GROUP OPENHW®

PROVEN PROCESSOR IP

CV32E40P User Manual

Release v1.8.0

OpenHW Group

Apr 18, 2024

CONTENTS:

1 Changelog **1**
- 1.1 cv32e40p_v1.8.0 (v2 RTL Freeze tentative) . . . 1
- 1.2 cv32e40p_v1.7.2 . . . 1
- 1.3 cv32e40p_v1.7.1 . . . 1
- 1.4 cv32e40p_v1.7.0 . . . 1
- 1.5 cv32e40p_v1.6.0 . . . 1
- 1.6 cv32e40p_v1.5.0 . . . 1
- 1.7 cv32e40p_v1.4.1 . . . 2
- 1.8 cv32e40p_v1.4.0 . . . 2
- 1.9 cv32e40p_v1.3.2 . . . 2
- 1.10 cv32e40p_v1.3.1 . . . 2
- 1.11 cv32e40p_v1.3.0 . . . 2
- 1.12 cv32e40p_v1.2.1 . . . 2
- 1.13 cv32e40p_v1.2.0 . . . 2
- 1.14 cv32e40p_v1.0.0_doc . . . 2
- 1.15 cv32e40p_v1.1.0 . . . 3
- 1.16 cv32e40p_v1.0.0 . . . 3
- 1.17 pulpissimo-v1.0.0 . . . 3
- 1.18 pulpino-v1.0.0 . . . 3

2 Introduction **5**
- 2.1 License . . . 6
- 2.2 Bus Interfaces . . . 6
- 2.3 Standards Compliance . . . 6
- 2.4 Contents . . . 7
- 2.5 History . . . 8
  - 2.5.1 Memory-Protocol . . . 8
  - 2.5.2 RV32F Extensions . . . 8
  - 2.5.3 RV32A Extensions, Security and Memory Protection . . . 8
  - 2.5.4 CSR Address Re-Mapping . . . 8
  - 2.5.5 Interrupts . . . 8
  - 2.5.6 PULP HWLoop Spec . . . 9
  - 2.5.7 Compliancy, bug fixing, code clean-up, and documentation . . . 9
- 2.6 References . . . 9
- 2.7 Contributors . . . 9

3 Core Integration **11**
- 3.1 Instantiation Template . . . 11
- 3.2 Parameters . . . 13
- 3.3 Interfaces . . . 14

3.4 Clock Gating Cell . . . 14
3.5 Synthesis guidelines . . . 15
3.5.1 ASIC Synthesis . . . 15
3.5.2 FPGA Synthesis . . . 15
3.5.3 Synthesizing with the FPU . . . 15

4 Floating Point Unit (FPU) 17
4.1 CVFPU parameters . . . 17
4.2 FP Register File . . . 19
4.3 FP CSR . . . 19
4.4 Reminder for programmers . . . 19

5 Verification 21
5.1 v1.0.0 verification . . . 21
5.2 v2.0.0 verification . . . 23
5.2.1 Formal verification . . . 24
5.2.2 Simulation verification . . . 24
5.2.3 Results summary . . . 24
5.3 Tracer . . . 26
5.3.1 Output file . . . 26
5.3.2 Trace output format . . . 26

6 CORE-V Hardware Loop feature 27
6.1 Hardware Loop constraints . . . 27
6.2 Hardware loops impact on application, exceptions handlers and debugger . . . 29
6.2.1 Application and ebreak/ecall exception handlers . . . 29
6.2.2 Interrupt handlers . . . 30
6.2.3 Illegal instruction exception handler . . . 30
6.2.4 Debugger . . . 30

7 CORE-V Instruction Set Custom Extensions 31
7.1 Pseudo-instructions . . . 31
7.2 Post-Increment Load & Store Instructions and Register-Register Load & Store Instructions . . . 32
7.2.1 Load operations . . . 32
7.2.2 Store operations . . . 33
7.2.3 Encoding . . . 33
7.3 Event Load Instruction . . . 35
7.3.1 Event Load operation . . . 35
7.3.2 Encoding . . . 35
7.4 Hardware Loops . . . 35
7.4.1 Hardware Loops operations . . . 35
7.4.2 Encoding . . . 36
7.5 ALU . . . 36
7.5.1 Bit Reverse Instruction . . . 37
7.5.2 Bit Manipulation operations . . . 39
7.5.3 Bit Manipulation Encoding . . . 40
7.5.4 General ALU operations . . . 40
7.5.5 General ALU Encoding . . . 42
7.5.6 Immediate Branching operations . . . 43
7.5.7 Immediate Branching Encoding . . . 44
7.6 Multiply-Accumulate . . . 44
7.6.1 16-Bit x 16-Bit Multiplication operations . . . 44
7.6.2 16-Bit x 16-Bit Multiplication pseudo-instructions . . . 45
7.6.3 16-Bit x 16-Bit Multiply-Accumulate operations . . . 45
7.6.4 32-Bit x 32-Bit Multiply-Accumulate operations . . . 45

7.6.5 Encoding . . . 46
7.7 SIMD . . . 46
7.7.1 SIMD ALU operations . . . 48
7.7.2 SIMD Comparison operations . . . 55
7.7.3 SIMD Comparison Encoding . . . 56
7.7.4 SIMD Complex-number operations . . . 57
7.7.5 SIMD Complex-number Encoding . . . 58

8 Performance Counters **59**
8.1 Event Selector . . . 59
8.2 Controlling the counters from software . . . 60
8.3 Parametrization at synthesis time . . . 60
8.4 Time Registers (`time(h)`) . . . 61

9 Control and Status Registers **63**
9.1 CSR Map . . . 63
9.2 CSR Descriptions . . . 65
9.2.1 Floating-point CSRs . . . 65
9.2.2 Hardware Loops CSRs . . . 67
9.2.3 Other CSRs . . . 67
9.2.4 Trigger CSRs . . . 71
9.2.5 Debug CSRs . . . 73
9.2.6 Performances counters . . . 74
9.2.7 ID CSRs . . . 78
9.2.8 Non-RISC-V CSRs . . . 81

10 Exceptions and Interrupts **83**
10.1 Interrupt Interface . . . 83
10.2 Interrupts . . . 84
10.3 Exceptions . . . 84
10.4 Nested Interrupt/Exception Handling . . . 85

11 Debug & Trigger **87**
11.1 Debug Interface . . . 88
11.2 Core Debug Registers . . . 88
11.3 Debug state . . . 89
11.4 EBREAK Behavior . . . 91
11.4.1 Scenario 1 : Enter Exception . . . 91
11.4.2 Scenario 2 : Enter Debug Mode . . . 91
11.4.3 Scenario 3 : Exit Program Buffer & Restart Debug Code . . . 92
11.5 Interrupts during Single-Step Behavior . . . 92

12 Pipeline Details **93**
12.1 Hazards . . . 94
12.2 Single- and Multi-Cycle Instructions . . . 94

13 Instruction Fetch **97**
13.1 Misaligned Accesses . . . 97
13.2 Protocol . . . 98

14 Load-Store-Unit (LSU) **101**
14.1 Misaligned Accesses . . . 101
14.2 Protocol . . . 102
14.3 Post-Incrementing Load and Store Instructions . . . 105

15 Register File **107**
15.1 Floating-Point Register File . . . 107

16 Sleep Unit **109**
16.1 Startup behavior . . . 110
16.2 WFI . . . 110
16.3 PULP Cluster Extension . . . 111

17 Core Versions and RTL Freeze Rules **113**
17.1 What happens after RTL Freeze? . . . 113
17.1.1 RTL changes on verified parameters . . . 113
17.1.2 A bug is found . . . 113
17.1.3 RTL changes on non-verified yet parameters . . . 113
17.1.4 PPA optimizations and new features . . . 114
17.2 Non-backward compatibility . . . 114
17.2.1 Parameters . . . 114
17.3 Released core versions . . . 115
17.3.1 cv32e40p_v1.0.0 . . . 115
17.3.2 cv32e40p_v2.0.0 . . . 115

18 Glossary **117**

CHAPTER

ONE

CHANGELOG

1.1 cv32e40p_v1.8.0 (v2 RTL Freeze tentative)

Released on 2024-04-18 - GitHub

1.2 cv32e40p_v1.7.2

Released on 2024-04-10 - GitHub

1.3 cv32e40p_v1.7.1

Released on 2024-04-10 - GitHub

1.4 cv32e40p_v1.7.0

Released on 2024-03-26 - GitHub

1.5 cv32e40p_v1.6.0

Released on 2024-03-22 - GitHub

1.6 cv32e40p_v1.5.0

Released on 2023-11-30 - GitHub

1.7 cv32e40p_v1.4.1

Released on 2023-09-08 - GitHub

1.8 cv32e40p_v1.4.0

Released on 2023-08-11 - GitHub

1.9 cv32e40p_v1.3.2

Released on 2023-06-27 - GitHub

1.10 cv32e40p_v1.3.1

Released on 2023-05-16 - GitHub

1.11 cv32e40p_v1.3.0

Released on 2023-04-18 - GitHub

1.12 cv32e40p_v1.2.1

Released on 2023-01-26 - GitHub

1.13 cv32e40p_v1.2.0

Released on 2022-12-16 - GitHub

1.14 cv32e40p_v1.0.0_doc

Released on 2022-12-01 - GitHub

1.15 cv32e40p_v1.1.0

Released on 2022-11-14 - GitHub

1.16 cv32e40p_v1.0.0

Released on 2020-12-10 - GitHub

1.17 pulpissimo-v1.0.0

Released on 2018-01-23 - GitHub

1.18 pulpino-v1.0.0

Released on 2018-01-23 - GitHub

CHAPTER

TWO

INTRODUCTION

CV32E40P is a 4-stage in-order 32-bit RISC-V processor core. The ISA of CV32E40P has been extended to support multiple additional instructions including hardware loops, post-increment load and store instructions, additional ALU instructions and SIMD instructions that are not part of the standard RISC-V ISA. Figure 2.1 shows a block diagram of the top level with the core and the FPU.

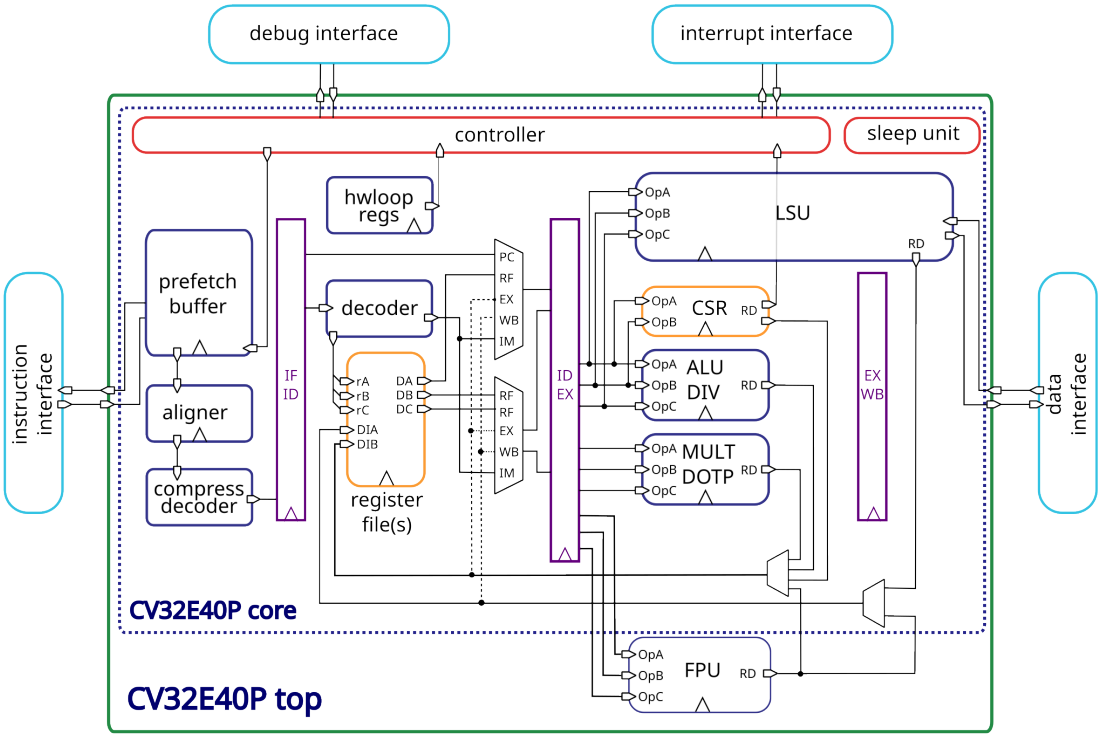

Figure 2.1: Block Diagram of CV32E40P RISC-V Core

2.1 License

Copyright 2023 OpenHW Group.

Copyright 2018 ETH Zurich and University of Bologna.

Copyright and related rights are licensed under the Solderpad Hardware License, Version 0.51 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://solderpad.org/licenses/SHL-0.51. Unless required by applicable law or agreed to in writing, software, hardware and materials distributed under this License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

2.2 Bus Interfaces

The Instruction Fetch and Load/Store data bus interfaces are compliant to the **OBI** (Open Bus Interface) protocol. See 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.

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

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

The following custom instruction set extensions are available.

Table 2.2: CV32E40P Custom Instruction Set Extensions

Custom Extension	Version	Configurability
Xcv: CORE-V PULP ISA Extensions	1.0	optionally enabled with the `COREV_PULP` parameter
Xcvelw: CORE-V PULP Cluster ISA Extension	1.0	optionally enabled with the `COREV_CLUSTER` 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 Contents

- *Core Integration* provides the instantiation template and gives descriptions of the design parameters as well as the input and output ports. It gives synthesis guidelines as well, especially with respect to the Floating-Point Unit.
- *Floating Point Unit (FPU)* describes the Floating Point Unit (FPU).
- *Verification* gives a brief overview of the verification methodology.
- *CORE-V Hardware Loop feature* describes the PULP Hardware Loop extension.
- *CORE-V Instruction Set Custom Extensions* describes the custom instruction set extensions.
- *Performance Counters* gives an overview of the performance monitors and event counters available in CV32E40P.
- The control and status registers are explained in *Control and Status Registers*.
- *Exceptions and Interrupts* deals with the infrastructure for handling exceptions and interrupts.
- *Debug & Trigger* gives a brief overview on the debug infrastructure.
- *Pipeline Details* described the overal pipeline structure.
- The instruction and data interfaces of CV32E40P are explained in *Instruction Fetch* and *Load-Store-Unit (LSU)*, respectively.
- The register-file is described in *Register File*.
- *Sleep Unit* describes the Sleep unit including the PULP Cluster extension.
- *Core Versions and RTL Freeze Rules* describes the core versioning.
- *Glossary* provides definitions of used terminology.

2.5 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:

2.5.1 Memory-Protocol

The Instruction and Data memory interfaces are now compliant with the OBI protocol (see 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).

2.5.2 RV32F Extensions

Previously, RI5CY could select with a parameter whether the FPU was instantiated inside the EX stage or via the APU interface. Now in CV32E40P, the FPU is not instantiated in the core EX stage anymore. A new file called cv32e40p_top.sv is instantiating the core together with the FPU and APU interface is not visible on I/Os. This is this new top level which has been used for Verification and Implementation.

2.5.3 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.

2.5.4 CSR Address Re-Mapping

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 now fully compliant with the RISC-V spec on performance counters side. And the custom PULP HWLoop CSRs have been moved from the 0x7C* to RISC-V user custom read-only 0xCC0-0xCFF address space.

2.5.5 Interrupts

RI5CY used to have a req plus a 5 bits 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.

2.5.6 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 constraints for the HWLoop (see *CORE-V Hardware Loop feature*).

2.5.7 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.

