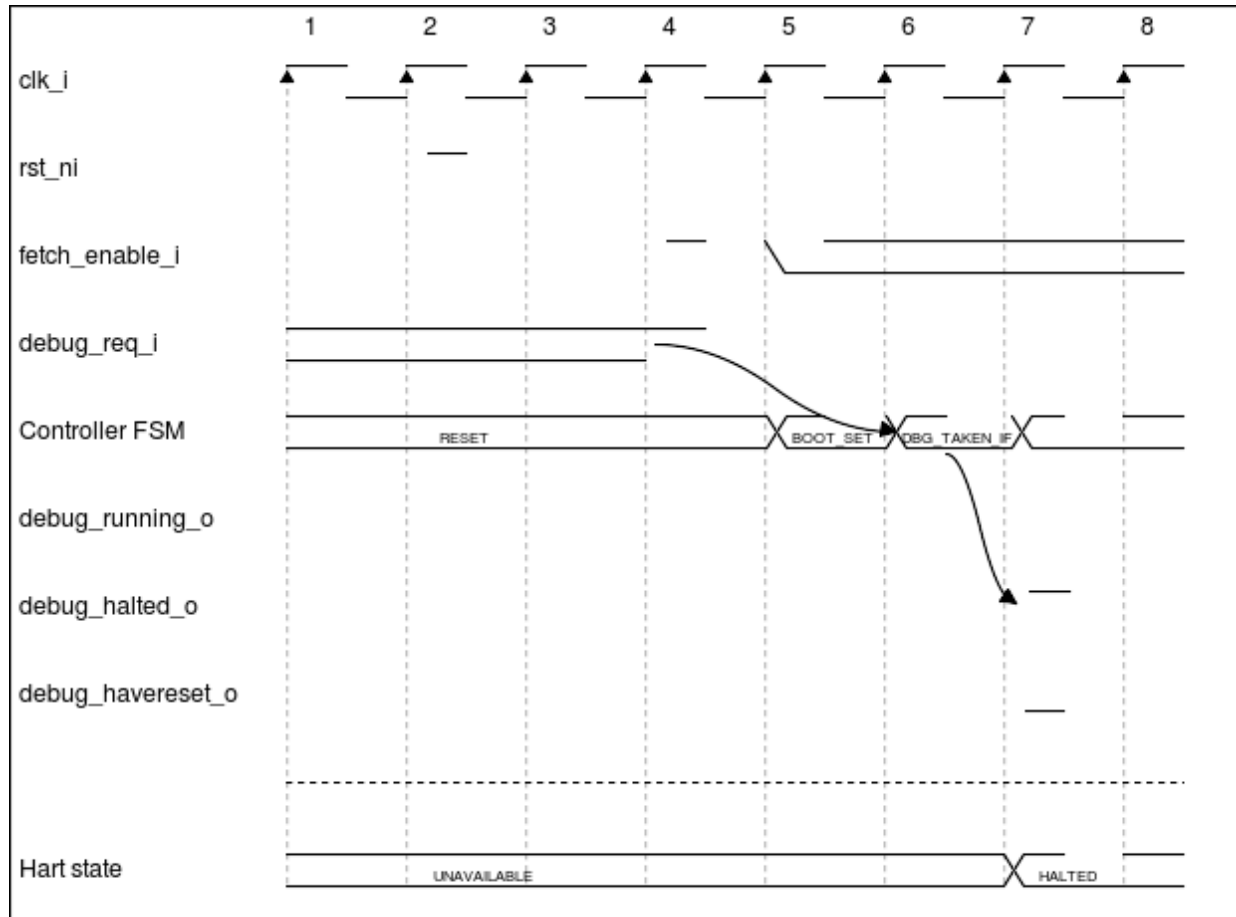


Figure 16.2: Transition into debug halted state

halted state. If `debug_req_i` is asserted at a later point in time, then the CV32E40P might transition through the `running` state on its way 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).

## 16.4 EBREAK Behavior

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

### 16.4.1 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 CV32E40P 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.*

### 16.4.2 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`.*

### 16.4.3 Scenario 3 : Exit Program Buffer & Restart Debug Code

Executing 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

## 16.5 Interrupts during Single-Step Behavior

The CV32E40P CPU is not compliant with the intended interpretation of the RISC-V Debug spec 0.13.2 specification when interrupts occur during Single-Steps. However, the intended behavior has been clarified a posteriori only in version 1.0.0. See <https://github.com/riscv/riscv-debug-spec/issues/510>. The CV32E40P executes the first instruction of the interrupt handler and retires it before re-entering in Debug Mode, which is prohibited in version 1.0.0 but not specified in 0.13.2. For details about the specific use-case, please refer to <https://github.com/openhwgroup/core-v-verif/issues/904>.



## TRACER

The module `cv32e40p_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 `cv32e40p_core`. It can be enabled during simulation by defining `CV32E40P_TRACE_EXECUTION`.

### 17.1 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.

### 17.2 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				

(continues on next page)

(continued from previous page)

142	67	0000015c	c622	c.swsp	x8,12(x2)	x2:0x00002000	↵
↵x8:0x00000000 PA:0x0000200c store:0x00000000 load:0xffffffff							

## CORE-V INSTRUCTION SET EXTENSIONS

CV32E40P 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*.
- Single Instruction Multiple Data (aka SIMD) extensions, 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.

### 18.1 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`.

### 18.1.1 Load Operations

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

## 18.1.2 Store Operations

Mnemonic	Description
<b>Register-Immediate Stores with Post-Increment</b>	
<b>cv.sb rs2, Imm(rs1!)</b>	Mem8(rs1) = rs2 rs1 += Sext(Imm[11:0])
<b>cv.sh rs2, Imm(rs1!)</b>	Mem16(rs1) = rs2 rs1 += Sext(Imm[11:0])
<b>cv.sw rs2, Imm(rs1!)</b>	Mem32(rs1) = rs2 rs1 += Sext(Imm[11:0])
<b>Register-Register Stores with Post-Increment</b>	
<b>cv.sb rs2, rs3(rs1!)</b>	Mem8(rs1) = rs2 rs1 += rs3
<b>cv.sh rs2, rs3(rs1!)</b>	Mem16(rs1) = rs2 rs1 += rs3
<b>cv.sw rs2, rs3(rs1!)</b>	Mem32(rs1) = rs2 rs1 += rs3
<b>Register-Register Stores</b>	
<b>cv.sb rs2, rs3(rs1)</b>	Mem8(rs1 + rs3) = rs2
<b>cv.sh rs2 rs3(rs1)</b>	Mem16(rs1 + rs3) = rs2
<b>cv.sw rs2, rs3(rs1)</b>	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	<b>cv.lb rD, Imm(rs1!)</b>
offset	base	100	dest	000 1011	<b>cv.lbu rD, Imm(rs1!)</b>
offset	base	001	dest	000 1011	<b>cv.lh rD, Imm(rs1!)</b>
offset	base	101	dest	000 1011	<b>cv.lhu rD, Imm(rs1!)</b>
offset	base	010	dest	000 1011	<b>cv.lw rD, Imm(rs1!)</b>

31 : 25	24 : 20	19 :15	14 : 12	11 :07	06 : 00	
funct7	rs2	rs1	funct3	rd	opcode	Mnemonic
000 0000	offset	base	011	dest	010 1011	<b>cv.lb rD, rs2(rs1!)</b>
000 1000	offset	base	011	dest	010 1011	<b>cv.lbu rD, rs2(rs1!)</b>
000 0001	offset	base	011	dest	010 1011	<b>cv.lh rD, rs2(rs1!)</b>
000 1001	offset	base	011	dest	010 1011	<b>cv.lhu rD, rs2(rs1!)</b>
000 0010	offset	base	011	dest	010 1011	<b>cv.lw rD, rs2(rs1!)</b>

31 : 25	24 : 20	19 :15	14 : 12	11 :07	06 : 00	
funct7	rs2	rs1	funct3	rd	opcode	Mnemonic
000 0100	offset	base	011	dest	010 1011	<b>cv.lb rD, rs2(rs1)</b>
000 1100	offset	base	011	dest	010 1011	<b>cv.lbu rD, rs2(rs1)</b>
000 0101	offset	base	011	dest	010 1011	<b>cv.lh rD, rs2(rs1)</b>
000 1101	offset	base	011	dest	010 1011	<b>cv.lhu rD, rs2(rs1)</b>
000 0110	offset	base	011	dest	010 1011	<b>cv.lw rD, rs2(rs1)</b>

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	<b>cv.sb rs2, Imm(rs1!)</b>
offset[11:5]	src	base	001	offset[4:0]	010 1011	<b>cv.sh rs2, Imm(rs1!)</b>
offset[11:5]	src	base	010	offset[4:0]	010 1011	<b>cv.sw rs2, Imm(rs1!)</b>

31 : 25	24 : 20	19 :15	14 : 12	11 :07	06 : 00	
funct7	rs2	rs1	funct3	rd	opcode	Mnemonic
001 0000	src	base	011	offset	010 1011	<b>cv.sb rs2, rs3(rs1!)</b>
001 0001	src	base	011	offset	010 1011	<b>cv.sh rs2, rs3(rs1!)</b>
001 0010	src	base	011	offset	010 1011	<b>cv.sw rs2, rs3(rs1!)</b>

31 : 25	24 : 20	19 :15	14 : 12	11 :07	06 : 00	
funct7	rs2	rs1	funct3	rs3	opcode	Mnemonic
001 0100	src	base	011	offset	010 1011	<b>cv.sb rs2, rs3(rs1)</b>
001 0101	src	base	011	offset	010 1011	<b>cv.sh rs2, rs3(rs1)</b>
001 0110	src	base	011	offset	010 1011	<b>cv.sw rs2, rs3(rs1)</b>

## 18.2 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 CV32E40P to enter a sleep state as explained in [PULP Cluster Extension](#).

### 18.2.1 Load Operations

Mnemonic	Description
<b>Event Load</b>	
<b>cv.elw rD, Imm(rs1)</b>	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	011	dest	000 1011	<b>cv.elw rD, Imm(rs1)</b>

## 18.3 Hardware Loops

CV32E40P 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.

### 18.3.1 Operations

#### Long Hardware Loop Setup instructions

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

#### Short Hardware Loop Setup Instructions

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

#### 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	100	0000	L	010 1011	<b>cv.starti L, uimmL</b>
uimmL[11:0]	00000	100	0010	L	010 1011	<b>cv.endi L, uimmL</b>
uimmL[11:0]	00000	100	0100	L	010 1011	<b>cv.counti L, uimmL</b>
0000 0000 0000	src1	100	0101	L	010 1011	<b>cv.count L, rs1</b>
uimmL[11:0]	uimmS[4:0]	100	0110	L	010 1011	<b>cv.setupi L, uimmL, uimmS</b>
uimmL[11:0]	src1	100	0111	L	010 1011	<b>cv.setup L, rs1, uimmL</b>

## 18.4 ALU

CV32E40P 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 *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  $rs2[4:0]$