2.6 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)

2.7 Contributors

Andreas Traber (atraber@iis.ee.ethz.ch)
Michael Gautschi (gautschi@iis.ee.ethz.ch)
Pasquale Davide Schiavone (pschiavo@iis.ee.ethz.ch)

Arjan Bink (arjan.bink@silabs.com)
Paul Zavalney (paul.zavalney@silabs.com)

Pascal Gouédo (pascal.gouedo@dolphin.fr)

Micrel Lab and Multitherman Lab
University of Bologna, Italy

Integrated Systems Lab
ETH Zürich, Switzerland

CHAPTER

THREE

CORE INTEGRATION

The main module is named `cv32e40p_top` and can be found in `cv32e40p_top.sv`. Below, the instantiation template is given and the parameters and interfaces are described.

Note: `cv32e40p_top` instantiates former `cv32e40p_core` and a wrapped `fpnew_top`. It is highly suggested to use `cv32e40p_top` in place of `cv32e40p_core` as it allows to easily enable/disable FPU parameter with no interface change. As mentioned in *Non-backward compatibility*, v2.0.0 `cv32e40p_core` has **slight** modifications that makes it not backward compatible with v1.0.0 one in some cases. It is worth mentioning that if the core in its v1 version was/is instantiated without parameters setting, there is still backward compatibility as all parameters default value are set to v1 values.

3.1 Instantiation Template

```
cv32e40p_top #(
    .FPU                      ( 0 ),
    .FPU_ADDMUL_LAT           ( 0 ),
    .FPU_OTHERS_LAT           ( 0 ),
    .ZFINX                    ( 0 ),
    .COREV_PULP               ( 0 ),
    .COREV_CLUSTER            ( 0 ),
    .NUM_MHPMCOUNTERS         ( 1 )
) u_core (
    // Clock and reset
    .rst_ni                   (),
    .clk_i                    (),
    .scan_cg_en_i             (),

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

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

(continues on next page)

(continued from previous page)

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

    // Data memory interface
    .data_addr_o              (),
    .data_req_o               (),
    .data_gnt_i               (),
    .data_we_o                (),
    .data_be_o                (),
    .data_wdata_o             (),
    .data_rvalid_i            (),
    .data_rdata_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           ()
);
```

3.2 Parameters

Table 3.1: Parameters

Name	Type/Range	Default	Description
FPU	bit	0	Enable Floating Point Unit (FPU) support, see *Floating Point Unit (FPU)*
FPU_ADDMUL_LAT	int	0	Number of pipeline registers for Floating-Point addition and multiplication instructions, see *Floating Point Unit (FPU)*
FPU_OTHERS_LAT	int	0	Number of pipeline registers for Floating-Point comparison, conversion and classify instructions, see *Floating Point Unit (FPU)*
ZFINX	bit	0	Enable Floating Point instructions to use the General Purpose register file instead of requiring a dedicated Floating Point register file, see *Floating Point Unit (FPU)*. Only allowed to be set to 1 if FPU = 1
COREV_PULP	bit	0	Enable all of the custom PULP ISA extensions (except **cv.elw**) (see *CORE-V Instruction Set Custom Extensions*) and all custom CSRs (see *Control and Status Registers*). Examples of PULP ISA extensions are post-incrementing load and stores (see *Post-Increment Load & Store Instructions and Register-Register Load & Store Instructions*) and hardware loops (see *Hardware Loops*).
COREV_CLUSTER	bit	0	Enable PULP Cluster support (**cv.elw**), see *PULP Cluster Extension*
NUM_MHPMCOUNTERS	int (0..29)	1	Number of MHPMCOUNTER performance counters, see *Performance Counters*

3.3 Interfaces

Table 3.2: Interfaces

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

3.4 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 in both `cv32e40p_sleep_unit.sv` and `cv32e40p_top.sv`.

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.

3.5 Synthesis guidelines

The CV32E40P core is fully synthesizable. It has been designed mainly for ASIC designs, but FPGA synthesis is supported as well.

The top level module is called cv32e40p_top and includes both the core and the FPU. All the core files are in `rtl` and `rtl/include` folders (all synthesizable) while all the FPU files are in `rtl/vendor/pulp_platform_common_cells`, `rtl/vendor/pulp_platform_fpnew` and `rtl/vendor/pulp_platform_fpu_div_sqrt`. .. while all the FPU files are in `rtl/vendor/pulp_platform_common_cells`, `rtl/vendor/pulp_platform_fpnew` and `rtl/vendor/opene906`. cv32e40p_fpu_manifest.flist is listing all the required files.

The user must provide a clock-gating module that instantiates the functionally equivalent clock-gating cell of the target technology. This file must have the same interface and module name as 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.

3.5.1 ASIC Synthesis

ASIC synthesis is supported for CV32E40P. The whole design is completely synchronous and uses positive-edge triggered flip-flops. The core occupies an area of about XX kGE. With the FPU, the area increases to about XX kGE (XX kGE FPU, XX kGE additional register file). A technology specific implementation of a clock gating cell as described in *Clock Gating Cell* needs to be provided.

3.5.2 FPGA Synthesis

FPGA synthesis is supported for CV32E40P and it has been successfully implemented using both AMD® Vivado® and Intel® Quartus® Prime Pro Edition tools.

Due to some advanced System Verilog features used by CV32E40P RTL design, Intel® Quartus® Prime Standard Edition isn't able to parse some CV32E40P System Verilog files.

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

3.5.3 Synthesizing with the FPU

By default the pipeline of the FPU is purely combinatorial (FPU_*_LAT = 0). In this case FPU instructions latency is the same than simple ALU operations (except multicycle FDIV/FSQRT ones). But as FPU operations are much more complex than ALU ones, maximum achievable frequency is much lower than ALU one when FPU is enabled.

If this can be fine for low frequency systems, it is possible to indicate how many pipeline registers are instantiated in the FPU to reach higher target frequency. This is done by adjusting FPU_*_LAT CV32E40P parameters setting to perfectly fit target frequency.

It should be noted that any additional pipeline register is impacting FPU instructions latency and could cause performances degradation depending of applications using Floating-Point operations.

Those pipeline registers are all added at the end of the FPU pipeline with all operators before them. Optimal frequency is only achievable using automatic retiming commands in implementation tools. As an exemple, this can be done for Synopsys® Design Compiler with the following command:

“set_optimize_registers true -designs [get_object_name [get_designs “*cv32e40p_fp_wrapper*”]]”.

CHAPTER

FOUR

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 `cv32e40p_top` top level module to 1. This will extend the CV32E40P decoder accordingly and will instantiate the FPU. The FPU repository used by the CV32E40P is available at https://github.com/openhwgroup/cvfpu and its documentation can be found here. CVFPU v0.8.1 release has been copied in CV32E40P repository inside rtl/vendor (used for verification and implementation) so all core and FPU RTL files should be taken from CV32E40P repository.

cv32e40p_fpu_manifest file is listing all necessary files for both the Core and CVFPU.

4.1 CVFPU parameters

As CVFPU is an highly configurable IP, here is the list of its parameters and their actual value used when CVFPU is intantiated through a wrapper in `cv32e40p_top` module.

Table 4.1: CVFPU Features parameter

Name	Type/Range	Value	Description
`Width`	int	32	**Datapath Width** Specifies the width of the input and output data ports and of the datapath.
`EnableVectors`	logic	0	**Vectorial Hardware Generation** Controls the generation of packed-SIMD computation units.
`EnableNanBox`	logic	0	**NaN-Boxing Check Control** Controls whether input value NaN-boxing is enforced.
`FpFmtMask`	fmt_logic_t	{1, 0, 0, 0, 0}	**Enabled Floating-Point Formats** Enables respectively: IEEE Single-Precision format IEEE Double-Precision format IEEE Half-Precision format Custom Byte-Precision format Custom Alternate Half-Precision format
`IntFmtMask`	ifmt_logic_t	{0, 0, 1, 0}	**Enabled Integer Formats** Enables respectively: Byte format Half-Word format Word format Double-Word format

Table 4.2: CVFPU Implementation parameter

Name	Type/Range	Value	Description
PipeRegs	op-grp_fmt_unsigned_t	{ {FPU_ADDMUL_LAT, 0, 0, 0, 0}, {default: 1}, {default: FPU_OTHERS_LAT}, {default: FPU_OTHERS_LAT} }	**Number of Pipelining Stages** This parameter sets a number of pipeline stages to be inserted into the computational units per operation group, per FP format. As such, latencies for different operations and different formats can be freely configured. Respectively: ADDition/MULtiplication operation group DIVision/SQuare RooT operation group NON COMPuting operation group CONVersion operation group FPU_ADDMUL_LAT and FPU_OTHERS_LAT are cv32e40p_top parameters.
UnitTypes	op-grp_fmt_unit_types_t	{ {default: MERGED}, {default: MERGED}, {default: PARALLEL}, {default: MERGED} }	**HW Unit Implementation** This parameter allows to control resources by either removing operation units for certain formats and operations, or merging multiple formats into one. Respectively: ADDition/MULtiplication operation group DIVision/SQuare RooT operation group NON COMPuting operation group CONVersion operation group
PipeConfig	pipe_config_t	AFTER	**Pipeline Register Placement** This parameter controls where pipeling registers (number defined by PipeRegs) are placed in each operational unit. AFTER means they are all placed at the output of each operational unit. See *Synthesizing with the FPU* advices to get best synthesis results.

Table 4.3: Other CVFPU parameters

Name	Type/Range	Value	Description
TagType		logic	The SystemVerilog data type of the operation tag input and output ports.
TrueSIMDClass	int	0	Vectorial mode classify operation RISC-V compliancy.
EnableSIMDMask	int	0	Inactive vectorial lanes floating-point status flags masking.

4.2 FP Register File

By default a dedicated register file consisting of 32 floating-point registers, `f0-f31`, is instantiated. This default behavior can be overruled by setting the parameter **ZFINX** of the `cv32e40p_top` top level module 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 explained in *Cycle counts per instruction type* table.

To allow FPU unit to be put in sleep mode at the same time the core is doing so, a clock gating cell is instantiated in `cv32e40p_top` top level module as well with its enable signal being inverted `core_sleep_o` core output.

4.3 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.

4.4 Reminder for programmers

As mentioned in RISC-V Privileged Architecture specification, `mstatus`.FS should be set to Initial to be able to use FP instructions. If `mstatus`.FS = Off (reset value), any instruction that attempts to read or write the Floating-Point state (F registers or F CSRs) will cause an illegal instruction exception.

Upon interrupt or context switch events, `mstatus`.SD should be read to see if Floating-Point state has been altered. If following executed program (interrupt routine or whatsover) is going to use FP instructions and only if `mstatus`.SD = 1 (means FS = Dirty), then the whole FP state (F registers and F CSRs) should be saved in memory and program should set `mstatus`.FS to Clean. When returning to interrupted or main program, if `mstatus`.FS = Clean then the whole FP state should be restored from memory.

CHAPTER

FIVE

VERIFICATION

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

5.1 v1.0.0 verification

In early 2021 the CV32E40P achieved Functional RTL Freeze (released with cv32e40p_v1.0.0 version), meaning that is has been fully verified as per its Verification Plan. Final functional, code and test coverage reports can be found here.

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[1].

A breakdown of the RTL issues is as follows:

Table 5.1: How RTL Issues Were Found in v1.0.0

"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 5.2: Breakdown of Issues found by Simulation in v1.0.0

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:

[1] 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.

Table 5.3: Issue Classification in v1.0.0

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.

5.2 v2.0.0 verification

The table below lists the 7 configurations with `cv32e40p_top` parameters values verified in the scope of CV32E40Pv2 project using both Formal-based and Simulation-based methodologies.

Table 5.4: Verified configurations

Verified Configurations (CFG_"config name")							
Top Parameters	**P**	**P_F0**	**P_F1**	**P_F2**	**P_Z0**	**P_Z1**	**P_Z2**
COREV_PULP	1	1	1	1	1	1	1
COREV_CLUSTER	0	0	0	0	0	0	0
FPU	0	1	1	1	1	1	1
ZFINX	0	0	0	0	1	1	1
FPU_ADDMUL_LAT	0	0	1	2	0	1	2
FPU_OTHERS_LAT	0	0	1	2	0	1	2

Verification environment is described in CORE-V Verification Strategy and used the so-called Step-and-Compare 2.0 methodology. It is using an Imperas® model connected in the test-bench through an RVVI interface as shown by following figure:

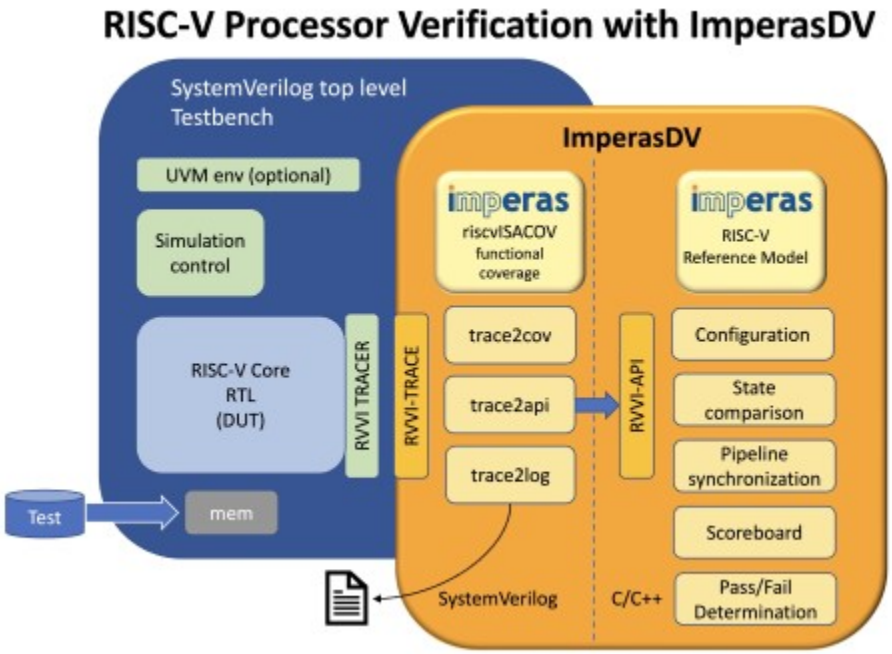

Figure 5.1: ImperasDV framework

CV32E40Pv2 achieved RTL Freeze (released with cv32e40p_v2.0.0 version) mid-April 2024, meaning that is has been fully verified as per its Verification Plan. Final functional, code and test coverage reports can be found here: CV32E40P v2.0.0 Report.

It is to be mentioned that CV32E40Pv2 has successfully executed RISCOF (RISC-V COmpatibility Framework) for RV32IMCF extensions. Report can be found here.

5.2.1 Formal verification

To accelerate the verification of more than 300 XPULP instructions, Formal Verification methodology has been used with Siemens EDA Onespin tool and its RISC-V ISA Processor Verification app.

The XPULP instructions pseudo-code description using Sail language have been added to the RISC-V ISA app to successfully formally verify all the CV32E40P instructions, including the previously verified standard IMC together with the new F, Zfinx and XPULP extensions and all additional custom CSRs.

Example:

```
{
  "name": "CV.SDOTUP.B",
  "disassembly": "cv.sdotup.b {rd},{rs1},{rs2}",
  "decoding": "1001100 rs2 rs1 001 rd/rs3 1111011",
  "restrictions": "",
  "execution": "X(rd) = X(rs3) + EXTZ(mul(X(rs1)[7..0],X(rs2)[7..0])) +
                                 EXTZ(mul(X(rs1)[15..8],X(rs2)[15..8])) +
                                 EXTZ(mul(X(rs1)[23..16],X(rs2)[23..16])) +
                                 EXTZ(mul(X(rs1)[31..24],X(rs2)[31..24]))"
},
```

Those SAIL instructions description are then used to automatically generate more than 430 assertions and 29 CSRs descriptions. Those assertions have been applied on the 7 different configurations listed in *Verified configurations* table.