### 18.4.1 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 0 1 0 1 0 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 00100001011010110010010001001011

```

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.

## 18.4.2 Bit Manipulation Operations

Mnemonic	Description
<b>cv.extractrD, rs1, Is3, Is2</b>	$rD = \text{Sext}(rs1[\min(Is3+Is2,31):Is2])$
<b>cv.extractuD, rs1, Is3, Is2</b>	$rD = \text{Zext}(rs1[\min(Is3+Is2,31):Is2])$
<b>cv.extractrD, rs1, rs2</b>	$rD = \text{Sext}(rs1[\min(rs2[9:5]+rs2[4:0],31):rs2[4:0]])$
<b>cv.extractuD, rs1, rs2</b>	$rD = \text{Zext}(rs1[\min(rs2[9:5]+rs2[4:0],31):rs2[4:0]])$
<b>cv.insert rD, rs1, Is3, Is2</b>	$rD[\min(Is3+Is2,31):Is2] = rs1[Is3:\max(Is3+Is2,31)-31]$ The rest of the bits of rD are untouched and keep their previous value.
<b>cv.insert rD, rs1, rs2</b>	$rD[\min(rs2[9:5]+rs2[4:0],31):rs2[4:0]] = rs1[rs2[9:5]:\max(rs2[9:5]+rs2[4:0],31)-31]$ The rest of the bits of rD are untouched and keep their previous value.
<b>cv.bclr rD, rs1, Is3, Is2</b>	$rD[\min(Is3+Is2,31):Is2]$ bits set to 0 The rest of the bits of rD are passed through from rs1 and are not modified.
<b>cv.bclrr rD, rs1, rs2</b>	$rD[\min(rs2[9:5]+rs2[4:0],31):rs2[4:0]]$ bits set to 0 The rest of the bits of rD are passed through from rs1 and are not modified.
<b>cv.bset rD, rs1, Is3, Is2</b>	$rD[\min(Is3+Is2,31):Is2]$ bits set to 1 The rest of the bits of rD are passed through from rs1 and are not modified.
<b>cv.bsetr rD, rs1, rs2</b>	$rD[\min(rs2[9:5]+rs2[4:0],31):rs2[4:0]]$ bits set to 1 The rest of the bits of rD are passed through from rs1 and are not modified.
<b>cv.ffl rD, rs1</b>	$rD$ = 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.
<b>cv.flf rD, rs1</b>	$rD$ = 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.
<b>cv.clb rD, rs1</b>	$rD$ = count leading bits of rs1 Number of consecutive 1's or 0's starting from MSB. If rs1 is 0, rD will be 0.
<b>cv.cnt rD, rs1</b>	$rD$ = Population count of rs1 Number of bits set in rs1.
<b>cv.ror rD, rs1, rs2</b>	$rD = \text{RotateRight}(rs1, rs2)$
<b>cv.bitrev rD, rs1, Is3, Is2</b>	Given an input rs1 it returns a bit reversed representation assuming FFT on $2^{Is2}$ points in Radix $2^{(Is3+1)}$ . $Is3$ 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.

### 18.4.3 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
00	Luimm5[4:0]	Iuimm5[4:0]	src	000	dest	101 1011	<b>cv.extract rD, rs1, ls3, ls2</b>
01	Luimm5[4:0]	Iuimm5[4:0]	src	000	dest	101 1011	<b>cv.extractu rD, rs1, ls3, ls2</b>
10	Luimm5[4:0]	Iuimm5[4:0]	src	000	dest	101 1011	<b>cv.insert rD, rs1, ls3, ls2</b>
00	Luimm5[4:0]	Iuimm5[4:0]	src	001	dest	101 1011	<b>cv.bclr rD, rs1, ls3, ls2</b>
01	Luimm5[4:0]	Iuimm5[4:0]	src	001	dest	101 1011	<b>cv.bset rD, rs1, ls3, ls2</b>
11	{3'b000, Luimm2[1:0]}	Iuimm5[4:0]	src	001	dest	101 1011	<b>cv.bitrev rD, rs1, ls3, ls2</b>

31 : 25	24 : 20	19 :15	14 : 12	11 : 7	6 : 0	
funct7	rs2	rs1	funct3	rD	opcode	
001 1000	src2	src1	011	dest	010 1011	<b>cv.extractr rD, rs1, rs2</b>
001 1001	src2	src1	011	dest	010 1011	<b>cv.extractur rD, rs1, rs2</b>
001 1010	src2	src1	011	dest	010 1011	<b>cv.insertr rD, rs1, rs2</b>
001 1100	src2	src1	011	dest	010 1011	<b>cv.bclrr rD, rs1, rs2</b>
001 1101	src2	src1	011	dest	010 1011	<b>cv.bsetr rD, rs1, rs2</b>
010 0000	src2	src1	011	dest	010 1011	<b>cv.ror rD, rs1, rs2</b>
010 0001	00000	src1	011	dest	010 1011	<b>cv.ff1 rD, rs1</b>
010 0010	00000	src1	011	dest	010 1011	<b>cv.fl1 rD, rs1</b>
010 0011	00000	src1	011	dest	010 1011	<b>cv.clb rD, rs1</b>
010 0100	00000	src1	011	dest	010 1011	<b>cv.cnt rD, rs1</b>

### 18.4.4 General ALU Operations

Mnemonic		Description
<b>cv.abs</b>	<b>rD, rs1</b>	$rD = rs1 < 0 ? -rs1 : rs1$
<b>cv.slet</b>	<b>rD, rs1, rs2</b>	$rD = rs1 \leq rs2 ? 1 : 0$ Note: Comparison is signed.
<b>cv.sletu</b>	<b>rD, rs1, rs2</b>	$rD = rs1 \leq rs2 ? 1 : 0$ Note: Comparison is unsigned.
<b>cv.min</b>	<b>rD, rs1, rs2</b>	$rD = rs1 < rs2 ? rs1 : rs2$ Note: Comparison is signed.
<b>cv.minu</b>	<b>rD, rs1, rs2</b>	$rD = rs1 < rs2 ? rs1 : rs2$ Note: Comparison is unsigned.
<b>cv.max</b>	<b>rD, rs1, rs2</b>	$rD = rs1 < rs2 ? rs2 : rs1$ Note: Comparison is signed.
<b>cv.maxu</b>	<b>rD, rs1, rs2</b>	$rD = rs1 < rs2 ? rs2 : rs1$ Note: Comparison is unsigned.
<b>cv.exths</b>	<b>rD, rs1</b>	$rD = \text{Sext}(rs1[15:0])$

continues on next page

Table 18.1 – continued from previous page

Mnemonic		Description
cv.exthz	rD, rs1	$rD = \text{Zext}(rs1[15:0])$
cv.extbs	rD, rs1	$rD = \text{Sext}(rs1[7:0])$
cv.extbz	rD, rs1	$rD = \text{Zext}(rs1[7:0])$
cv.clip	rD, rs1, Is2	if $rs1 \leq -2^{(Is2-1)}$ , $rD = -2^{(Is2-1)}$ , else if $rs1 \geq 2^{(Is2-1)-1}$ , $rD = 2^{(Is2-1)-1}$ , else $rD = rs1$ Note: If Is2 is equal to 0, $-2^{(Is2-1)}$ is equivalent to -1 while $(2^{(Is2-1)-1})$ is equivalent to 0.
cv.clipu	rD, rs1, Is2	if $rs1 \leq 0$ , $rD = 0$ , else if $rs1 \geq 2^{(Is2-1)-1}$ , $rD = 2^{(Is2-1)-1}$ , else $rD = rs1$ Note: If Is2 is equal to 0, $(2^{(Is2-1)-1})$ is equivalent to 0.
cv.clipr	rD, rs1, rs2	if $rs1 \leq -(rs2+1)$ , $rD = -(rs2+1)$ , else if $rs1 \geq rs2$ , $rD = rs2$ , else $rD = rs1$
cv.clipur	rD, rs1, rs2	if $rs1 \leq 0$ , $rD = 0$ , else if $rs1 \geq rs2$ , $rD = rs2$ , else $rD = rs1$
cv.addN	rD, rs1, rs2, Is3	$rD = (rs1 + rs2) \ggg Is3$ Note: Arithmetic shift right. Setting Is3 to 2 replaces former p.avg.
cv.adduN	rD, rs1, rs2, Is3	$rD = (rs1 + rs2) \gg Is3$ Note: Logical shift right. Setting Is3 to 2 replaces former p.avg.
cv.addRN	rD, rs1, rs2, Is3	$rD = (rs1 + rs2 + 2^{(Is3-1)}) \ggg Is3$ Note: Arithmetic shift right.
cv.adduRN	rD, rs1, rs2, Is3	$rD = (rs1 + rs2 + 2^{(Is3-1)}) \gg Is3$ Note: Logical shift right.
cv.subN	rD, rs1, rs2, Is3	$rD = (rs1 - rs2) \ggg Is3$ Note: Arithmetic shift right.
cv.subuN	rD, rs1, rs2, Is3	$rD = (rs1 - rs2) \gg Is3$ Note: Logical shift right.
cv.subRN	rD, rs1, rs2, Is3	$rD = (rs1 - rs2 + 2^{(Is3-1)}) \ggg Is3$ Note: Arithmetic shift right.
cv.subuRN	rD, rs1, rs2, Is3	$rD = (rs1 - rs2 + 2^{(Is3-1)}) \gg Is3$ Note: Logical shift right.
cv.addNr	rD, rs1, rs2	$rD = (rD + rs1) \ggg rs2[4:0]$ Note: Arithmetic shift right.
cv.adduNr	rD, rs1, rs2	$rD = (rD + rs1) \gg rs2[4:0]$ Note: Logical shift right.

continues on next page

Table 18.1 – continued from previous page

Mnemonic		Description
<b>cv.addRNR</b>	<b>rD, rs1, rs2</b>	$rD = (rD + rs1 + 2^{(rs2[4:0]-1)}) >>> rs2[4:0]$ Note: Arithmetic shift right.
<b>cv.adduRNR</b>	<b>rD, rs1, rs2</b>	$rD = (rD + rs1 + 2^{(rs2[4:0]-1)}) >> rs2[4:0]$ Note: Logical shift right.
<b>cv.subNR</b>	<b>rD, rs1, rs2</b>	$rD = (rD - rs1) >>> rs2[4:0]$ Note: Arithmetic shift right.
<b>cv.subuNR</b>	<b>rD, rs1, rs2</b>	$rD = (rD - rs1) >> rs2[4:0]$ Note: Logical shift right.
<b>cv.subRNR</b>	<b>rD, rs1, rs2</b>	$rD = (rD - rs1 + 2^{(rs2[4:0]-1)}) >>> rs2[4:0]$ Note: Arithmetic shift right.
<b>cv.subuRNR</b>	<b>rD, rs1, rs2</b>	$rD = (rD - rs1 + 2^{(rs2[4:0]-1)}) >> rs2[4:0]$ Note: Logical shift right.

### 18.4.5 General ALU Encoding

31 : 25	24 : 20	19 : 15	14 : 12	11 : 7	6 : 0	
funct7	rs2	rs1	funct	rD	opcode	
010 1000	00000	src1	011	dest	010 1011	<b>cv.abs rD, rs1</b>
010 1001	src2	src1	011	dest	010 1011	<b>cv.slet rD, rs1, rs2</b>
010 1010	src2	src1	011	dest	010 1011	<b>cv.sletu rD, rs1, rs2</b>
010 1011	src2	src1	011	dest	010 1011	<b>cv.min rD, rs1, rs2</b>
010 1100	src2	src1	011	dest	010 1011	<b>cv.minu rD, rs1, rs2</b>
010 1101	src2	src1	011	dest	010 1011	<b>cv.max rD, rs1, rs2</b>
010 1110	src2	src1	011	dest	010 1011	<b>cv.maxu rD, rs1, rs2</b>
010 1111	00000	src1	011	dest	010 1011	<b>cv.exths rD, rs1</b>
011 0000	00000	src1	011	dest	010 1011	<b>cv.exthz rD, rs1</b>
011 0001	00000	src1	011	dest	010 1011	<b>cv.extbs rD, rs1</b>
011 0010	00000	src1	011	dest	010 1011	<b>cv.extbz rD, rs1</b>

31 : 25	24 : 20	19 : 15	14 : 12	11 : 7	6 : 0	
funct7	ls2[4:0]	rs1	funct3	rD	opcode	
011 1000	luimm5[4:0]	src1	011	dest	010 1011	<b>cv.clip rD, rs1, ls2</b>
011 1001	luimm5[4:0]	src1	011	dest	010 1011	<b>cv.clipu rD, rs1, ls2</b>
011 1010	src2	src1	011	dest	010 1011	<b>cv.clipr rD, rs1, rs2</b>
011 1011	src2	src1	011	dest	010 1011	<b>cv.clipur rD, rs1, rs2</b>

31:30	29 : 25	24 :20	19 :15	14 : 12	11 : 7	6 : 0	
f2	ls3[4:0]	rs2	rs1	funct3	rD	opcode	
00	Luimm5[4:0]	src2	src1	010	dest	101 1011	<b>cv.addN rD, rs1, rs2, ls3</b>
01	Luimm5[4:0]	src2	src1	010	dest	101 1011	<b>cv.adduN rD, rs1, rs2, ls3</b>
10	Luimm5[4:0]	src2	src1	010	dest	101 1011	<b>cv.addRN rD, rs1, rs2, ls3</b>
11	Luimm5[4:0]	src2	src1	010	dest	101 1011	<b>cv.adduRN rD, rs1, rs2, ls3</b>
00	Luimm5[4:0]	src2	src1	011	dest	101 1011	<b>cv.subN rD, rs1, rs2, ls3</b>
01	Luimm5[4:0]	src2	src1	011	dest	101 1011	<b>cv.subuN rD, rs1, rs2, ls3</b>
10	Luimm5[4:0]	src2	src1	011	dest	101 1011	<b>cv.subRN rD, rs1, rs2, ls3</b>
11	Luimm5[4:0]	src2	src1	011	dest	101 1011	<b>cv.subuRN rD, rs1, rs2, ls3</b>

31 : 25	24 : 20	19 :15	14 : 12	11 : 7	6 : 0	
funct7	ls3[4:0]	rs1	funct3	rD	opcode	
100 0000	src2	src1	011	dest	010 1011	<b>cv.addNr rD, rs1, rs2</b>
100 0001	src2	src1	011	dest	010 1011	<b>cv.adduNr rD, rs1, rs</b>
100 0010	src2	src1	011	dest	010 1011	<b>cv.addRNr rD, rs1, rs</b>
100 0011	src2	src1	011	dest	010 1011	<b>cv.adduRNr rD, rs1, rs2</b>
100 0100	src2	src1	011	dest	010 1011	<b>cv.subNr rD, rs1, rs2</b>
100 0101	src2	src1	011	dest	010 1011	<b>cv.subuNr rD, rs1, rs2</b>
100 0110	src2	src1	011	dest	010 1011	<b>cv.subRNr rD, rs1, rs2</b>
100 0111	src2	src1	011	dest	010 1011	<b>cv.subuRNr rD, rs1, rs2</b>

### 18.4.6 Immediate Branching Operations

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

### 18.4.7 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	110	Imm12[4:1]	Imm12[11]	000 1011	<b>cv.beqimm rs1, Imm5, Imm12</b>
Imm12[12]	Imm12[10:5]	Imm5	src1	111	Imm12[4:1]	Imm12[11]	000 1011	<b>cv.bneimm rs1, Imm5, Imm12</b>

## 18.5 Multiply-Accumulate

CV32E40P 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`.

### 18.5.1 MAC Operations

#### 32-Bit x 32-Bit Multiplication Operations

Mnemonic	Description	
<b>cv.mac</b>	<b>rD, rs1, rs2</b>	$rD = rD + rs1 * rs2$
<b>cv.msu</b>	<b>rD, rs1, rs2</b>	$rD = rD - rs1 * rs2$

#### 16-Bit x 16-Bit Multiplication

Mnemonic	Description	
<b>cv.muluN</b>	<b>rD, rs1, rs2, Is3</b>	$rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0])) \gg Is3$ Note: Logical shift right.
<b>cv.mulhhuN</b>	<b>rD, rs1, rs2, Is3</b>	$rD[31:0] = (Zext(rs1[31:16]) * Zext(rs2[31:16])) \gg Is3$ Note: Logical shift right.
<b>cv.mulsN</b>	<b>rD, rs1, rs2, Is3</b>	$rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0])) \ggg Is3$ Note: Arithmetic shift right.
<b>cv.mulhhsN</b>	<b>rD, rs1, rs2, Is3</b>	$rD[31:0] = (Sext(rs1[31:16]) * Sext(rs2[31:16])) \ggg Is3$ Note: Arithmetic shift right.
<b>cv.muluRN</b>	<b>rD, rs1, rs2, Is3</b>	$rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0]) + 2^{(Is3-1)}) \gg Is3$ Note: Logical shift right.
<b>cv.mulhhuRN</b>	<b>rD, rs1, rs2, Is3</b>	$rD[31:0] = (Zext(rs1[31:16]) * Zext(rs2[31:16]) + 2^{(Is3-1)}) \gg Is3$ Note: Logical shift right.
<b>cv.mulsRN</b>	<b>rD, rs1, rs2, Is3</b>	$rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0]) + 2^{(Is3-1)}) \ggg Is3$ Note: Arithmetic shift right.