RTL code coverage is generated using Siemens EDA Onespin Quantify tool which uses RTL mutation to check assertions quality and can produce standard UCDB database that can be merged with simulation one afterwards.

5.2.2 Simulation verification

core-v-verif verification environment for v1.0.0 was using a *step&compare* methodology with an instruction set simulator (ISS) from Imperas Software as the reference model. This strategy was successful, but inefficient because the *step&compare* logic in the testbench must compensate for the cycle-time effects of events that are asynchronous to the instruction stream such as interrupts, debug resets plus bus errors and random delays on instruction fetch and load/store memory buses. For verification of v2.0.0 release of the CV32E40P core, the step-and-compare and the ISS have been replaced by a true reference model (RM) called ImperasDV. In addition, the Imperas Reference Model has been extended to support the v2 XPULP instructions specification.

Another innovation for v2.0.0 was the adoption of a standardized tracer interface to the DUT and RM, based on the open-source RISC-V Verification Interface (RVVI). The use of well documented, standardized interfaces greatly simplifies the integration of the DUT with the RM.

5.2.3 Results summary

30 issues were identified by Formal Verification and 10 by Simulation methodologies, all have been resolved.

Here is the breakdown of all the issues:

Table 5.5: How Issues Were Found in v2.0.0

"Found By"	Count	Note
Formal Verification	30	All related to features enabled by `COREV_PULP` or `FPU`.
Simulation	10	
Lint	2	

A classification of the Formal Verification issues by type and their description are listed in the following tables:

Table 5.6: Breakdown of Issues found by Formal Verification in v2.0.0

Type	Count	Note
User Manual	12	Instructions description leading to mis-interpretation
RTL bugs	18	See classification below

Table 5.7: Formal Verification Issues Classification in v2.0.0

Issue Type	Count	Note
Illegal instructions exception	5	F and XPULP instructions corner cases or CSR accesses not flagged as Illegal instructions exception.
Multi-cycle F instructions	8	FDIV, FSQRT or respective F instructions (when FPU_ADDMUL_LAT or FPU_OTHERS_LAT = 2) are executed in the background and the pipeline can continue to execute other instructions as long as there is no Read-After-Write or Write-After-Write dependency. When the multi-cycle F instructions are finally writing back their result in the Register File, this register update can corrupt on-going instructions behaviour or result. This is the case for Misaligned Loads, Post-Incremented Load/Stores, MULH, JALR or cv.add*NR/cv.sub*NR.
F instructions result or flags	5	F result or flags computations is incorrect with respect to IEEE 754-2008 standard.

A classification of the Simulation issues by type and their description are listed in the following tables:

Table 5.8: Breakdown of Issues found by Simulation in v2.0.0

Type	Count	Note
RTL bugs	10	See classification below

Table 5.9: Simulation Issues Classification in v2.0.0

Issue Type	Count	Note
Multi-cycle F instructions	5	Data forward violation between XPULP instructions and muticycle F instructions.
Hardware Loops	3	Incorrect behavior when count programmed with 0 value. lpendX CSR updated by a cancelled instruction. lpcountX not updated after a pipeline flush due to a CSR access.
Deadlock	1	Bug resolution for multicycle F instructions created a deadlock when conflicting Register File write between ALU and FPU.
MSTATUS.FS incorrect value	1	FS was not updated following any Floating Point Load instruction.

5.3 Tracer

The module `cv32e40p_rvfi_trace` 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_top`. It can be enabled during simulation by defining **CV32E40P_RVFI_TRACE_EXECUTION**.

5.3.1 Output file

All traced instructions are written to a log file. The log file is named `trace_core.log`.

5.3.2 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 physical address (PA) of the loaded or stored data is reported as well.

```
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=0x00007fff  x8:0x00008000
 136    64 0000015a 8c65     c.and   x8,x9       x8=0x00000000  x8:0x00007fff ␣
↪x9:0x00000000
 142    67 0000015c c622     c.swsp  x8,12(x2)   x2:0x00002000  x8:0x00000000␣
↪PA:0x0000200c
```

CHAPTER

SIX

CORE-V HARDWARE LOOP FEATURE

To increase the efficiency of small loops, CV32E40P supports hardware loops (HWLoop). They can be enabled by setting the `COREV_PULP` parameter. Hardware loops make executing a piece of code multiple times possible, without the overhead of branches penalty 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 just after the last one executed by the loop) and a counter that is decremented every time the last instruction of 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 and loop 1 is the outer loop.

6.1 Hardware Loop constraints

Following constraints must be respected by any toolchain compiler or by hand-written assembly code. `Violation of these constraints will not generate any hardware exception` and behaviour is undefined.

In order to catch **as early as possible** those software exceptions when executing a program either on a verification Reference Model or on a virtual platform Instruction Set Simulator, `those model/simulation platforms should generate an error` with a meaningfull message related to Hardware Loops constraints violation. Those constraint checks could be done only for each instruction in the hardware loop body, meaning when (lpstartX <= PC <= lpendX - 4) and (lpcountX > 0).

The HWLoop constraints are:

- HWLoop starti, endi, setupi and setup instructions addresses must be 32-bit aligned (PC-related instructions).
- Start and End addresses of an HWLoop body must be 32-bit aligned.
- End Address must be strictly greater than Start Address.
- HWLoop #0 (resp. #1) start and end addresses **must not be modified** if HWLoop #0 (resp. #1) count is different than 0.
- End address of an HWLoop must point to the instruction just after the last one of the HWLoop body.
- HWLoop body must contain at least 3 instructions.
- When both loops are nested, at least 1 instruction should be present between last innermost HWLoop (must be #0) instruction and last outermost HWLoop (must be #1) instruction. In other words the End address of the outermost HWLoop must be at least 8 bytes further than the End address of the innermost HWLoop (HWLoop[1].endaddress >= HWLoop[0].endaddress + 8).

In the example below the first "addi %[j], %[j], 2;" instruction is the one added due to this constraint. The code could have been simpler by using only one "addi %[j], %[j], 4;" instruction but to respect this constraint it has been split in two instructions.

- HWLoop must always be entered from its start location (no branch/jump to a location inside a HWLoop body).
- No HWLoop #0 (resp. #1) CSR should be modified inside the HWLoop #0 (resp. #1) body.
- No Compressed instructions (RVC) allowed in the HWLoop body.
- No jump or branch instructions allowed in the HWLoop body.
- No memory ordering instructions (fence, fence.i) allowed in the HWLoop body.
- No privileged instructions (mret, dret, wfi) allowed in the HWLoop body, except for ebreak and ecall.

The rationale of NOT generating any hardware exception when violating any of those constraints is that it would add resources (32-bit adders and substractors needed for the third and fourth rules) which are costly in area and power consumption. These additional (and costly) resources would be present just to catch situations that should never happen. This in an architectural choice in order to keep CV32E40P area and power consumption to its lowest level.

The rationale of putting the end-of-loop label to the first instruction after the last one of the loop body is that it greatly simplifies compiler optimization (relative to basic blocks management).

In order to use hardware loops, the compiler needs to setup the loops beforehand with cv.start/i, cv.end/i, cv.count/i or cv.setup/i instructions. The compiler will use HWLoop automatically whenever possible without the need of assembly.

For debugging, interrupts and context switches, the hardware loop registers are mapped into the CSR custom read-only address space. To read them csrr instructions should be used and to write them register flavour of hardware loop instructions should be used. Using csrw instructions to write hardware loop registers will generate an illegal instruction exception. The CSR HWLoop registers are described in the *Control and Status Registers* section.

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

```
asm volatile (
    "add %[i],x0, x0;"
    "add %[j],x0, x0;"
    ".balign 4;"
    "cv.starti 1, start1;"
    "cv.endi   1, end1;"
    "cv.count  1, %[N];"
    "any instructions here"
    ".balign 4;"
    "cv.starti 0, start0;"
    "cv.endi   0, end0;"
    "any instructions here"
    ".balign 4;"
    ".option norvc;"
    "start1:;"
    "    cv.count 0, %[N];"
    "    start0:;"
    "        addi %[i], %[i], 1;"
    "        addi %[i], %[i], 1;"
    "        addi %[i], %[i], 1;"
    "    end0:;"
    "    addi %[j], %[j], 2;"
    "    addi %[j], %[j], 2;"
    "end1:;"
    : [i] "+r" (i), [j] "+r" (j)
```

(continues on next page)

(continued from previous page)

```
    : [N] "r" (10)
);
```

As HWLoop feature is enabled as soon as lpcountX > 0, lpstartX and lpendX **must** be programmed **before** lpcountX to avoid unexpected behavior. For HWLoop where body contains up to 30 instructions, it is always better to use cv.setup* instructions which are updating all 3 HWLoop CSRs in the same cycle.

At the beginning of the HWLoop, the registers %[i] and %[j] are 0. The innermost loop, from start0 to (end0 - 4), adds to %[i] three times 1 and it is executed 10x10 times. Whereas the outermost loop, from start1 to (end1 - 4), 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.

6.2 Hardware loops impact on application, exceptions handlers and debugger

6.2.1 Application and ebreak/ecall exception handlers

When an ebreak or an ecall instruction is used in an application, special care should be given for those instruction handlers in case they are placed as the last instruction of an HWLoop. Those handlers should manage MEPC and lpcountX CSRs updates because an hw loop early-exit could happen if not done.

At the end of the handlers after restoring the context/CSRs, a piece of smart code should be added with following highest to lowest order of priority:

1. if MEPC = lpend0 - 4 and lpcount0 > 1 then MEPC should be set to lpstart0 and lpcount0 should be decremented by 1,
2. else if MEPC = lpend0 - 4 and lpcount0 = 1 then MEPC should be incremented by 4 and lpcount0 should be decremented by 1,
3. else if MEPC = lpend1 - 4 and lpcount1 > 1 then MEPC should be set to lpstart1 and lpcount1 should be decremented by 1,
4. else if MEPC = lpend1 - 4 and lpcount1 = 1 then MEPC should be incremented by 4 and lpcount1 should be decremented by 1,
5. else if (lpstart0 <= MEPC < lpend0 - 4) or (lpstart1 <= MEPC < lpend1 - 4) then MEPC should be incremented by 4,
6. else if instruction at MEPC location is either ecall or ebreak then MEPC should be incremented by 4,
7. else if instruction at MEPC location location is c.ebreak then MEPC should be incremented by 2.

The 2 last cases are the standard ones when ebreak/ecall are not inside an HWLopp.

6.2.2 Interrupt handlers

When an interrupt is happening on the last HWLoop instruction, its execution is cancelled, its address is saved in MEPC and its execution will be resumed when returning from interrupt handler. There is nothing special to be done in those interrupt handlers with respect to MEPC and lpcountX updates, they will be correctly managed by design when executing this last HWLoop instruction after interrupt handler execution.

Moreover since hardware loop could be used in interrupt routine, the registers have to be saved (resp. restored) at the beginning (resp. end) of the interrupt routine together with the general purpose registers.

6.2.3 Illegal instruction exception handler

Depending if an application is going to resume or not after Illegal instruction exception handler, same MEPC/HWLoops CSRs management than ebreak/ecall could be necessary.

6.2.4 Debugger

If ebreak is used to enter in Debug Mode (*Scenario 2 : Enter Debug Mode*) and put at the last instruction location of an HWLoop (not very likely to happen), same management than above should be done but on DPC rather than on MEPC.

When ebreak instruction is used as Software Breakpoint by a debugger when in debug mode and is placed at the last instruction location of an HWLoop in instruction memory, no special management is foreseen. When executing the Software Breakpoint/ebreak instruction, control is given back to the debugger which will manage the different cases. For instance in Single-Step case, original instruction is put back in instruction memory, a Single-Step command is executed on this last instruction (with desgin updating PC and lpcountX to correct values) and Software Breakpoint/ebreak is put back by the debugger in memory.

When ecall instruction is used by a debugger to execute System Calls and is placed at the last instruction location of an HWLoop in instruction memory, debugger ecall handler in debug rom should do the same than described above for application case.

CHAPTER

SEVEN

CORE-V INSTRUCTION SET CUSTOM EXTENSIONS

CV32E40P supports the following CORE-V ISA X Custom Extensions, which can be enabled by setting `COREV_PULP == 1`.

- Post-Increment load and stores, see *Post-Increment Load & Store Instructions and Register-Register Load & Store Instructions*, invoked in the tool chain with `-march=rv32i*_xcvmem`.
- Hardware Loop extension, see *Hardware Loops*, invoked in the tool chain with `-march=rv32i*_xcvhwlp`.
- ALU extensions, see *ALU*, which are divided into three sub-extensions:
  - bit manipulation instructions, invoked in the tool chain with `-march=rv32i*_xcvbitmanip`;
  - miscellaneous ALU instructions, invoked in the tool chain with `-march=rv32i*_xcvalu`; and
  - immediate branch instructions, invoked in the tool chain with `-march=rv32i*_xcvbi`.
- Multiply-Accumulate extensions, see *Multiply-Accumulate*, invoked in the tool chain with `-march=rv32i*_xcvmac`.
- Single Instruction Multiple Data (aka SIMD) extensions, see *SIMD*, invoked in the tool chain with `-march=rv32i*_xcvsimd`.

Additionally the event load instruction (**cv.elw**) is supported by setting `COREV_CLUSTER == 1`, see *Event Load Instruction*. This is a separate ISA extension, invoked in the tool chain with `-march=rv32i*_xcvelw`.

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

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

Note: Clang/LLVM assembler will be supported by 30 June 2023, with builtin function support by 31 December 2023.

7.1 Pseudo-instructions

This specification also includes documentation of some CORE-V pseudo-instructions. Pseudo-instructions are implemented in the assembler that are similar to a base instruction but provides control information to the assembler as opposed to generating its base instruction. This makes it easier to program as we gain clarity on the intention of the programmer.

- 16-Bit x 16-Bit Multiplication pseudo-instructions, see *16-Bit x 16-Bit Multiplication pseudo-instructions*.

7.2 Post-Increment Load & Store Instructions and Register-Register Load & Store Instructions

Post-Increment 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-increment load & store instructions and register-register load & store instructions are only supported if `COREV_PULP` == 1.

7.2.1 Load operations

Note: When same register is used as address and destination (rD == rs1) for post-incremented loads, loaded data has highest priority over incremented address when writing to this same register.