<b>cv.mulhhsRN</b>	<b>rD, rs1, rs2, Is3</b>	$rD[31:0] = (Sext(rs1[31:16]) * Sext(rs2[31:16]) + 2^{(Is3-1)}) \ggg Is3$ Note: Arithmetic shift right.

## 16-Bit x 16-Bit Multiply-Accumulate

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

## 18.5.2 MAC Encoding

31 : 25	24 :20	19 :15	14 : 12	11 : 7	6 : 0	
funct7	rs2	rs1	funct3	rD	opcode	
100 1000	src2	src1	011	dest	010 1011	<b>cv.mac rD, rs1, rs2</b>
100 1001	src2	src1	011	dest	010 1011	<b>cv.msu rD, rs1, rs2</b>

31:30	29 : 25	24 :20	19 :15	14 : 12	11 : 7	6 : 0	
f2	ls3[4:0]	rs2	rs1	funct3	rD	opcode	
00	Luimm5[4:0]	src2	src1	101	dest	101 1011	<b>cv.muluN rD, rs1, rs2, Is3</b>
01	Luimm5[4:0]	src2	src1	101	dest	101 1011	<b>cv.mulhhuN rD, rs1, rs2, Is3</b>
00	Luimm5[4:0]	src2	src1	100	dest	101 1011	<b>cv.mulsN rD, rs1, rs2, Is3</b>
01	Luimm5[4:0]	src2	src1	100	dest	101 1011	<b>cv.mulhhsN rD, rs1, rs2, Is3</b>
10	Luimm5[4:0]	src2	src1	101	dest	101 1011	<b>cv.muluRN rD, rs1, rs2, Is3</b>
11	Luimm5[4:0]	src2	src1	101	dest	101 1011	<b>cv.mulhhuRN rD, rs1, rs2, Is3</b>
10	Luimm5[4:0]	src2	src1	100	dest	101 1011	<b>cv.mulsRN rD, rs1, rs2, Is3</b>
11	Luimm5[4:0]	src2	src1	100	dest	101 1011	<b>cv.mulhhsRN rD, rs1, rs2, Is3</b>
00	Luimm5[4:0]	src2	src1	111	dest	101 1011	<b>cv.macuN rD, rs1, rs2, Is3</b>
01	Luimm5[4:0]	src2	src1	111	dest	101 1011	<b>cv.machhuN rD, rs1, rs2, Is3</b>
00	Luimm5[4:0]	src2	src1	110	dest	101 1011	<b>cv.macsN rD, rs1, rs2, Is3</b>
01	Luimm5[4:0]	src2	src1	110	dest	101 1011	<b>cv.machhsN rD, rs1, rs2, Is3</b>
10	Luimm5[4:0]	src2	src1	110	dest	101 1011	<b>cv.macsRN rD, rs1, rs2, Is3</b>
11	Luimm5[4:0]	src2	src1	110	dest	101 1011	<b>cv.machhsRN rD, rs1, rs2, Is3</b>
10	Luimm5[4:0]	src2	src1	111	dest	101 1011	<b>cv.macuRN rD, rs1, rs2, Is3</b>
11	Luimm5[4:0]	src2	src1	111	dest	101 1011	<b>cv.machhuRN rD, rs1, rs2, Is3</b>



## 18.6 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 of the tool chain** as it is not implementing auto-vectorization up to now. But those instructions can be used either with builtins or even in assembly.

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 bitwidth 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,x2,0x2A` performs:

$$x3[31:16] = x2[31:16] + 0xFFEA$$

$$x3[15: 0] = x2[15: 0] + 0xFFEA$$

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.

And I5, I4, I3, I2, I1 and I0 respectively represent bits 5, 4, 3, 2, 1 and 0 of the immediate value.

## 18.6.1 SIMD ALU Operations

Mnemonic	Description
<b>cv.add[.sc,.sci]{.h,.b}</b>	$rD[i] = (rs1[i] + op2[i]) \& 0xFFFF$
<b>cv.add{.div2,.div4,.div8}.h</b>	$rD[i] = ((rs1[i] + op2[i]) \& 0xFFFF) \gg \{1,2,3\}$
<b>cv.sub[.sc,.sci]{.h,.b}</b>	$rD[i] = (rs1[i] - op2[i]) \& 0xFFFF$
<b>cv.sub{.div2,.div4,.div8}.h</b>	$rD[i] = ((rs1[i] - op2[i]) \& 0xFFFF) \gg \{1,2,3\}$
<b>cv.avg[.sc,.sci]{.h,.b}</b>	$rD[i] = ((rs1[i] + op2[i]) \& \{0xFFFF, 0xFF\}) \gg 1$ Note: Arithmetic right shift.
<b>cv.avgu[.sc,.sci]{.h,.b}</b>	$rD[i] = ((rs1[i] + op2[i]) \& \{0xFFFF, 0xFF\}) \gg 1$
<b>cv.min[.sc,.sci]{.h,.b}</b>	$rD[i] = rs1[i] < op2[i] ? rs1[i] : op2[i]$
<b>cv.minu[.sc,.sci]{.h,.b}</b>	$rD[i] = rs1[i] < op2[i] ? rs1[i] : op2[i]$ Note: Immediate is zero-extended, comparison is unsigned.
<b>cv.max[.sc,.sci]{.h,.b}</b>	$rD[i] = rs1[i] > op2[i] ? rs1[i] : op2[i]$
<b>cv.maxu[.sc,.sci]{.h,.b}</b>	$rD[i] = rs1[i] > op2[i] ? rs1[i] : op2[i]$ Note: Immediate is zero-extended, comparison is unsigned.
<b>cv.srl[.sc,.sci]{.h,.b}</b>	$rD[i] = rs1[i] \gg op2[i]$ Note: Immediate is zero-extended, shift is logical.
<b>cv.sra[.sc,.sci]{.h,.b}</b>	$rD[i] = rs1[i] \ggg op2[i]$ Note: Immediate is zero-extended, shift is arithmetic.
<b>cv.sll[.sc,.sci]{.h,.b}</b>	$rD[i] = rs1[i] \ll op2[i]$ Note: Immediate is zero-extended, shift is logical.
<b>cv.or[.sc,.sci]{.h,.b}</b>	$rD[i] = rs1[i]   op2[i]$
<b>cv.xor[.sc,.sci]{.h,.b}</b>	$rD[i] = rs1[i] \wedge op2[i]$
<b>cv.and[.sc,.sci]{.h,.b}</b>	$rD[i] = rs1[i] \& op2[i]$
<b>cv.abs{.h,.b}</b>	$rD[i] = rs1 < 0 ? -rs1 : rs1$

## 18.6.2 SIMD Bit Manipulation Operations

Mnemonic	Description
<b>cv.extract.h</b>	$rD = \text{Sext}(rs1[I0*16+15:I0*16])$
<b>cv.extract.b</b>	$rD = \text{Sext}(rs1[(I1:I0)*8+7:(I1:I0)*8])$
<b>cv.extractu.h</b>	$rD = \text{Zext}(rs1[I0*16+15:I0*16])$
<b>cv.extractu.b</b>	$rD = \text{Zext}(rs1[(I1:I0)*8+7:(I1:I0)*8])$
<b>cv.insert.h</b>	$rD[I0*16+15:I0*16] = rs1[15:0]$ Note: The rest of the bits of rD are untouched and keep their previous value.
<b>cv.insert.b</b>	$rD[(I1:I0)*8+7:(I1:I0)*8] = rs1[7:0]$ Note: The rest of the bits of rD are untouched and keep their previous value.

## SIMD Dot Product Instructions

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

## SIMD Shuffle and Pack Instructions

Mnemonic	Description
<b>cv.shuffleb</b>	$rD[31:16] = rs1[rs2[16]*16+15:rs2[16]*16]$ $rD[15:0] = rs1[rs2[0]*16+15:rs2[0]*16]$
<b>cv.shuffleh</b>	$rD[31:16] = rs1[I1*16+15:I1*16]$ $rD[15:0] = rs1[I0*16+15:I0*16]$
<b>cv.shufflew</b>	$rD[31:24] = rs1[rs2[25:24]*8+7:rs2[25:24]*8]$ $rD[23:16] = rs1[rs2[17:16]*8+7:rs2[17:16]*8]$ $rD[15:8] = rs1[rs2[9:8]*8+7:rs2[9:8]*8]$ $rD[7:0] = rs1[rs2[1:0]*8+7:rs2[1:0]*8]$
<b>cv.shuffleb2</b>	$rD[31:16] = rs1[7:0]$ $rD[23:16] = rs1[(I5:I4)*8+7:(I5:I4)*8]$ $rD[15:8] = rs1[(I3:I2)*8+7:(I3:I2)*8]$ $rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]$
<b>cv.shuffleh2</b>	$rD[31:16] = rs1[15:8]$ $rD[23:16] = rs1[(I5:I4)*8+7:(I5:I4)*8]$ $rD[15:8] = rs1[(I3:I2)*8+7:(I3:I2)*8]$ $rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]$
<b>cv.shufflew2</b>	$rD[31:16] = rs1[23:16]$ $rD[23:16] = rs1[(I5:I4)*8+7:(I5:I4)*8]$ $rD[15:8] = rs1[(I3:I2)*8+7:(I3:I2)*8]$ $rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]$