Table 7.1: Load operations

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

7.2.2 Store operations

Table 7.2: Store operations

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

7.2.3 Encoding

Table 7.3: Post-Increment Register-Immediate Load operations encoding

31 : 20 **imm[11:0]**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 7 **rD**	6 : 0 **opcode**	**Mnemonic**
offset	base	000	dest	000 1011	**cv.lb rD, (rs1), Imm**
offset	base	100	dest	000 1011	**cv.lbu rD, (rs1), Imm**
offset	base	001	dest	000 1011	**cv.lh rD, (rs1), Imm**
offset	base	101	dest	000 1011	**cv.lhu rD, (rs1), Imm**
offset	base	010	dest	000 1011	**cv.lw rD, (rs1), Imm**

Table 7.4: Post-Increment Register-Register Load operations encoding

31 : 25 **funct7**	24 : 20 **rs2**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 7 **rD**	6 : 0 **opcode**	**Mnemonic**
000 0000	offset	base	011	dest	010 1011	**cv.lb rD, (rs1), rs2**
000 1000	offset	base	011	dest	010 1011	**cv.lbu rD, (rs1), rs2**
000 0001	offset	base	011	dest	010 1011	**cv.lh rD, (rs1), rs2**
000 1001	offset	base	011	dest	010 1011	**cv.lhu rD, (rs1), rs2**
000 0010	offset	base	011	dest	010 1011	**cv.lw rD, (rs1), rs2**

Table 7.5: Register-Register Load operations encoding

31 : 25 **funct7**	24 : 20 **rs2**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 7 **rD**	6 : 0 **opcode**	**Mnemonic**
000 0100	offset	base	011	dest	010 1011	**cv.lb rD, rs2(rs1)**
000 1100	offset	base	011	dest	010 1011	**cv.lbu rD, rs2(rs1)**
000 0101	offset	base	011	dest	010 1011	**cv.lh rD, rs2(rs1)**
000 1101	offset	base	011	dest	010 1011	**cv.lhu rD, rs2(rs1)**
000 0110	offset	base	011	dest	010 1011	**cv.lw rD, rs2(rs1)**

Table 7.6: Post-Increment Register-Immediate Store operations encoding

31 : 25 **imm[11:5]**	24 : 20 **rs2**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 7 **imm[4:0]**	6 : 0 **opcode**	**Mnemonic**
offset[11:5]	src	base	000	offset[4:0]	010 1011	**cv.sb rs2, (rs1), Imm**
offset[11:5]	src	base	001	offset[4:0]	010 1011	**cv.sh rs2, (rs1), Imm**
offset[11:5]	src	base	010	offset[4:0]	010 1011	**cv.sw rs2, (rs1), Imm**

Table 7.7: Post-Increment Register-Register Store operations encoding

31 : 25 **funct7**	24 : 20 **rs2**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 7 **rs3**	6 : 0 **opcode**	**Mnemonic**
001 0000	src	base	011	offset	010 1011	**cv.sb rs2, (rs1), rs3**
001 0001	src	base	011	offset	010 1011	**cv.sh rs2, (rs1), rs3**
001 0010	src	base	011	offse t	010 1011	**cv.sw rs2, (rs1), rs3**

Table 7.8: Register-Register Store operations encoding

31 : 25 **funct7**	24 : 20 **rs2**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 7 **rs3**	6 : 0 **opcode**	**Mnemonic**
001 0100	src	base	011	offset	010 1011	**cv.sb rs2, rs3(rs1)**
001 0101	src	base	011	offset	010 1011	**cv.sh rs2, rs3(rs1)**
001 0110	src	base	011	offset	010 1011	**cv.sw rs2, rs3(rs1)**

7.3 Event Load Instruction

The event load instruction **cv.elw** is only supported if the `COREV_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*.

7.3.1 Event Load operation

Table 7.9: Event Load operation

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

7.3.2 Encoding

Table 7.10: Event Load operation encoding

31 : 20 **imm[11:0]**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 7 **rD**	6 : 0 **opcode**	**Mnemonic**
offset	base	011	dest	000 1011	**cv.elw rD, Imm(rs1)**

7.4 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.

Due to start/end addresses constraint, the 2 LSBs are hardwired to 0. When using cv.start and cv.end instructions, the 2 LSBs of rs1 are ignored.

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

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

In the following tables, the hardware loop instructions are reported. In assembly, **L** is referred by 0 or 1.

7.4.1 Hardware Loops operations

Table 7.11: Long Hardware Loop Setup operations

Mnemonic	Description
cv.starti L, uimmL	lpstart[L] = PC + (uimmL << 2)
cv.start L, rs1	lpstart[L] = rs1
cv.endi L, uimmL	lpend[L] = PC + (uimmL << 2)
cv.end L, rs1	lpend[L] = rs1
cv.counti L, uimmL	lpcount[L] = uimmL
cv.count L, rs1	lpcount[L] = rs1

Table 7.12: Short Hardware Loop Setup operations

Mnemonic	Description
cv.setupi L, uimmL, uimmS	lpstart[L] = PC + 4 lpend[L] = PC + (uimmS << 2) lpcount[L] = uimmL
cv.setup L, rs1, uimmL	lpstart[L] = PC + 4 lpend[L] = PC + (uimmL << 2) lpcount[L] = rs1

7.4.2 Encoding

Table 7.13: Hardware Loops operations encoding

31 : 20 **uimmL[11:0]**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 8 **funct4**	7 **L**	6 : 0 **opcode**	**Mnemonic**
uimmL[11:0]	00000	100	0000	L	010 1011	**cv.starti L, uimmL**
0000 0000 0000	src1	100	0001	L	010 1011	**cv.start L, rs1**
uimmL[11:0]	00000	100	0010	L	010 1011	**cv.endi L, uimmL**
0000 0000 0000	src1	100	0011	L	010 1011	**cv.end L, rs1**
uimmL[11:0]	00000	100	0100	L	010 1011	**cv.counti L, uimmL**
0000 0000 0000	src1	100	0101	L	010 1011	**cv.count L, rs1**
uimmL[11:0]	uimmS[4:0]	100	0110	L	010 1011	**cv.setupi L, uimmL, uimmS**
uimmL[11:0]	src1	100	0111	L	010 1011	**cv.setup L, rs1, uimmL**

7.5 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 `COREV_PULP` == 1.

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

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

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

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

- Set Is3+1 or rs2[9:5]+1 bits at position Is2 or rs2[4:0]

7.5.1 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
. . . . . . . . . . . . . . . . . . . . . . . . H G F E D C B A
0 0 0 0 1 1 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 1 0 1 0 0 1 0 0 1 1 0
```

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

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

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

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

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

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

in:    0xC64A5933 11000110010010100101100100110011
shift: 0x64A59330 01100100101001011001001100110000
```

(continues on next page)