<b>cv.shuffleb3</b>	$rD[31:16] = rs1[31:24]$ $rD[23:16] = rs1[(I5:I4)*8+7:(I5:I4)*8]$ $rD[15:8] = rs1[(I3:I2)*8+7:(I3:I2)*8]$ $rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]$
<b>cv.shufflew3</b>	$rD[31:16] = ((rs2[17] == 1) ? rs1 : rD)[rs2[16]*16+15:rs2[16]*16]$ $rD[15:0] = ((rs2[1] == 1) ? rs1 : rD)[rs2[0]*16+15:rs2[0]*16]$
<b>cv.shufflew4</b>	$rD[31:24] = ((rs2[26] == 1) ? rs1 : rD)[rs2[25:24]*8+7:rs2[25:24]*8]$ $rD[23:16] = ((rs2[18] == 1) ? rs1 : rD)[rs2[17:16]*8+7:rs2[17:16]*8]$ $rD[15:8] = ((rs2[10] == 1) ? rs1 : rD)[rs2[9:8]*8+7:rs2[9:8]*8]$ $rD[7:0] = ((rs2[2] == 1) ? rs1 : rD)[rs2[1:0]*8+7:rs2[1:0]*8]$
<b>cv.packb</b>	$rD[31:16] = rs1[15:0]$ $rD[15:0] = rs2[15:0]$
<b>cv.packh</b>	$rD[31:16] = rs1[31:16]$ $rD[15:0] = rs2[31:16]$
<b>cv.packw</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.
<b>cv.packw2</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.

## 18.6.3 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	111 1011	cv.a
0 0000	0	0	src2	src1	100	dest	111 1011	cv.a
0 0000	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.a
0 0000	0	0	src2	src1	001	dest	111 1011	cv.a
0 0000	0	0	src2	src1	101	dest	111 1011	cv.a
0 0000	0	Imm6[5:0]		src1	111	dest	111 1011	cv.a
0 1011	1	0	src2	src1	010	dest	111 1011	cv.a
0 1011	1	0	src2	src1	100	dest	111 1011	cv.a
0 1011	1	0	src2	src1	110	dest	111 1011	cv.a
0 0001	0	0	src2	src1	000	dest	111 1011	cv.s
0 0001	0	0	src2	src1	100	dest	111 1011	cv.s
0 0001	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.s
0 0001	0	0	src2	src1	001	dest	111 1011	cv.s
0 0001	0	0	src2	src1	101	dest	111 1011	cv.s
0 0001	0	Imm6[5:0]		src1	111	dest	111 1011	cv.s
0 1100	1	0	src2	src1	010	dest	111 1011	cv.s
0 1100	1	0	src2	src1	100	dest	111 1011	cv.s
0 1100	1	0	src2	src1	110	dest	111 1011	cv.s
0 0010	0	0	src2	src1	000	dest	111 1011	cv.a
0 0010	0	0	src2	src1	100	dest	111 1011	cv.a
0 0010	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.a
0 0010	0	0	src2	src1	001	dest	111 1011	cv.a
0 0010	0	0	src2	src1	101	dest	111 1011	cv.a
0 0010	0	Imm6[5:0]		src1	111	dest	111 1011	cv.a
0 0011	0	0	src2	src1	000	dest	111 1011	cv.a
0 0011	0	0	src2	src1	100	dest	111 1011	cv.a
0 0011	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.a
0 0011	0	0	src2	src1	001	dest	111 1011	cv.a
0 0011	0	0	src2	src1	101	dest	111 1011	cv.a
0 0011	0	Imm6[5:0]		src1	111	dest	111 1011	cv.a
0 0100	0	0	src2	src1	000	dest	111 1011	cv.m
0 0100	0	0	src2	src1	100	dest	111 1011	cv.m
0 0100	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.m
0 0100	0	0	src2	src1	001	dest	111 1011	cv.m
0 0100	0	0	src2	src1	101	dest	111 1011	cv.m
0 0100	0	Imm6[5:0]		src1	111	dest	111 1011	cv.m
0 0101	0	0	src2	src1	000	dest	111 1011	cv.m
0 0101	0	0	src2	src1	100	dest	111 1011	cv.m
0 0101	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.m
0 0101	0	0	src2	src1	001	dest	111 1011	cv.m
0 0101	0	0	src2	src1	101	dest	111 1011	cv.m
0 0101	0	Imm6[5:0]		src1	111	dest	111 1011	cv.m
0 0110	0	0	src2	src1	000	dest	111 1011	cv.m
0 0110	0	0	src2	src1	100	dest	111 1011	cv.m
0 0110	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.m
0 0110	0	0	src2	src1	001	dest	111 1011	cv.m

Table 18.2 – 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	111 1011	cv.m
0 0110	0	Imm6[5:0]		src1	111	dest	111 1011	cv.m
0 0111	0	0	src2	src1	000	dest	111 1011	cv.m
0 0111	0	0	src2	src1	100	dest	111 1011	cv.m
0 0111	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.m
0 0111	0	0	src2	src1	001	dest	111 1011	cv.m
0 0111	0	0	src2	src1	101	dest	111 1011	cv.m
0 0111	0	Imm6[5:0]		src1	111	dest	111 1011	cv.m
0 1000	0	0	src2	src1	000	dest	111 1011	cv.sl
0 1000	0	0	src2	src1	100	dest	111 1011	cv.sl
0 1000	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.sl
0 1000	0	0	src2	src1	001	dest	111 1011	cv.sl
0 1000	0	0	src2	src1	101	dest	111 1011	cv.sl
0 1000	0	Imm6[5:0]		src1	111	dest	111 1011	cv.sl
0 1001	0	0	src2	src1	000	dest	111 1011	cv.sl
0 1001	0	0	src2	src1	100	dest	111 1011	cv.sl
0 1001	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.sl
0 1001	0	0	src2	src1	001	dest	111 1011	cv.sl
0 1001	0	0	src2	src1	101	dest	111 1011	cv.sl
0 1001	0	Imm6[5:0]		src1	111	dest	111 1011	cv.sl
0 1010	0	0	src2	src1	000	dest	111 1011	cv.sl
0 1010	0	0	src2	src1	100	dest	111 1011	cv.sl
0 1010	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.sl
0 1010	0	0	src2	src1	001	dest	111 1011	cv.sl
0 1010	0	0	src2	src1	101	dest	111 1011	cv.sl
0 1010	0	Imm6[5:0]		src1	111	dest	111 1011	cv.sl
0 1011	0	0	src2	src1	000	dest	111 1011	cv.o
0 1011	0	0	src2	src1	100	dest	111 1011	cv.o
0 1011	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.o
0 1011	0	0	src2	src1	001	dest	111 1011	cv.o
0 1011	0	0	src2	src1	101	dest	111 1011	cv.o
0 1011	0	Imm6[5:0]		src1	111	dest	111 1011	cv.o
0 1100	0	0	src2	src1	000	dest	111 1011	cv.x
0 1100	0	0	src2	src1	100	dest	111 1011	cv.x
0 1100	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.x
0 1100	0	0	src2	src1	001	dest	111 1011	cv.x
0 1100	0	0	src2	src1	101	dest	111 1011	cv.x
0 1100	0	Imm6[5:0]		src1	111	dest	111 1011	cv.x
0 1101	0	0	src2	src1	000	dest	111 1011	cv.a
0 1101	0	0	src2	src1	100	dest	111 1011	cv.a
0 1101	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.a
0 1101	0	0	src2	src1	001	dest	111 1011	cv.a
0 1101	0	0	src2	src1	101	dest	111 1011	cv.a
0 1101	0	Imm6[5:0]		src1	111	dest	111 1011	cv.a
0 1110	0	0	0	src1	000	dest	111 1011	cv.a
0 1110	0	0	0	src1	001	dest	111 1011	cv.a
1 1000	0	Imm6[5:0]		src1	000	dest	111 1011	cv.e
1 1000	0	Imm6[5:0]		src1	001	dest	111 1011	cv.e

Table 18.2 – 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 1000	0	Imm6[5:0]		src1	010	dest	111 1011	cv.ex
1 1000	0	Imm6[5:0]		src1	011	dest	111 1011	cv.ex
1 1000	0	Imm6[5:0]		src1	100	dest	111 1011	cv.in
1 1000	0	Imm6[5:0]		src1	101	dest	111 1011	cv.in
1 0000	0	0	src2	src1	000	dest	111 1011	cv.d
1 0000	0	0	src2	src1	100	dest	111 1011	cv.d
1 0000	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.d
1 0000	0	0	src2	src1	001	dest	111 1011	cv.d
1 0000	0	0	src2	src1	101	dest	111 1011	cv.d
1 0000	0	Imm6[5:0]		src1	111	dest	111 1011	cv.d
1 0001	0	0	src2	src1	000	dest	111 1011	cv.d
1 0001	0	0	src2	src1	100	dest	111 1011	cv.d
1 0001	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.d
1 0001	0	0	src2	src1	001	dest	111 1011	cv.d
1 0001	0	0	src2	src1	101	dest	111 1011	cv.d
1 0001	0	Imm6[5:0]		src1	111	dest	111 1011	cv.d
1 0010	0	0	src2	src1	000	dest	111 1011	cv.d
1 0010	0	0	src2	src1	100	dest	111 1011	cv.d
1 0010	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.d
1 0010	0	0	src2	src1	001	dest	111 1011	cv.d
1 0010	0	0	src2	src1	101	dest	111 1011	cv.d
1 0010	0	Imm6[5:0]		src1	111	dest	111 1011	cv.d
1 0011	0	0	src2	src1	000	dest	111 1011	cv.sc
1 0011	0	0	src2	src1	100	dest	111 1011	cv.sc
1 0011	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.sc
1 0011	0	0	src2	src1	001	dest	111 1011	cv.sc
1 0011	0	0	src2	src1	101	dest	111 1011	cv.sc
1 0011	0	Imm6[5:0]		src1	111	dest	111 1011	cv.sc
1 0100	0	0	src2	src1	000	dest	111 1011	cv.sc
1 0100	0	0	src2	src1	100	dest	111 1011	cv.sc
1 0100	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.sc
1 0100	0	0	src2	src1	001	dest	111 1011	cv.sc
1 0100	0	0	src2	src1	101	dest	111 1011	cv.sc
1 0100	0	Imm6[5:0]		src1	111	dest	111 1011	cv.sc
1 0101	0	0	src2	src1	000	dest	111 1011	cv.sc
1 0101	0	0	src2	src1	100	dest	111 1011	cv.sc
1 0101	0	Imm6[5:0]s		src1	110	dest	111 1011	cv.sc
1 0101	0	0	src2	src1	001	dest	111 1011	cv.sc
1 0101	0	0	src2	src1	101	dest	111 1011	cv.sc
1 0101	0	Imm6[5:0]		src1	111	dest	111 1011	cv.sc
1 1001	0	0	src2	src1	000	dest	111 1011	cv.sl
1 1001	0	Imm6[5:0]		src1	110	dest	111 1011	cv.sl
1 1001	0	0	src2	src1	001	dest	111 1011	cv.sl
1 1001	0	Imm6[5:0]		src1	111	dest	111 1011	cv.sl
1 1010	0	Imm6[5:0]		src1	111	dest	111 1011	cv.sl
1 1011	0	Imm6[5:0]		src1	111	dest	111 1011	cv.sl
1 1100	0	Imm6[5:0]		src1	111	dest	111 1011	cv.sl
1 1010	0	0	src2	src1	000	dest	111 1011	cv.sl

Table 18.2 – 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 1010	0	0	src2	src1	001	dest	111 1011	cv.sl
1 1101	0	0	src2	src1	000	dest	111 1011	cv.p
1 1101	0	0	src2	src1	100	dest	111 1011	cv.p
1 1110	0	0	src2	src1	001	dest	111 1011	cv.p
1 1111	0	0	src2	src1	001	dest	111 1011	cv.p

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

## 18.6.4 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.cmp <sub>eq</sub> [.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] == op2 ? '1' : '0'
cv.cmp <sub>ne</sub> [.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] != op2 ? '1' : '0'
cv.cmp <sub>gt</sub> [.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] > op2 ? '1' : '0'
cv.cmp <sub>ge</sub> [.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] >= op2 ? '1' : '0'
cv.cmp <sub>lt</sub> [.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] < op2 ? '1' : '0'
cv.cmp <sub>le</sub> [.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] <= op2 ? '1' : '0'
cv.cmp <sub>gtu</sub> [.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] > op2 ? '1' : '0' Note: Unsigned comparison.
cv.cmp <sub>geu</sub> [.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] >= op2 ? '1' : '0' Note: Unsigned comparison.
cv.cmp <sub>ltu</sub> [.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] < op2 ? '1' : '0' Note: Unsigned comparison.
cv.cmp <sub>leu</sub> [.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] <= op2 ? '1' : '0' Note: Unsigned comparison.

## 18.6.5 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	111 1011	cv.c
0 0000	1	0	src2	src1	100	dest	111 1011	cv.c
0 0000	1	Imm6[5:0]		src1	110	dest	111 1011	cv.c
0 0000	1	0	src2	src1	001	dest	111 1011	cv.c
0 0000	1	0	src2	src1	101	dest	111 1011	cv.c
0 0000	1	Imm6[5:0]		src1	111	dest	111 1011	cv.c
0 0001	1	0	src2	src1	000	dest	111 1011	cv.c



Table 18.3 – 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	100	dest	111 1011	cv.0
0 0001	1	Imm6[5:0]		src1	110	dest	111 1011	cv.0
0 0001	1	0	src2	src1	001	dest	111 1011	cv.0
0 0001	1	0	src2	src1	101	dest	111 1011	cv.0
0 0001	1	Imm6[5:0]		src1	111	dest	111 1011	cv.0
0 0010	1	0	src2	src1	000	dest	111 1011	cv.0
0 0010	1	0	src2	src1	100	dest	111 1011	cv.0
0 0010	1	Imm6[5:0]		src1	110	dest	111 1011	cv.0
0 0010	1	0	src2	src1	001	dest	111 1011	cv.0
0 0010	1	0	src2	src1	101	dest	111 1011	cv.0
0 0010	1	Imm6[5:0]		src1	111	dest	111 1011	cv.0
0 0011	1	0	src2	src1	000	dest	111 1011	cv.0
0 0011	1	0	src2	src1	100	dest	111 1011	cv.0
0 0011	1	Imm6[5:0]		src1	110	dest	111 1011	cv.0
0 0011	1	0	src2	src1	001	dest	111 1011	cv.0
0 0011	1	0	src2	src1	101	dest	111 1011	cv.0
0 0011	1	Imm6[5:0]		src1	111	dest	111 1011	cv.0
0 0100	1	0	src2	src1	000	dest	111 1011	cv.0
0 0100	1	0	src2	src1	100	dest	111 1011	cv.0
0 0100	1	Imm6[5:0]		src1	110	dest	111 1011	cv.0
0 0100	1	0	src2	src1	001	dest	111 1011	cv.0
0 0100	1	0	src2	src1	101	dest	111 1011	cv.0
0 0100	1	Imm6[5:0]		src1	111	dest	111 1011	cv.0
0 0101	1	0	src2	src1	000	dest	111 1011	cv.0
0 0101	1	0	src2	src1	100	dest	111 1011	cv.0
0 0101	1	Imm6[5:0]		src1	110	dest	111 1011	cv.0
0 0101	1	0	src2	src1	001	dest	111 1011	cv.0
0 0101	1	0	src2	src1	101	dest	111 1011	cv.0
0 0101	1	Imm6[5:0]		src1	111	dest	111 1011	cv.0
0 0110	1	0	src2	src1	000	dest	111 1011	cv.0