(continued from previous page)

```
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.

7.5.2 Bit Manipulation operations

Table 7.14: Bit Manipulation operations

Mnemonic	Description
cv.extract rD, rs1, Is3, Is2	rD = Sext(rs1[min(Is3+Is2,31):Is2]) Note: Sign extension is done over the MSB of the extracted part.
cv.extractu rD, rs1, Is3, Is2	rD = Zext(rs1[min(Is3+Is2,31):Is2])
cv.extractr rD, rs1, rs2	rD = Sext(rs1[min(rs2[9:5]+rs2[4:0],31):rs2[4:0]]) Note: Sign extension is done over the MSB of the extracted part.
cv.extractur rD, rs1, rs2	rD = Zext(rs1[min(rs2[9:5]+rs2[4:0],31):rs2[4:0]])
cv.insert rD, rs1, Is3, Is2	rD[min(Is3+Is2,31):Is2] = rs1[Is3-(max(Is3+Is2,31)-31):0] The rest of the bits of rD are untouched and keep their previous value. Is3 + Is2 must be < 32.
cv.insertr rD, rs1, rs2	rD[min(rs2[9:5]+rs2[4:0],31):rs2[4:0]] = rs1[rs2[9:5]-(max(rs2[9:5]+rs2[4:0],31)-31):0] The rest of the bits of rD are untouched and keep their previous value. Is3 + Is2 must be < 32.
cv.bclr rD, rs1, Is3, Is2	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.
cv.bclrr rD, rs1, rs2	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.
cv.bset rD, rs1, Is3, Is2	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.
cv.bsetr rD, rs1, rs2	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.
cv.ff1 rD, rs1	rD = bit position of the first bit set in rs1, starting from LSB. If bit 0 is set, rD will be 0. If only bit 31 is set, rD will be 31. If rs1 is 0, rD will be 32.
cv.fl1 rD, rs1	rD = bit position of the last bit set in rs1, starting from MSB. If bit 31 is set, rD will be 31. If only bit 0 is set, rD will be 0. If rs1 is 0, rD will be 32.
cv.clb rD, rs1	rD = count leading bits of rs1 Number of consecutive 1's or 0's starting from MSB. If rs1 is 0, rD will be 0. If rs1 is different than 0, returns (number - 1).
cv.cnt rD, rs1	rD = Population count of rs1 Number of bits set in rs1.
cv.ror rD, rs1, rs2	rD = RotateRight(rs1, rs2)
cv.bitrev rD, rs1, Is3, Is2	Given an input rs1 it returns a bit reversed representation assuming FFT on 2^Is2 points in Radix 2^(Is3+1). Is3 can be either 0 (radix-2), 1 (radix-4) or 2 (radix-8). Note: When Is3 = 3, instruction has the same bahavior as if it was 0 (radix-2).

7.5.3 Bit Manipulation Encoding

Table 7.15: Immediate Bit Manipulation operations encoding

31:30	29 : 25	24 : 20	19 : 15	14 : 12	11 : 7	6 : 0	
f2	**Is3[4:0]**	**Is2[4:0]**	**rs1**	**funct3**	**rD**	**opcode**	**Mnemonic**
00	Luimm5[4:0]	Luimm5[4:0]	src	000	dest	101 1011	**cv.extract rD, rs1, Is3, Is2**
01	Luimm5[4:0]	Luimm5[4:0]	src	000	dest	101 1011	**cv.extractu rD, rs1, Is3, Is2**
10	Luimm5[4:0]	Luimm5[4:0]	src	000	dest	101 1011	**cv.insert rD, rs1, Is3, Is2**
00	Luimm5[4:0]	Luimm5[4:0]	src	001	dest	101 1011	**cv.bclr rD, rs1, Is3, Is2**
01	Luimm5[4:0]	Luimm5[4:0]	src	001	dest	101 1011	**cv.bset rD, rs1, Is3, Is2**
11	000, Luimm2[1:0]	Luimm5[4:0]	src	001	dest	101 1011	**cv.bitrev rD, rs1, Is3, Is2**

Table 7.16: Register Bit Manipulation operations encoding

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	**cv.extractr rD, rs1, rs2**
001 1001	src2	src1	011	dest	010 1011	**cv.extractur rD, rs1, rs2**
001 1010	src2	src1	011	dest	010 1011	**cv.insertr rD, rs1, rs2**
001 1100	src2	src1	011	dest	010 1011	**cv.bclrr rD, rs1, rs2**
001 1101	src2	scr1	011	dest	010 1011	**cv.bsetr rD, rs1, rs2**
010 0000	src2	src1	011	dest	010 1011	**cv.ror rD, rs1, rs2**
010 0001	00000	src1	011	dest	010 1011	**cv.ff1 rD, rs1**
010 0010	00000	src1	011	dest	010 1011	**cv.fl1 rD, rs1**
010 0011	00000	src1	011	dest	010 1011	**cv.clb rD, rs1**
010 0100	00000	src1	011	dest	010 1011	**cv.cnt rD, rs1**

7.5.4 General ALU operations

Table 7.17: General ALU operations

Mnemonic	Description
cv.abs rD, rs1	rD = rs1 < 0 ? -rs1 : rs1
cv.sle rD, rs1, rs2	rD = rs1 <= rs2 ? 1 : 0 Note: Comparison is signed.
cv.sleu rD, rs1, rs2	rD = rs1 <= rs2 ? 1 : 0 Note: Comparison is unsigned.
cv.min rD, rs1, rs2	rD = rs1 < rs2 ? rs1 : rs2 Note: Comparison is signed.
cv.minu rD, rs1, rs2	rD = rs1 < rs2 ? rs1 : rs2 Note: Comparison is unsigned.
cv.max rD, rs1, rs2	rD = rs1 < rs2 ? rs2 : rs1 Note: Comparison is signed.

continues on next page

Table 7.17 – continued from previous page

Mnemonic	Description
cv.maxu rD, rs1, rs2	rD = rs1 < rs2 ? rs2 : rs1 Note: Comparison is unsigned.
cv.exths rD, rs1	rD = Sext(rs1[15:0])
cv.exthz rD, rs1	rD = Zext(rs1[15:0])
cv.extbs rD, rs1	rD = Sext(rs1[7:0])
cv.extbz rD, rs1	rD = Zext(rs1[7:0])
cv.clip rD, rs1, Is2	if rs1 <= -2^(Is2-1), rD = -2^(Is2-1), else if rs1 >= 2^(Is2-1)-1, rD = 2^(Is2-1)-1, else rD = rs1 Note: If 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 <= 0, rD = 0, else if rs1 >= 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	rs2' = rs2 & 0x7FFFFFFF if rs1 <= -(rs2'+1), rD = -(rs2'+1), else if rs1 >=rs2', rD = rs2', else rD = rs1
cv.clipur rD, rs1, rs2	rs2' = rs2 & 0x7FFFFFFF if rs1 <= 0, rD = 0, else if rs1 >= rs2', rD = rs2', else rD = rs1
cv.addN rD, rs1, rs2, Is3	rD = (rs1 + rs2) >>> Is3 Note: Arithmetic shift right. Setting Is3 to 1 replaces former cv.avg.
cv.adduN rD, rs1, rs2, Is3	rD = (rs1 + rs2) >> Is3 Note: Logical shift right. Setting Is3 to 1 replaces former cv.avgu.
cv.addRN rD, rs1, rs2, Is3	rD = (rs1 + rs2 + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right. If Is3 is equal to 0, 2^(Is3-1) is equivalent to 0.
cv.adduRN rD, rs1, rs2, Is3	rD = (rs1 + rs2 + 2^(Is3-1))) >> Is3 Note: Logical shift right. If Is3 is equal to 0, 2^(Is3-1) is equivalent to 0.
cv.subN rD, rs1, rs2, Is3	rD = (rs1 - rs2) >>> Is3 Note: Arithmetic shift right.
cv.subuN rD, rs1, rs2, Is3	rD = (rs1 - rs2) >> Is3 Note: Logical shift right.
cv.subRN rD, rs1, rs2, Is3	rD = (rs1 - rs2 + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right. If Is3 is equal to 0, 2^(Is3-1) is equivalent to 0.
cv.subuRN rD, rs1, rs2, Is3	rD = (rs1 - rs2 + 2^(Is3-1))) >> Is3 Note: Logical shift right. If Is3 is equal to 0, 2^(Is3-1) is equivalent to 0.
cv.addNr rD, rs1, rs2	rD = (rD + rs1) >>> rs2[4:0] Note: Arithmetic shift right.
cv.adduNr rD, rs1, rs2	rD = (rD + rs1) >> rs2[4:0] Note: Logical shift right.

continues on next page

Table 7.17 – continued from previous page

Mnemonic	Description
cv.addRNr rD, rs1, rs2	rD = (rD + rs1 + 2^(rs2[4:0]-1)) >>> rs2[4:0] Note: Arithmetic shift right. If rs2[4:0] is equal to 0, 2^(rs2[4:0]-1) is equivalent to 0.
cv.adduRNr rD, rs1, rs2	rD = (rD + rs1 + 2^(rs2[4:0]-1))) >> rs2[4:0] Note: Logical shift right. If rs2[4:0] is equal to 0, 2^(rs2[4:0]-1) is equivalent to 0.
cv.subNr rD, rs1, rs2	rD = (rD - rs1) >>> rs2[4:0] Note: Arithmetic shift right.
cv.subuNr rD, rs1, rs2	rD = (rD - rs1) >> rs2[4:0] Note: Logical shift right.
cv.subRNr rD, rs1, rs2	rD = (rD - rs1+ 2^(rs2[4:0]-1)) >>> rs2[4:0] Note: Arithmetic shift right. If rs2[4:0] is equal to 0, 2^(rs2[4:0]-1) is equivalent to 0.
cv.subuRNr rD, rs1, rs2	rD = (rD - rs1+ 2^(rs2[4:0]-1))) >> rs2[4:0] Note: Logical shift right. If rs2[4:0] is equal to 0, 2^(rs2[4:0]-1) is equivalent to 0.

7.5.5 General ALU Encoding

Table 7.18: General ALU operations encoding

31 : 25 **funct7**	24 : 20 **rs2**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 7 **rD**	6 : 0 **opcode**	
010 1000	00000	src1	011	dest	010 1011	**cv.abs rD, rs1**
010 1001	src2	src1	011	dest	010 1011	**cv.sle rD, rs1, rs2**
010 1010	src2	src1	011	dest	010 1011	**cv.sleu rD, rs1, rs2**
010 1011	src2	src1	011	dest	010 1011	**cv.min rD, rs1, rs2**
010 1100	src2	src1	011	dest	010 1011	**cv.minu rD, rs1, rs2**
010 1101	src2	src1	011	dest	010 1011	**cv.max rD, rs1, rs2**
010 1110	src2	src1	011	dest	010 1011	**cv.maxu rD, rs1, rs2**
011 0000	00000	src1	011	dest	010 1011	**cv.exths rD, rs1**
011 0001	00000	src1	011	dest	010 1011	**cv.exthz rD, rs1**
011 0010	00000	src1	011	dest	010 1011	**cv.extbs rD, rs1**
011 0011	00000	src1	011	dest	010 1011	**cv.extbz rD, rs1**

Table 7.19: General ALU operations encoding

31 : 25 **funct7**	24 : 20 **Is2[4:0]**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 7 **rD**	6 : 0 **opcode**	
011 1000	Luimm5[4:0]	src1	011	dest	010 1011	**cv.clip rD, rs1, Is2**
011 1001	Luimm5[4:0]	src1	011	dest	010 1011	**cv.clipu rD, rs1, Is2**
011 1010	src2	src1	011	dest	010 1011	**cv.clipr rD, rs1, rs2**
011 1011	src2	src1	011	dest	010 1011	**cv.clipur rD, rs1, rs2**

Table 7.20: General ALU operations encoding

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	**cv.addN rD, rs1, rs2, Is3**
01	Luimm5[4:0]	src2	src1	010	dest	101 1011	**cv.adduN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	010	dest	101 1011	**cv.addRN rD, rs1, rs2, Is3**
11	Luimm5[4:0]	src2	src1	010	dest	101 1011	**cv.adduRN rD, rs1, rs2, Is3**
00	Luimm5[4:0]	src2	src1	011	dest	101 1011	**cv.subN rD, rs1, rs2, Is3**
01	Luimm5[4:0]	src2	src1	011	dest	101 1011	**cv.subuN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	011	dest	101 1011	**cv.subRN rD, rs1, rs2, Is3**
11	Luimm5[4:0]	src2	src1	011	dest	101 1011	**cv.subuRN rD, rs1, rs2, Is3**

Table 7.21: General ALU operations encoding

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	**cv.addNr rD, rs1, rs2**
100 0001	src2	src1	011	dest	010 1011	**cv.adduNr rD, rs1, rs**
100 0010	src2	src1	011	dest	010 1011	**cv.addRNr rD, rs1, rs**
100 0011	src2	src1	011	dest	010 1011	**cv.adduRNr rD, rs1, rs2**
100 0100	src2	src1	011	dest	010 1011	**cv.subNr rD, rs1, rs2**
100 0101	src2	src1	011	dest	010 1011	**cv.subuNr rD, rs1, rs2**
100 0110	src2	src1	011	dest	010 1011	**cv.subRNr rD, rs1, rs2**
100 0111	src2	src1	011	dest	010 1011	**cv.subuRNr rD, rs1, rs2**

7.5.6 Immediate Branching operations

Table 7.22: Immediate Branching operations

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

7.5.7 Immediate Branching Encoding

Table 7.23: Immediate Branching encoding

31 **Imm12[12]**	30 : 25 **Imm12[10:5]**	24 : 20 **Imm5**	19 : 15 **rs1**	14 : 12 **funct3**	11 : 8 **Imm12**	7 **Imm12**	6 : 0 **opcode**	
Imm12[12]	Imm12[10:5]	Imm5	src1	110	Imm12[4:1]	Imm12[11]	000 1011	**cv.beqimm rs1, Imm5, Imm12**
Imm12[12]	Imm12[10:5]	Imm5	src1	111	Imm12[4:1]	Imm12[11]	000 1011	**cv.bneimm rs1, Imm5, Imm12**

7.6 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 `COREV_PULP` == 1.

7.6.1 16-Bit x 16-Bit Multiplication operations

Table 7.24: 16-Bit x 16-Bit Multiplication operations

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

7.6.2 16-Bit x 16-Bit Multiplication pseudo-instructions

Table 7.25: 16-Bit x 16-Bit Multiplication pseudo-instructions

Mnemonic	Base Instruction	Description
cv.mulu rD, rs1, rs2	**cv.muluN rD, rs1, rs2, 0**	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0])) >> 0 Note: Logical shift right.
cv.mulhhu rD, rs1, rs2	**cv.mulhhuN rD, rs1, rs2, 0**	rD[31:0] = (Zext(rs1[31:16]) * Zext(rs2[31:16])) >> 0 Note: Logical shift right.
cv.muls rD, rs1, rs2	**cv.mulsN rD, rs1, rs2, 0**	rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0])) >> 0 Note: Arithmetic shift right.
cv.mulhhs rD, rs1, rs2	**cv.mulhhsN rD, rs1, rs2, 0**	rD[31:0] = (Sext(rs1[31:16]) * Sext(rs2[31:16])) >> 0 Note: Arithmetic shift right.

7.6.3 16-Bit x 16-Bit Multiply-Accumulate operations

Table 7.26: 16-Bit x 16-Bit Multiply-Accumulate operations

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

7.6.4 32-Bit x 32-Bit Multiply-Accumulate operations

Table 7.27: 32-Bit x 32-Bit Multiply-Accumulate operations

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

7.6.5 Encoding

Table 7.28: 16-Bit x 16-Bit Multiplication encoding

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	**cv.muluN rD, rs1, rs2, Is3**
01	Luimm5[4:0]	src2	src1	101	dest	101 1011	**cv.mulhhuN rD, rs1, rs2, Is3**
00	Luimm5[4:0]	src2	src1	100	dest	101 1011	**cv.mulsN rD, rs1, rs2, Is3**
01	Luimm5[4:0]	src2	src1	100	dest	101 1011	**cv.mulhhsN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	101	dest	101 1011	**cv.muluRN rD, rs1, rs2, Is3**
11	Luimm5[4:0]	src2	src1	101	dest	101 1011	**cv.mulhhuRN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	100	dest	101 1011	**cv.mulsRN rD, rs1, rs2, Is3**
11	Luimm5[4:0]	src2	src1	100	dest	101 1011	**cv.mulhhsRN rD, rs1, rs2, Is3**

Table 7.29: 16-Bit x 16-Bit Multiply-Accumulate encoding

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	111	dest	101 1011	**cv.macuN rD, rs1, rs2, Is3**
01	Luimm5[4:0]	src2	src1	111	dest	101 1011	**cv.machhuN rD, rs1, rs2, Is3**
00	Luimm5[4:0]	src2	src1	110	dest	101 1011	**cv.macsN rD, rs1, rs2, Is3**
01	Luimm5[4:0]	src2	src1	110	dest	101 1011	**cv.machhsN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	111	dest	101 1011	**cv.macuRN rD, rs1, rs2, Is3**
11	Luimm5[4:0]	src2	src1	111	dest	101 1011	**cv.machhuRN rD, rs1, rs2, Is3**
10	Luimm5[4:0]	src2	src1	110	dest	101 1011	**cv.macsRN rD, rs1, rs2, Is3**
11	Luimm5[4:0]	src2	src1	110	dest	101 1011	**cv.machhsRN rD, rs1, rs2, Is3**

Table 7.30: 32-Bit x 32-Bit Multiply-Accumulate 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	**cv.mac rD, rs1, rs2**
100 1001	src2	src1	011	dest	010 1011	**cv.msu rD, rs1, rs2**

7.7 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 `COREV_PULP` == 1.

Note: See the comments at the start of *CORE-V Instruction Set Custom Extensions* on availability of the compiler tool chains. Support for SIMD will be primarily through assembly code and builtin functions, with no auto-vectorization

and limited other optimization. Simple auto-vectorization (add, sub…) and optimization will be evaluated once a stable GCC toolchain is available.

SIMD instructions are available in two flavors:

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

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

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

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

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

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

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

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

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

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

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

3. Immediate scalar replication mode (.sci), vector-scalar operation. Operand 1 is treated as vector, while operand 2 is treated as a scalar and comes from a 6-bit immediate.

   The immediate is either sign- or zero-extended depending on the operation. If not specified, the immediate is sign-extended with the exception of all cv.shuffle* where it is always unsigned.

   e.g. cv.add.sci.h x3,x2,-22 performs:

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

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

And finally for all the SIMD Bit Manipulation instructions, Imm6 is zero-extended.

In the following tables, 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.

7.7.1 SIMD ALU operations

Table 7.31: SIMD ALU operations

Mnemonic	Description
cv.add[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = (rs1[i] + op2[i]) & {0xFFFF, 0xFF}
cv.sub[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = (rs1[i] - op2[i]) & {0xFFFF, 0xFF}
cv.avg[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = ((rs1[i] + op2[i]) & {0xFFFF, 0xFF}) >> 1 Note: Arithmetic right shift.
cv.avgu[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = ((rs1[i] + op2[i]) & {0xFFFF, 0xFF}) >> 1 Note: Immediate is zero-extended, shift is logical.
cv.min[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = rs1[i] < op2[i] ? rs1[i] : op2[i]
cv.minu[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = rs1[i] < op2[i] ? rs1[i] : op2[i] Note: Immediate is zero-extended, comparison is unsigned.
cv.max[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = rs1[i] > op2[i] ? rs1[i] : op2[i]
cv.maxu[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = rs1[i] > op2[i] ? rs1[i] : op2[i] Note: Immediate is zero-extended, comparison is unsigned.
cv.srl[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = rs1[i] >> op2[i] Note: Immediate is zero-extended, shift is logical. Only Imm6[3:0] and rs2[3:0] are used for .h instruction and Imm6[2:0] and rs2[2:0] for .b instruction. In .sci case, unused Imm6 bits must be set to 0.
cv.sra[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = rs1[i] >>> op2[i] Note: Immediate is zero-extended, shift is arithmetic. Only Imm6[3:0] and rs2[3:0] are used for .h instruction and Imm6[2:0] and rs2[2:0] for .b instruction. In .sci case, unused Imm6 bits must be set to 0.
cv.sll[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = rs1[i] << op2[i] Note: Immediate is zero-extended, shift is logical. Only Imm6[3:0] and rs2[3:0] are used for .h instruction and Imm6[2:0] and rs2[2:0] for .b instruction. In .sci case, unused Imm6 bits must be set to 0.
cv.or[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = rs1[i] | op2[i]
cv.xor[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = rs1[i] ^ op2[i]
cv.and[.sc,.sci]{.h,.b} rD, rs1, [rs2, Imm6]	rD[i] = rs1[i] & op2[i]
cv.abs{.h,.b} rD, rs1	rD[i] = rs1[i] < 0 ? -rs1[i] : rs1[i]

SIMD Bit Manipulation operations

Table 7.32: SIMD Bit Manipulation operations

Mnemonic	Description
cv.extract.h rD, rs1, Imm6	rD = Sext(rs1[I0*16+15:I0*16]) Note: Only Imm6[0] bit is used and other Imm6 bits must be set to 0.
cv.extract.b rD, rs1, Imm6	rD = Sext(rs1[(I1:I0)*8+7:(I1:I0)*8]) Note: Only Imm6[1:0] bits are used and other Imm6 bits must be set to 0.
cv.extractu.h rD, rs1, Imm6	rD = Zext(rs1[I0*16+15:I0*16]) Note: Only Imm6[0] bit is used and other Imm6 bits must be set to 0.
cv.extractu.b rD, rs1, Imm6	rD = Zext(rs1[(I1:I0)*8+7:(I1:I0)*8]) Note: Only Imm6[1:0] bits are used and other Imm6 bits must be set to 0.
cv.insert.h rD, rs1, Imm6	rD[I0*16+15:I0*16] = rs1[15:0] Note: The rest of the bits of rD are untouched and keep their previous value. Only Imm6[0] bit is used and other Imm6 bits must be set to 0.
cv.insert.b rD, rs1, Imm6	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. Only Imm6[1:0] bits are used and other Imm6 bits must be set to 0.

SIMD Dot Product operations

Table 7.33: SIMD Dot Product operations

Mnemonic	Description
cv.dotup[.sc,.sci].h rD, rs1, [rs2, Imm6]	rD = rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operands are unsigned.
cv.dotup[.sc,.sci].b rD, rs1, [rs2, Imm6]	rD = rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operands are unsigned.
cv.dotusp[.sc,.sci].h rD, rs1, [rs2, Imm6]	rD = rs1[0] * op2[0] + rs1[1] * op2[1] Note: rs1 is treated as unsigned, while op2 is treated as signed.
cv.dotusp[.sc,.sci].b rD, rs1, [rs2, Imm6]	rD = rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: rs1 is treated as unsigned, while op2 is treated as signed.
cv.dotsp[.sc,.sci].h rD, rs1, [rs2, Imm6]	rD = rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operands are signed.
cv.dotsp[.sc,.sci].b rD, rs1, [rs2, Imm6]	rD = rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operands are signed.
cv.sdotup[.sc,.sci].h rD, rs1, [rs2, Imm6]	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operands are unsigned.
cv.sdotup[.sc,.sci].b rD, rs1, [rs2, Imm6]	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operands are unsigned.
cv.sdotusp[.sc,.sci].h rD, rs1, [rs2, Imm6]	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] Note: rs1 is treated as unsigned while op2 is treated as signed.
cv.sdotusp[.sc,.sci].b rD, rs1, [rs2, Imm6]	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: rs1 is treated as unsigned while op2 is treated as signed.
cv.sdotsp[.sc,.sci].h rD, rs1, [rs2, Imm6]	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operands are signed.
cv.sdotsp[.sc,.sci].b rD, rs1, [rs2, Imm6]	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operands are signed.

SIMD Shuffle and Pack operations

Table 7.34: SIMD Shuffle and Pack operations

Mnemonic	Description
cv.shuffle.h rD, rs1, rs2	rD[31:16] = rs1[rs2[16]*16+15:rs2[16]*16] rD[15:0] = rs1[rs2[0]*16+15:rs2[0]*16]
cv.shuffle.sci.h rD, rs1, Imm6	rD[31:16] = rs1[I1*16+15:I1*16] rD[15:0] = rs1[I0*16+15:I0*16] Note: Only Imm6[1:0] bits are used and other Imm6 bits must be set to 0.
cv.shuffle.b rD, rs1, rs2	rD[31:24] = rs1[rs2[25:24]*8+7:rs2[25:24]*8] rD[23:16] = rs1[rs2[17:16]*8+7:rs2[17:16]*8] rD[15:8] = rs1[rs2[9:8]*8+7:rs2[9:8]*8] rD[7:0] = rs1[rs2[1:0]*8+7:rs2[1:0]*8]
cv.shuffleI0.sci.b rD, rs1, Imm6	rD[31:24] = rs1[7:0] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
cv.shuffleI1.sci.b rD, rs1, Imm6	rD[31:24] = rs1[15:8] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
cv.shuffleI2.sci.b rD, rs1, Imm6	rD[31:24] = rs1[23:16] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
cv.shuffleI3.sci.b rD, rs1, Imm6	rD[31:24] = rs1[31:24] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
cv.shuffle2.h rD, rs1, rs2	rD[31:16] = ((rs2[17] == 1) ? rs1 : rD)[rs2[16]*16+15:rs2[16]*16] rD[15:0] = ((rs2[1] == 1) ? rs1 : rD)[rs2[0]*16+15:rs2[0]*16]
cv.shuffle2.b rD, rs1, rs2	rD[31:24] = ((rs2[26] == 1) ? rs1 : rD)[rs2[25:24]*8+7:rs2[25:24]*8] rD[23:16] = ((rs2[18] == 1) ? rs1 : rD)[rs2[17:16]*8+7:rs2[17:16]*8] rD[15:8] = ((rs2[10] == 1) ? rs1 : rD)[rs2[9:8]*8+7:rs2[9:8]*8] rD[7:0] = ((rs2[2] == 1) ? rs1 : rD)[rs2[1:0]*8+7:rs2[1:0]*8]
cv.pack rD, rs1, rs2	rD[31:16] = rs1[15:0] rD[15:0] = rs2[15:0]
cv.pack.h rD, rs1, rs2	rD[31:16] = rs1[31:16] rD[15:0] = rs2[31:16]
cv.packhi.b rD, rs1, rs2	rD[31:24] = rs1[7:0] rD[23:16] = rs2[7:0] Note: The rest of the bits of rD are untouched and keep their previous value.
cv.packlo.b rD, rs1, rs2	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.

SIMD ALU Encoding

Table 7.35: 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.add.h rD, rs1, rs2**
0 0000	0	0	src2	src1	100	dest	111 1011	**cv.add.sc.h rD, rs1, rs2**
0 0000	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.add.sci.h rD, rs1, Imm6**
0 0000	0	0	src2	src1	001	dest	111 1011	**cv.add.b rD, rs1, rs2**
0 0000	0	0	src2	src1	101	dest	111 1011	**cv.add.sc.b rD, rs1, rs2**
0 0000	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.add.sci.b rD, rs1, Imm6**
0 0001	0	0	src2	src1	000	dest	111 1011	**cv.sub.h rD, rs1, rs2**
0 0001	0	0	src2	src1	100	dest	111 1011	**cv.sub.sc.h rD, rs1, rs2**
0 0001	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.sub.sci.h rD, rs1, Imm6**
0 0001	0	0	src2	src1	001	dest	111 1011	**cv.sub.b rD, rs1, rs2**
0 0001	0	0	src2	src1	101	dest	111 1011	**cv.sub.sc.b rD, rs1, rs2**
0 0001	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.sub.sci.b rD, rs1, Imm6**
0 0010	0	0	src2	src1	000	dest	111 1011	**cv.avg.h rD, rs1, rs2**
0 0010	0	0	src2	src1	100	dest	111 1011	**cv.avg.sc.h rD, rs1, rs2**
0 0010	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.avg.sci.h rD, rs1, Imm6**
0 0010	0	0	src2	src1	001	dest	111 1011	**cv.avg.b rD, rs1, rs2**
0 0010	0	0	src2	src1	101	dest	111 1011	**cv.avg.sc.b rD, rs1, rs2**
0 0010	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.avg.sci.b rD, rs1, Imm6**
0 0011	0	0	src2	src1	000	dest	111 1011	**cv.avgu.h rD, rs1, rs2**
0 0011	0	0	src2	src1	100	dest	111 1011	**cv.avgu.sc.h rD, rs1, rs2**
0 0011	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.avgu.sci.h rD, rs1, Imm6**
0 0011	0	0	src2	src1	001	dest	111 1011	**cv.avgu.b rD, rs1, rs2**
0 0011	0	0	src2	src1	101	dest	111 1011	**cv.avgu.sc.b rD, rs1, rs2**
0 0011	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.avgu.sci.b rD, rs1, Imm6**
0 0100	0	0	src2	src1	000	dest	111 1011	**cv.min.h rD, rs1, rs2**
0 0100	0	0	src2	src1	100	dest	111 1011	**cv.min.sc.h rD, rs1, rs2**
0 0100	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.min.sci.h rD, rs1, Imm6**
0 0100	0	0	src2	src1	001	dest	111 1011	**cv.min.b rD, rs1, rs2**
0 0100	0	0	src2	src1	101	dest	111 1011	**cv.min.sc.b rD, rs1, rs2**
0 0100	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.min.sci.b rD, rs1, Imm6**
0 0101	0	0	src2	src1	000	dest	111 1011	**cv.minu.h rD, rs1, rs2**
0 0101	0	0	src2	src1	100	dest	111 1011	**cv.minu.sc.h rD, rs1, rs2**
0 0101	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.minu.sci.h rD, rs1, Imm6**
0 0101	0	0	src2	src1	001	dest	111 1011	**cv.minu.b rD, rs1, rs2**
0 0101	0	0	src2	src1	101	dest	111 1011	**cv.minu.sc.b rD, rs1, rs2**
0 0101	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.minu.sci.b rD, rs1, Imm6**
0 0110	0	0	src2	src1	000	dest	111 1011	**cv.max.h rD, rs1, rs2**
0 0110	0	0	src2	src1	100	dest	111 1011	**cv.max.sc.h rD, rs1, rs2**
0 0110	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.max.sci.h rD, rs1, Imm6**
0 0110	0	0	src2	src1	001	dest	111 1011	**cv.max.b rD, rs1, rs2**
0 0110	0	0	src2	src1	101	dest	111 1011	**cv.max.sc.b rD, rs1, rs2**
0 0110	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.max.sci.b rD, rs1, Imm6**
0 0111	0	0	src2	src1	000	dest	111 1011	**cv.maxu.h rD, rs1, rs2**
0 0111	0	0	src2	src1	100	dest	111 1011	**cv.maxu.sc.h rD, rs1, rs2**
0 0111	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.maxu.sci.h rD, rs1, Imm6**

continues on next page

Table 7.35 – 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 0111	0	0	src2	src1	001	dest	111 1011	**cv.maxu.b rD, rs1, rs2**
0 0111	0	0	src2	src1	101	dest	111 1011	**cv.maxu.sc.b rD, rs1, rs2**
0 0111	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.maxu.sci.b rD, rs1, Imm6**
0 1000	0	0	src2	src1	000	dest	111 1011	**cv.srl.h rD, rs1, rs2**
0 1000	0	0	src2	src1	100	dest	111 1011	**cv.srl.sc.h rD, rs1, rs2**
0 1000	0	Imm6[0] 00 Imm6[3:1]		src1	110	dest	111 1011	**cv.srl.sci.h rD, rs1, Imm6**
0 1000	0	0	src2	src1	001	dest	111 1011	**cv.srl.b rD, rs1, rs2**
0 1000	0	0	src2	src1	101	dest	111 1011	**cv.srl.sc.b rD, rs1, rs2**
0 1000	0	Imm6[0] 000 Imm6[2:1]		src1	111	dest	111 1011	**cv.srl.sci.b rD, rs1, Imm6**
0 1001	0	0	src2	src1	000	dest	111 1011	**cv.sra.h rD, rs1, rs2**
0 1001	0	0	src2	src1	100	dest	111 1011	**cv.sra.sc.h rD, rs1, rs2**
0 1001	0	Imm6[0] 00 Imm6[3:1]		src1	110	dest	111 1011	**cv.sra.sci.h rD, rs1, Imm6**
0 1001	0	0	src2	src1	001	dest	111 1011	**cv.sra.b rD, rs1, rs2**
0 1001	0	0	src2	src1	101	dest	111 1011	**cv.sra.sc.b rD, rs1, rs2**
0 1001	0	Imm6[0] 000 Imm6[2:1]		src1	111	dest	111 1011	**cv.sra.sci.b rD, rs1, Imm6**
0 1010	0	0	src2	src1	000	dest	111 1011	**cv.sll.h rD, rs1, rs2**
0 1010	0	0	src2	src1	100	dest	111 1011	**cv.sll.sc.h rD, rs1, rs2**
0 1010	0	Imm6[0] 00 Imm6[3:1]		src1	110	dest	111 1011	**cv.sll.sci.h rD, rs1, Imm6**
0 1010	0	0	src2	src1	001	dest	111 1011	**cv.sll.b rD, rs1, rs2**
0 1010	0	0	src2	src1	101	dest	111 1011	**cv.sll.sc.b rD, rs1, rs2**
0 1010	0	Imm6[0] 000 Imm6[2:1]		src1	111	dest	111 1011	**cv.sll.sci.b rD, rs1, Imm6**
0 1011	0	0	src2	src1	000	dest	111 1011	**cv.or.h rD, rs1, rs2**
0 1011	0	0	src2	src1	100	dest	111 1011	**cv.or.sc.h rD, rs1, rs2**
0 1011	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.or.sci.h rD, rs1, Imm6**
0 1011	0	0	src2	src1	001	dest	111 1011	**cv.or.b rD, rs1, rs2**
0 1011	0	0	src2	src1	101	dest	111 1011	**cv.or.sc.b rD, rs1, rs2**
0 1011	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.or.sci.b rD, rs1, Imm6**
0 1100	0	0	src2	src1	000	dest	111 1011	**cv.xor.h rD, rs1, rs2**
0 1100	0	0	src2	src1	100	dest	111 1011	**cv.xor.sc.h rD, rs1, rs2**
0 1100	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.xor.sci.h rD, rs1, Imm6**
0 1100	0	0	src2	src1	001	dest	111 1011	**cv.xor.b rD, rs1, rs2**
0 1100	0	0	src2	src1	101	dest	111 1011	**cv.xor.sc.b rD, rs1, rs2**
0 1100	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.xor.sci.b rD, rs1, Imm6**
0 1101	0	0	src2	src1	000	dest	111 1011	**cv.and.h rD, rs1, rs2**
0 1101	0	0	src2	src1	100	dest	111 1011	**cv.and.sc.h rD, rs1, rs2**
0 1101	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.and.sci.h rD, rs1, Imm6**
0 1101	0	0	src2	src1	001	dest	111 1011	**cv.and.b rD, rs1, rs2**
0 1101	0	0	src2	src1	101	dest	111 1011	**cv.and.sc.b rD, rs1, rs2**
0 1101	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.and.sci.b rD, rs1, Imm6**
0 1110	0	0	0	src1	000	dest	111 1011	**cv.abs.h rD, rs1**
0 1110	0	0	0	src1	001	dest	111 1011	**cv.abs.b rD, rs1**
1 0111	0	Imm6[0] 00000		src1	000	dest	111 1011	**cv.extract.h rD, rs1, Imm6**

continues on next page

Table 7.35 – 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 0111	0	Imm6[0] 0000 Imm6[1]		src1	001	dest	111 1011	**cv.extract.b rD, rs1, Imm6**
1 0111	0	Imm6[0] 00000		src1	010	dest	111 1011	**cv.extractu.h rD, rs1, Imm6**
1 0111	0	Imm6[0] 0000 Imm6[1]		src1	011	dest	111 1011	**cv.extractu.b rD, rs1, Imm6**
1 0111	0	Imm6[0] 00000		src1	100	dest	111 1011	**cv.insert.h rD, rs1, Imm6**
1 0111	0	Imm6[0] 0000 Imm6[1]		src1	101	dest	111 1011	**cv.insert.b rD, rs1, Imm6**
1 0000	0	0	src2	src1	000	dest	111 1011	**cv.dotup.h rD, rs1, rs2**