0 0110	1	0	src2	src1	100	dest	111 1011	cv.0
0 0110	1	Imm6[5:0]		src1	110	dest	111 1011	cv.0
0 0110	1	0	src2	src1	001	dest	111 1011	cv.0
0 0110	1	0	src2	src1	101	dest	111 1011	cv.0
0 0110	1	Imm6[5:0]		src1	111	dest	111 1011	cv.0
0 0111	1	0	src2	src1	000	dest	111 1011	cv.0
0 0111	1	0	src2	src1	100	dest	111 1011	cv.0
0 0111	1	Imm6[5:0]		src1	110	dest	111 1011	cv.0
0 0111	1	0	src2	src1	001	dest	111 1011	cv.0
0 0111	1	0	src2	src1	101	dest	111 1011	cv.0
0 0111	1	Imm6[5:0]		src1	111	dest	111 1011	cv.0
0 1000	1	0	src2	src1	000	dest	111 1011	cv.0
0 1000	1	0	src2	src1	100	dest	111 1011	cv.0
0 1000	1	Imm6[5:0]		src1	110	dest	111 1011	cv.0
0 1000	1	0	src2	src1	001	dest	111 1011	cv.0
0 1000	1	0	src2	src1	101	dest	111 1011	cv.0
0 1000	1	Imm6[5:0]		src1	111	dest	111 1011	cv.0
0 1001	1	0	src2	src1	000	dest	111 1011	cv.0

Table 18.3 – 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	100	dest	111 1011	cv.subrotmj rD, rs1, rs2
0 1001	1	Imm6[5:0]		src1	110	dest	111 1011	cv.subrotmj.div2 rD, rs1, rs2
0 1001	1	0	src2	src1	001	dest	111 1011	cv.subrotmj.div4 rD, rs1, rs2
0 1001	1	0	src2	src1	101	dest	111 1011	cv.subrotmj.div8 rD, rs1, rs2
0 1001	1	Imm6[5:0]		src1	111	dest	111 1011	cv.cplxconj rD, rs1

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

## 18.6.6 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, rs1, rs2	$rD[0] = ((rs1[1] - rs2[1]) \& 0xFFFF) \gg \{0,1,2,3\}$ $rD[1] = ((rs2[0] - rs1[0]) \& 0xFFFF) \gg \{0,1,2,3\}$
cv.cplxconj rD, rs1	$rD[0] = rs1[0]$ $rD[1] = -rs1[1]$
cv.cplxmul.r{/,div2,div4,div8} rD, rs1, rs2	$rD[15:0] = (rs1[0]*rs2[0] - rs1[1]*rs2[1]) \gg \{15,16,17,18\}$ $rD[31:16] = rD[31:16]$
cv.cplxmul.i{/,div2,div4,div8} rD, rs1, rs2	$rD[31:16] = (rs1[0]*rs2[1] + rs1[1]*rs2[0]) \gg \{15,16,17,18\}$ $rD[15:0] = rD[15:0]$

## 18.6.7 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	0	src2	src1	000	dest	111 1011	cv.subrotmj rD, rs1, rs2
0 1101	1	0	src2	src1	010	dest	111 1011	cv.subrotmj.div2 rD, rs1, rs2
0 1101	1	0	src2	src1	100	dest	111 1011	cv.subrotmj.div4 rD, rs1, rs2
0 1101	1	0	src2	src1	110	dest	111 1011	cv.subrotmj.div8 rD, rs1, rs2
0 1011	1	0	00000	src1	000	dest	111 1011	cv.cplxconj rD, rs1
0 1010	1	0	src2	src1	000	dest	111 1011	cv.cplxmul.r rD, rs1, rs2
0 1010	1	0	src2	src1	010	dest	111 1011	cv.cplxmul.r.div2 rD, rs1, rs2
0 1010	1	0	src2	src1	100	dest	111 1011	cv.cplxmul.r.div4 rD, rs1, rs2
0 1010	1	0	src2	src1	110	dest	111 1011	cv.cplxmul.r.div8 rD, rs1, rs2
0 1010	1	0	src2	src1	001	dest	111 1011	cv.cplxmul.i rD, rs1, rs2
0 1010	1	0	src2	src1	011	dest	111 1011	cv.cplxmul.i.div2 rD, rs1, rs2
0 1010	1	0	src2	src1	101	dest	111 1011	cv.cplxmul.i.div4 rD, rs1, rs2
0 1010	1	0	src2	src1	111	dest	111 1011	cv.cplxmul.i.div8 rD, rs1, rs2

## CORE VERSIONS AND RTL FREEZE RULES

The CV32E40P 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 RTL Freeze version of the core is identified by a GitHub tag with the format `cv32e40p_vMAJOR.MINOR.PATCH` (e.g. `cv32e40p_v1.0.0`). In addition, the release date is reported in the documentation.

### 19.1 What happens after RTL Freeze?

#### 19.1.1 RTL changes on verified parameters

Minor changes to the RTL on a frozen parameter set (e.g., nicer RTL code, clearer RTL code, etc) are allowed if, and only if, they are logically equivalent to the frozen (tagged) version of the core. This is guaranteed by a CI flow that checks that pull requests are logically equivalent to a specific tag of the core as explained [here](#). For example, suppose we re-write “better” a portion of the ALU that affects the frozen set of parameters of the version `cv32e40p_v1.0.0`, for instance, the adder. In that case, the proposed changes are compared with the code based on `cv32e40p_v1.0.0`, and if they are logically equivalent, they are accepted. Otherwise, they are rejected. See below for more case scenarios.

#### 19.1.2 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.

#### 19.1.3 RTL changes on non-verified yet 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.

### 19.1.4 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.

Figure 19.1 shows the aforementioned rules.

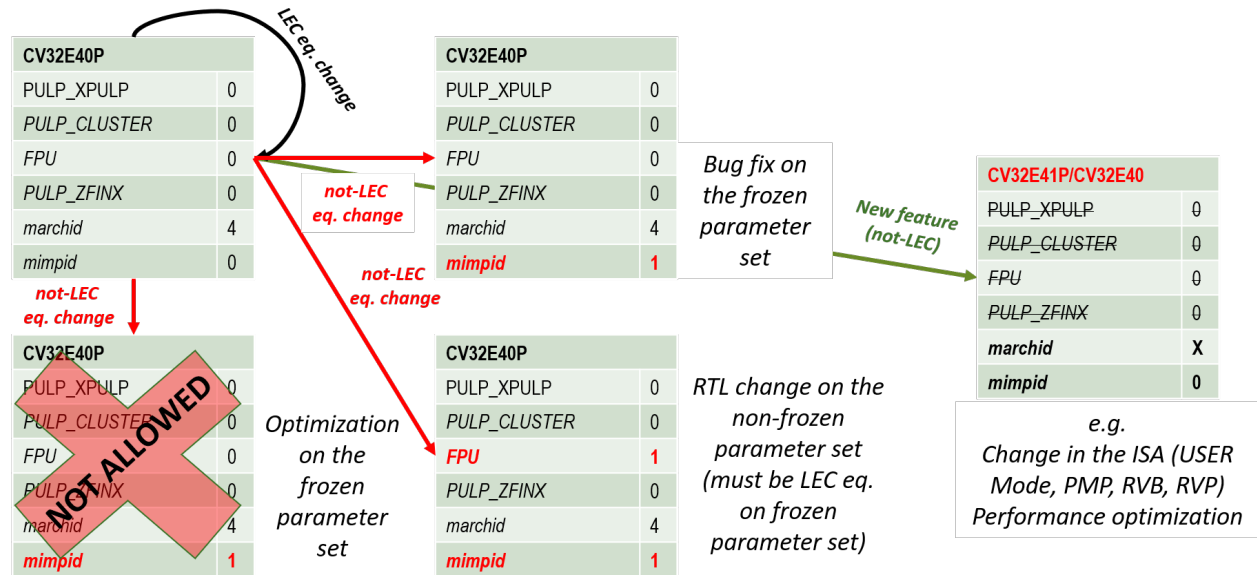


Figure 19.1: Versions control of CV32E40P

## 19.2 Released core versions

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

### 19.3 mimpid=0

The `mimpid=0` refers to the CV32E40P core verified with the following parameters:

Name	Value
FPU	0
NUM_MHPMCOUNTERS	1
PULP_CLUSTER	0
PULP_XPULP	0
PULP_ZFINX	0

Following, all the GitHub tags related to `mimpid=0`.

Git Tag	Tagged By	Date	Reason for Release	Comment
cv32e40p_v1.0.0	Arjan Bink	2020-12-10	RTL Freeze	

The list of open (waived) issues at the time of applying the cv32e40p\_v1.0.0 tag can be found at:

- [https://github.com/openhwgroup/core-v-docs/blob/master/program/milestones/CV32E40P/RTL\\_Freeze\\_v1.0.0/Design\\_openissues.md](https://github.com/openhwgroup/core-v-docs/blob/master/program/milestones/CV32E40P/RTL_Freeze_v1.0.0/Design_openissues.md)
- [https://github.com/openhwgroup/core-v-docs/blob/master/program/milestones/CV32E40P/RTL\\_Freeze\\_v1.0.0/Verification\\_openissues.md](https://github.com/openhwgroup/core-v-docs/blob/master/program/milestones/CV32E40P/RTL_Freeze_v1.0.0/Verification_openissues.md)
- [https://github.com/openhwgroup/core-v-docs/blob/master/program/milestones/CV32E40P/RTL\\_Freeze\\_v1.0.0/Documentation\\_openissues.md](https://github.com/openhwgroup/core-v-docs/blob/master/program/milestones/CV32E40P/RTL_Freeze_v1.0.0/Documentation_openissues.md)



## 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