1 0000	0	0	src2	src1	100	dest	111 1011	**cv.dotup.sc.h rD, rs1, rs2**
1 0000	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.dotup.sci.h rD, rs1, Imm6**
1 0000	0	0	src2	src1	001	dest	111 1011	**cv.dotup.b rD, rs1, rs2**
1 0000	0	0	src2	src1	101	dest	111 1011	**cv.dotup.sc.b rD, rs1, rs2**
1 0000	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.dotup.sci.b rD, rs1, Imm6**
1 0001	0	0	src2	src1	000	dest	111 1011	**cv.dotusp.h rD, rs1, rs2**
1 0001	0	0	src2	src1	100	dest	111 1011	**cv.dotusp.sc.h rD, rs1, rs2**
1 0001	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.dotusp.sci.h rD, rs1, Imm6**
1 0001	0	0	src2	src1	001	dest	111 1011	**cv.dotusp.b rD, rs1, rs2**
1 0001	0	0	src2	src1	101	dest	111 1011	**cv.dotusp.sc.b rD, rs1, rs2**
1 0001	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.dotusp.sci.b rD, rs1, Imm6**
1 0010	0	0	src2	src1	000	dest	111 1011	**cv.dotsp.h rD, rs1, rs2**
1 0010	0	0	src2	src1	100	dest	111 1011	**cv.dotsp.sc.h rD, rs1, rs2**
1 0010	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.dotsp.sci.h rD, rs1, Imm6**
1 0010	0	0	src2	src1	001	dest	111 1011	**cv.dotsp.b rD, rs1, rs2**
1 0010	0	0	src2	src1	101	dest	111 1011	**cv.dotsp.sc.b rD, rs1, rs2**
1 0010	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.dotsp.sci.b rD, rs1, Imm6**
1 0011	0	0	src2	src1	000	dest	111 1011	**cv.sdotup.h rD, rs1, rs2**
1 0011	0	0	src2	src1	100	dest	111 1011	**cv.sdotup.sc.h rD, rs1, rs2**
1 0011	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.sdotup.sci.h rD, rs1, Imm6**
1 0011	0	0	src2	src1	001	dest	111 1011	**cv.sdotup.b rD, rs1, rs2**
1 0011	0	0	src2	src1	101	dest	111 1011	**cv.sdotup.sc.b rD, rs1, rs2**
1 0011	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.sdotup.sci.b rD, rs1, Imm6**
1 0100	0	0	src2	src1	000	dest	111 1011	**cv.sdotusp.h rD, rs1, rs2**
1 0100	0	0	src2	src1	100	dest	111 1011	**cv.sdotusp.sc.h rD, rs1, rs2**
1 0100	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.sdotusp.sci.h rD, rs1, Imm6**
1 0100	0	0	src2	src1	001	dest	111 1011	**cv.sdotusp.b rD, rs1, rs2**
1 0100	0	0	src2	src1	101	dest	111 1011	**cv.sdotusp.sc.b rD, rs1, rs2**
1 0100	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.sdotusp.sci.b rD, rs1, Imm6**
1 0101	0	0	src2	src1	000	dest	111 1011	**cv.sdotsp.h rD, rs1, rs2**
1 0101	0	0	src2	src1	100	dest	111 1011	**cv.sdotsp.sc.h rD, rs1, rs2**
1 0101	0	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.sdotsp.sci.h rD, rs1, Imm6**
1 0101	0	0	src2	src1	001	dest	111 1011	**cv.sdotsp.b rD, rs1, rs2**
1 0101	0	0	src2	src1	101	dest	111 1011	**cv.sdotsp.sc.b rD, rs1, rs2**
1 0101	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.sdotsp.sci.b rD, rs1, Imm6**
1 1000	0	0	src2	src1	000	dest	111 1011	**cv.shuffle.h rD, rs1, rs2**
1 1000	0	Imm6[0] 0000 Imm6[1]		src1	110	dest	111 1011	**cv.shuffle.sci.h rD, rs1, Imm6**
1 1000	0	0	src2	src1	001	dest	111 1011	**cv.shuffle.b rD, rs1, rs2**

continues on next page

Table 7.35 – 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[0|5:1]		src1	111	dest	111 1011	**cv.shuffleI0.sci.b rD, rs1, Imm6**
1 1001	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.shuffleI1.sci.b rD, rs1, Imm6**
1 1010	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.shuffleI2.sci.b rD, rs1, Imm6**
1 1011	0	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.shuffleI3.sci.b rD, rs1, Imm6**
1 1100	0	0	src2	src1	000	dest	111 1011	**cv.shuffle2.h rD, rs1, rs2**
1 1100	0	0	src2	src1	001	dest	111 1011	**cv.shuffle2.b rD, rs1, rs2**
1 1110	0	0	src2	src1	000	dest	111 1011	**cv.pack rD, rs1, rs2**
1 1110	0	1	src2	src1	000	dest	111 1011	**cv.pack.h rD, rs1, rs2**
1 1111	0	1	src2	src1	001	dest	111 1011	**cv.packhi.b rD, rs1, rs2**
1 1111	0	0	src2	src1	001	dest	111 1011	**cv.packlo.b rD, rs1, rs2**

7.7.2 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.

Table 7.36: SIMD Comparison operations

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

7.7.3 SIMD Comparison Encoding

Table 7.37: 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.cmpeq.h rD, rs1, rs2**
0 0000	1	0	src2	src1	100	dest	111 1011	**cv.cmpeq.sc.h rD, rs1, rs2**
0 0000	1	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.cmpeq.sci.h rD, rs1, Imm6**
0 0000	1	0	src2	src1	001	dest	111 1011	**cv.cmpeq.b rD, rs1, rs2**
0 0000	1	0	src2	src1	101	dest	111 1011	**cv.cmpeq.sc.b rD, rs1, rs2**
0 0000	1	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.cmpeq.sci.b rD, rs1, Imm6**
0 0001	1	0	src2	src1	000	dest	111 1011	**cv.cmpne.h rD, rs1, rs2**
0 0001	1	0	src2	src1	100	dest	111 1011	**cv.cmpne.sc.h rD, rs1, rs2**
0 0001	1	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.cmpne.sci.h rD, rs1, Imm6**
0 0001	1	0	src2	src1	001	dest	111 1011	**cv.cmpne.b rD, rs1, rs2**
0 0001	1	0	src2	src1	101	dest	111 1011	**cv.cmpne.sc.b rD, rs1, rs2**
0 0001	1	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.cmpne.sci.b rD, rs1, Imm6**
0 0010	1	0	src2	src1	000	dest	111 1011	**cv.cmpgt.h rD, rs1, rs2**
0 0010	1	0	src2	src1	100	dest	111 1011	**cv.cmpgt.sc.h rD, rs1, rs2**
0 0010	1	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.cmpgt.sci.h rD, rs1, Imm6**
0 0010	1	0	src2	src1	001	dest	111 1011	**cv.cmpgt.b rD, rs1, rs2**
0 0010	1	0	src2	src1	101	dest	111 1011	**cv.cmpgt.sc.b rD, rs1, rs2**
0 0010	1	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.cmpgt.sci.b rD, rs1, Imm6**
0 0011	1	0	src2	src1	000	dest	111 1011	**cv.cmpge.h rD, rs1, rs2**
0 0011	1	0	src2	src1	100	dest	111 1011	**cv.cmpge.sc.h rD, rs1, rs2**
0 0011	1	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.cmpge.sci.h rD, rs1, Imm6**
0 0011	1	0	src2	src1	001	dest	111 1011	**cv.cmpge.b rD, rs1, rs2**
0 0011	1	0	src2	src1	101	dest	111 1011	**cv.cmpge.sc.b rD, rs1, rs2**
0 0011	1	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.cmpge.sci.b rD, rs1, Imm6**
0 0100	1	0	src2	src1	000	dest	111 1011	**cv.cmplt.h rD, rs1, rs2**
0 0100	1	0	src2	src1	100	dest	111 1011	**cv.cmplt.sc.h rD, rs1, rs2**
0 0100	1	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.cmplt.sci.h rD, rs1, Imm6**
0 0100	1	0	src2	src1	001	dest	111 1011	**cv.cmplt.b rD, rs1, rs2**
0 0100	1	0	src2	src1	101	dest	111 1011	**cv.cmplt.sc.b rD, rs1, rs2**
0 0100	1	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.cmplt.sci.b rD, rs1, Imm6**
0 0101	1	0	src2	src1	000	dest	111 1011	**cv.cmple.h rD, rs1, rs2**
0 0101	1	0	src2	src1	100	dest	111 1011	**cv.cmple.sc.h rD, rs1, rs2**
0 0101	1	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.cmple.sci.h rD, rs1, Imm6**
0 0101	1	0	src2	src1	001	dest	111 1011	**cv.cmple.b rD, rs1, rs2**
0 0101	1	0	src2	src1	101	dest	111 1011	**cv.cmple.sc.b rD, rs1, rs2**
0 0101	1	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.cmple.sci.b rD, rs1, Imm6**
0 0110	1	0	src2	src1	000	dest	111 1011	**cv.cmpgtu.h rD, rs1, rs2**
0 0110	1	0	src2	src1	100	dest	111 1011	**cv.cmpgtu.sc.h rD, rs1, rs2**
0 0110	1	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.cmpgtu.sci.h rD, rs1, Imm6**
0 0110	1	0	src2	src1	001	dest	111 1011	**cv.cmpgtu.b rD, rs1, rs2**
0 0110	1	0	src2	src1	101	dest	111 1011	**cv.cmpgtu.sc.b rD, rs1, rs2**
0 0110	1	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.cmpgtu.sci.b rD, rs1, Imm6**
0 0111	1	0	src2	src1	000	dest	111 1011	**cv.cmpgeu.h rD, rs1, rs2**
0 0111	1	0	src2	src1	100	dest	111 1011	**cv.cmpgeu.sc.h rD, rs1, rs2**
0 0111	1	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.cmpgeu.sci.h rD, rs1, Imm6**

continues on next page

Table 7.37 – 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 0111	1	0	src2	src1	001	dest	111 1011	**cv.cmpgeu.b rD, rs1, rs2**
0 0111	1	0	src2	src1	101	dest	111 1011	**cv.cmpgeu.sc.b rD, rs1, rs2**
0 0111	1	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.cmpgeu.sci.b rD, rs1, Imm6**
0 1000	1	0	src2	src1	000	dest	111 1011	**cv.cmpltu.h rD, rs1, rs2**
0 1000	1	0	src2	src1	100	dest	111 1011	**cv.cmpltu.sc.h rD, rs1, rs2**
0 1000	1	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.cmpltu.sci.h rD, rs1, Imm6**
0 1000	1	0	src2	src1	001	dest	111 1011	**cv.cmpltu.b rD, rs1, rs2**
0 1000	1	0	src2	src1	101	dest	111 1011	**cv.cmpltu.sc.b rD, rs1, rs2**
0 1000	1	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.cmpltu.sci.b rD, rs1, Imm6**
0 1001	1	0	src2	src1	000	dest	111 1011	**cv.cmpleu.h rD, rs1, rs2**
0 1001	1	0	src2	src1	100	dest	111 1011	**cv.cmpleu.sc.h rD, rs1, rs2**
0 1001	1	Imm6[0|5:1]		src1	110	dest	111 1011	**cv.cmpleu.sci.h rD, rs1, Imm6**
0 1001	1	0	src2	src1	001	dest	111 1011	**cv.cmpleu.b rD, rs1, rs2**
0 1001	1	0	src2	src1	101	dest	111 1011	**cv.cmpleu.sc.b rD, rs1, rs2**
0 1001	1	Imm6[0|5:1]		src1	111	dest	111 1011	**cv.cmpleu.sci.b rD, rs1, Imm6**

7.7.4 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, complex multiplications and complex additions/substractions. 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.

Table 7.38: SIMD Complex-number operations

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

7.7.5 SIMD Complex-number Encoding

Table 7.39: SIMD Complex-number encoding

31 : 27	26	25	24 : 20	19 : 15	14 : 12	11 : 7	6 : 0	
funct5	**F**		**rs2**	**rs1**	**funct3**	**rD**	**opcode**	
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	1	src2	src1	000	dest	111 1011	**cv.cplxmul.i rD, rs1, rs2**
0 1010	1	1	src2	src1	010	dest	111 1011	**cv.cplxmul.i.div2 rD, rs1, rs2**
0 1010	1	1	src2	src1	100	dest	111 1011	**cv.cplxmul.i.div4 rD, rs1, rs2**
0 1010	1	1	src2	src1	110	dest	111 1011	**cv.cplxmul.i.div8 rD, rs1, rs2**
0 1011	1	0	00000	src1	000	dest	111 1011	**cv.cplxconj rD, rs1**
0 1100	1	0	src2	src1	000	dest	111 1011	**cv.subrotmj rD, rs1, rs2**
0 1100	1	0	src2	src1	010	dest	111 1011	**cv.subrotmj.div2 rD, rs1, rs2**
0 1100	1	0	src2	src1	100	dest	111 1011	**cv.subrotmj.div4 rD, rs1, rs2**
0 1100	1	0	src2	src1	110	dest	111 1011	**cv.subrotmj.div8 rD, rs1, rs2**
0 1101	1	0	src2	src1	010	dest	111 1011	**cv.add.div2 rD, rs1, rs2**
0 1101	1	0	src2	src1	100	dest	111 1011	**cv.add.div4 rD, rs1, rs2**
0 1101	1	0	src2	src1	110	dest	111 1011	**cv.add.div8 rD, rs1, rs2**
0 1110	1	0	src2	src1	010	dest	111 1011	**cv.sub.div2 rD, rs1, rs2**
0 1110	1	0	src2	src1	100	dest	111 1011	**cv.sub.div4 rD, rs1, rs2**
0 1110	1	0	src2	src1	110	dest	111 1011	**cv.sub.div8 rD, rs1, rs2**

CHAPTER

EIGHT

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 `mhpmcounter3(h)` - `mhpmcounter31(h)` and the corresponding event selector CSRs `mhpmevent3` - `mhpmevent31`, and the `mcountinhibit` CSR to individually enable/disable the counters. `mcycle(h)` and `minstret(h)` are always available.

All counters are 64 bit wide.

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

Unimplemented counters always read 0.

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

8.1 Event Selector

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

Table 8.1: Event Selector

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

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

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

8.2 Controlling the counters from software

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

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

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

8.3 Parametrization at synthesis time

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

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

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

- 64 flops for `mhpmcounterX`

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

8.4 Time Registers (`time(h)`)

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

CHAPTER

NINE

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.

9.1 CSR Map

Table 9.1 lists all implemented CSRs. Two columns in Table 9.1 may require additional explanation:

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 privilge level:

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

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

In the **Description** column there is a specific comment which identifies those CSRs that are dependent on the value of specific parameters. If these parameters are not set as indicated in Table 9.1 then the associated CSR is not implemented. If the column does not mention any parameter then the associated CSR is always implemented.

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

Table 9.1: Control and Status Register Map

CSR Address	Name	Privilege	Description
User CSRs			
0x001	`fflags`	URW	Floating-point accrued exceptions. Only present if `FPU` = 1
0x002	`frm`	URW	Floating-point dynamic rounding mode. Only present if `FPU` = 1
0x003	`fcsr`	URW	Floating-point control and status register. Only present if `FPU` = 1
0xC00	`cycle`	URO	(HPM) Cycle Counter
0xC02	`instret`	URO	(HPM) Instructions-Retired Counter

continues on next page

Table 9.1 – continued from previous page

CSR Address	Name	Privilege	Description
0xC03	hpmcounter3	URO	(HPM) Performance-Monitoring Counter 3
. . . .			
0xC1F	hpmcounter31	URO	(HPM) Performance-Monitoring Counter 31
0xC80	cycleh	URO	(HPM) Upper 32 bits Cycle Counter
0xC82	instreth	URO	(HPM) Upper 32 bits Instructions-Retired Counter
0xC83	hpmcounterh3	URO	(HPM) Upper 32 bits Performance-Monitoring Counter 3
. . . .			
0xC9F	hpmcounterh31	URO	(HPM) Upper 32 bits Performance-Monitoring Counter 31
User Custom CSRs			
0xCC0	lpstart0	URO	Hardware Loop 0 Start. Only present if COREV_PULP = 1
0xCC1	lpend0	URO	Hardware Loop 0 End. Only present if COREV_PULP = 1
0xCC2	lpcount0	URO	Hardware Loop 0 Counter. Only present if COREV_PULP = 1
0xCC4	lpstart1	URO	Hardware Loop 1 Start. Only present if COREV_PULP = 1
0xCC5	lpend1	URO	Hardware Loop 1 End. Only present if COREV_PULP = 1
0xCC6	lpcount1	URO	Hardware Loop 1 Counter. Only present if COREV_PULP = 1
0xCD0	uhartid	URO	Hardware Thread ID Only present if COREV_PULP = 1
0xCD1	privlv	URO	Privilege Level Only present if COREV_PULP = 1
0xCD2	zfinx	URO	ZFINX ISA Only present if COREV_PULP = 1 & (FPU = 0 | (FPU = 1 & ZFINX = 1))
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 Event Selector 3
. . . .			
0x33F	mhpmevent31	MRW	(HPM) Machine Performance-Monitoring Event Selector 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

continues on next page

Table 9.1 – continued from previous page

CSR Address	Name	Privilege	Description
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 Counter
0xB03	mhpmcounter3	MRW	(HPM) Machine Performance-Monitoring Counter 3
. . . .			
0xB1F	mhpmcounter31	MRW	(HPM) Machine Performance-Monitoring Counter 31
0xB80	mcycleh	MRW	(HPM) Upper 32 bits Machine Cycle Counter
0xB82	minstreth	MRW	(HPM) Upper 32 bits Machine Instructions-Retired Counter
0xB83	mhpmcounterh3	MRW	(HPM) Upper 32 bits Machine Performance-Monitoring Counter 3
. . . .			
0xB9F	mhpmcounterh31	MRW	(HPM) Upper 32 bits Machine Performance-Monitoring Counter 31
0xF11	mvendorid	MRO	Machine Vendor ID
0xF12	marchid	MRO	Machine Architecture ID
0xF13	mimpid	MRO	Machine Implementation ID
0xF14	mhartid	MRO	Hardware Thread ID

9.2 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 9.1 (or a higher privilege level):

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

9.2.1 Floating-point CSRs

Floating-point accrued exceptions (fflags)

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

Reset Value: 0x0000_0000

Detailed:

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

Floating-point dynamic rounding mode (`frm`)

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

Reset Value: 0x0000_0000

Detailed:

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

Floating-point control and status register (`fcsr`)

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

Reset Value: 0x0000_0000

Detailed:

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

9.2.2 Hardware Loops CSRs

HWLoop Start Address 0/1 (lpstart0/1)

CSR Address: 0xCC0/0xCC4 (only present if COREV_PULP = 1)

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:2	URO	Start Address of the HWLoop 0/1.
1:0	URO	0

HWLoop End Address 0/1 (lpend0/1)

CSR Address: 0xCC1/0xCC5 (only present if COREV_PULP = 1)

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:2	URO	End Address of the HWLoop 0/1.
1:0	URO	0

HWLoop Count Address 0/1 (lpcount0/1)

CSR Address: 0xCC2/0xCC6 (only present if COREV_PULP = 1)

Reset Value: 0x0000_0000

Detailed:

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

9.2.3 Other CSRs

Machine Status (mstatus)

CSR Address: 0x300

Reset Value: 0x0000_1800

Detailed:

Bit #	Mode	Description
31	RO	**SD:** State Dirty SD set to 1 if **FS** = 11 meaning Floating point State is dirty so save/restore is needed in case of context switch. 0 if `FPU` = 0 or (`FPU` = 1 and `ZFINX` = 1).
30:15	RO	0, Unimplemented.
14:13	RW	**FS:** Floating point State (See note below) 00 = Off 01 = Initial 10 = Clean 11 = Dirty 0 if `FPU` = 0 or (`FPU` = 1 and `ZFINX` = 1).
12:11	RW	**MPP:** Machine Previous Priviledge mode Hardwired to 11 when the User mode is not enabled.
10:8	RO	0, Unimplemented.
7	RW	**MPIE:** Machine Previous Interrupt Enable When an exception is encountered, MPIE will be set to MIE. When the mret instruction is executed, the value of MPIE will be stored to MIE.
6:4	RO	0, Unimplemented.
3	RW	**MIE:** Machine Interrupt Enable If you want to enable interrupt handling in your exception handler, set the Interrupt Enable MIE to 1 inside your handler code.
2:0	RO	0, Unimplemented.

Note: As allowed by RISC-V ISA and to simplify MSTATUS.FS update in the design, the state is updated to Dirty when executing any F instructions except for all FSW ones.

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] (x between 16 and 31).
15:12	RO	0
11	RW	**MEIE:** Machine External Interrupt Enable If set, irq_i[11] is enabled.
10:8	RO	0
7	RW	**MTIE:** Machine Timer Interrupt Enable If set, irq_i[7] is enabled.
6:4	RO	0
3	RW	**MSIE:** Machine Software Interrupt Enable If set, irq_i[3] is enabled.
2:0	RO	0

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] 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.

Machine Scratch (mscratch)

CSR Address: 0x340

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RW	Scratch value

Machine Exception PC (mepc)

CSR Address: 0x341

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:1	RW	**MEPC:** Machine Exception Program Counter
0	R0	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.

Machine Cause (mcause)

CSR Address: 0x342

Reset Value: 0x0000_0000

Detailed:

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

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

Machine Trap Value (mtval)

CSR Address: 0x343

Reset Value: 0x0000_0000

Detailed:

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

Machine Interrupt Pending register (mip)

CSR Address: 0x344

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:16	RO	Machine Fast Interrupts Pending If bit x is set, interrupt irq_i[x] is pending (x between 16 and 31).
15:12	RO	0
11	RO	**MEIP:** Machine External Interrupt Pending If set, irq_i[11] is pending.
10:8	RO	0
7	RO	**MTIP:** Machine Timer Interrupt Pending If set, irq_i[7] is pending.
6:4	RO	0
3	RO	**MSIP:** Machine Software Interrupt Pending If set, irq_i[3] is pending.
2:0	RO	0

9.2.4 Trigger CSRs

Trigger Select register (tselect)

CSR Address: 0x7A0

Reset Value: 0x0000_0000

Detailed:

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

Accessible in Debug Mode or M-Mode.

Trigger Data register 1 (tdata1)

CSR Address: 0x7A1

Reset Value: 0x2800_1040

Detailed:

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

Trigger Data register 2 (tdata2)

CSR Address: 0x7A2

Reset Value: 0x0000_0000

Detailed:

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

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

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.

Trigger Info (tinfo)

CSR Address: 0x7A4

Reset Value: 0x0000_0004

Detailed:

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

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

Accessible in Debug Mode or M-Mode.

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.

9.2.5 Debug CSRs

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

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.

Debug Scratch register 0/1 (dscratch0/1)

CSR Address: 0x7B2/0x7B3

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RW	DSCRATCH0/1

9.2.6 Performances counters

Machine Counter-Inhibit register (mcountinhibit)

CSR Address: 0x320

Reset Value: 0x0000_000D

Detailed:

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

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

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

CSR Address: 0x323 - 0x33F

Reset Value: 0x0000_0000

Detailed:

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

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

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.

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.

Machine Performance Monitoring Counter (`mhpmcounter3 .. mhpmcounter31`)

CSR Address: 0xB03 - 0xB1F

Reset Value: 0x0000_0000

Detailed:

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

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

Upper 32 bits 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.

Upper 32 bits 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.

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

CSR Address: 0xB83 - 0xB9F

Reset Value: 0x0000_0000

Detailed:

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

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

Cycle Counter (cycle)

CSR Address: 0xC00

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RO	0

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

Instructions-Retired Counter (`instret`)

CSR Address: 0xC02

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RO	0

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

Performance Monitoring Counter (`hpmcounter3 .. hpmcounter31`)

CSR Address: 0xC03 - 0xC1F

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RO	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.

Upper 32 bits Cycle Counter (`cycleh`)

CSR Address: 0xC80

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RO	0

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

Upper 32 bits Instructions-Retired Counter (`instreth`)

CSR Address: 0xC82

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RO	0

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

Upper 32 bits Performance Monitoring Counter (`hpmcounter3h .. hpmcounter31h`)

CSR Address: 0xC83 - 0xC9F

Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:0	RO	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.

9.2.7 ID CSRs

Machine ISA (`misa`)

CSR Address: 0x301

Reset Value: defined

Detailed:

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

Writes are ignored and all bitfields in the `misa` CSR area read as 0 except for the following:

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

Machine Vendor ID (`mvendorid`)

CSR Address: 0xF11

Reset Value: 0x0000_0602

Detailed:

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

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

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

Machine Implementation ID (`mimpid`)

CSR Address: 0xF13

Reset Value: Defined

Detailed:

Bit #	Mode	Description
31 : 1	RO	0
0	RO	1 if `FPU` = 1 or `COREV_PULP` = 1 or `COREV_CLUSTER` = 1 else 0.

Hardware Thread ID (`mhartid`)

CSR Address: 0xF14

Reset Value: Defined

Detailed:

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

9.2.8 Non-RISC-V CSRs

User Hardware Thread ID (uhartid)

CSR Address: 0xCD0 (only present if COREV_PULP = 1)

Reset Value: Defined

Detailed:

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

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

Privilege Level (privlv)

CSR Address: 0xCD1 (only present if COREV_PULP = 1)

Reset Value: 0x0000_0003

Detailed:

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

ZFINX ISA (zfinx)

CSR Address: 0xCD2 (only present if COREV_PULP = 1 & (FPU = 0 | (FPU = 1 & ZFINX = 1)))

Reset Value: Defined

Bit #	Mode	Description
31:1	RO	0
0	RO	1 if FPU = 1 and ZFINX = 1 else 0.

CHAPTER

TEN

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.

10.1 Interrupt Interface

Table 10.1 describes the interrupt interface.

Table 10.1: Interrupt interface signals

Signal	Direction	Description
`irq_i[31:0]`	input	Level sensistive active high interrupt inputs. Not all interrupt inputs can be used on CV32E40P. Specifically irq_i[15:12], irq_i[10:8], irq_i[6:4] and irq_i[2:0] shall be tied to 0 externally as they are reserved for future standard use (or for cores which are not Machine mode only) in the RISC-V Privileged specification. irq_i[11], irq_i[7], and irq_i[3] correspond to the Machine External Interrupt (MEI), Machine Timer Interrupt (MTI), and Machine Software Interrupt (MSI) respectively. The irq_i[31:16] interrupts are a CV32E40P specific extension to the RISC-V Basic (a.k.a. CLINT) interrupt scheme.
`irq_ack_o`	output	Interrupt acknowledge Set to 1 for one cycle when the interrupt with ID `irq_id_o[4:0]` is taken.
`irq_id_o[4:0]`	output	Interrupt index for taken interrupt Only valid when `irq_ack_o` = 1.

10.2 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 sourcee, e.g. by an external interrupt controller. `irq_ack_o` is a 1 `clk_i` cycle pulse during which `irq_id_o[4:0]` reflects the index in `irq_id[*]` of the taken interrupt.

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

10.3 Exceptions

CV32E40P can trigger an exception due to the following exception causes:

Table 10.2: Exceptions

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 *CORE-V Instruction Set Custom Extensions*). For example, in case the parameter `FPU` is set to 0, the CV32E40P raises an illegal instruction exception for any RVF instruction or CSR instruction trying to access F CSRs. The same concerns PULP extensions everytime both parameters `COREV_PULP` and `CORE_CLUSTER` are set to 0 (see *Core Integration*).

10.4 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.

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

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

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

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

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

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

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

Nesting of interrupts/exceptions in hardware is not supported.

CHAPTER

ELEVEN

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 (either integer of floating-point one) 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.

11.1 Debug Interface

Table 11.1: Debug interface signals

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.

11.2 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.

11.3 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` or `debug_halted_o` is asserted at all times.

Figure 11.1 and show Figure 11.2 show typical examples of transitioning into the `running` and `halted` states.

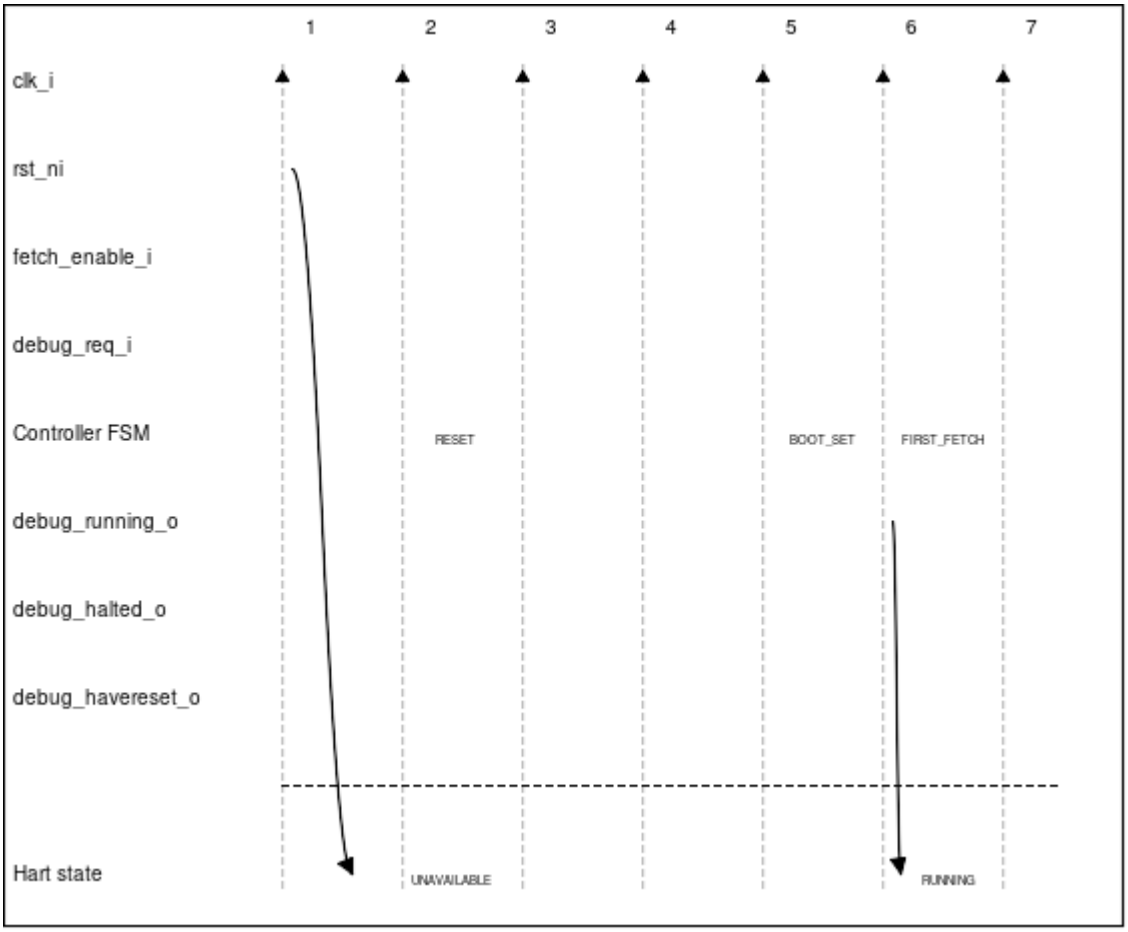

Figure 11.1: Transition into debug `running` state

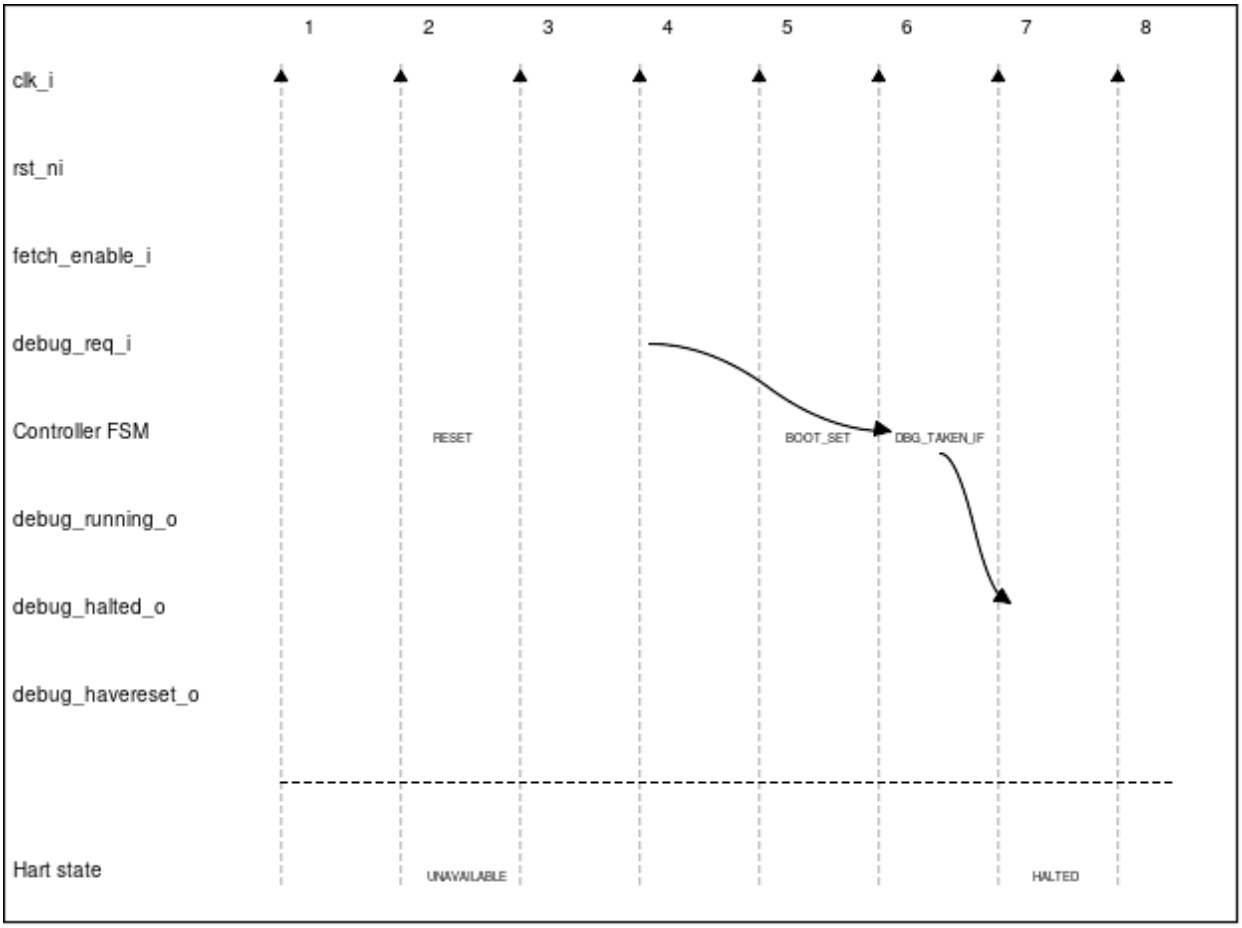

Figure 11.2: Transition into debug `halted` state

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 `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 ways to the `halted` state.
- If `debug_req_i` is asserted during the `running` state, the core will eventually transition into the `halted` state (typically after a couple of cycles).

11.4 EBREAK Behavior

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

11.4.1 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).

As mentioned in *Hardware loops impact on application, exceptions handlers and debugger*, some additional cases exist for MEPC update when ebreak is the last instruction of an Hardware Loop.

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.

11.4.2 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.

There is no forseeen situtation where it would be needed to enter in Debug Mode only on the last instruction of an Hardware Loop but just in case this is mentioned in *Hardware loops impact on application, exceptions handlers and debugger* as well.

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.

11.4.3 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

11.5 Interrupts during Single-Step Behavior

The CV32E40P 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.

CHAPTER

TWELVE

PIPELINE DETAILS

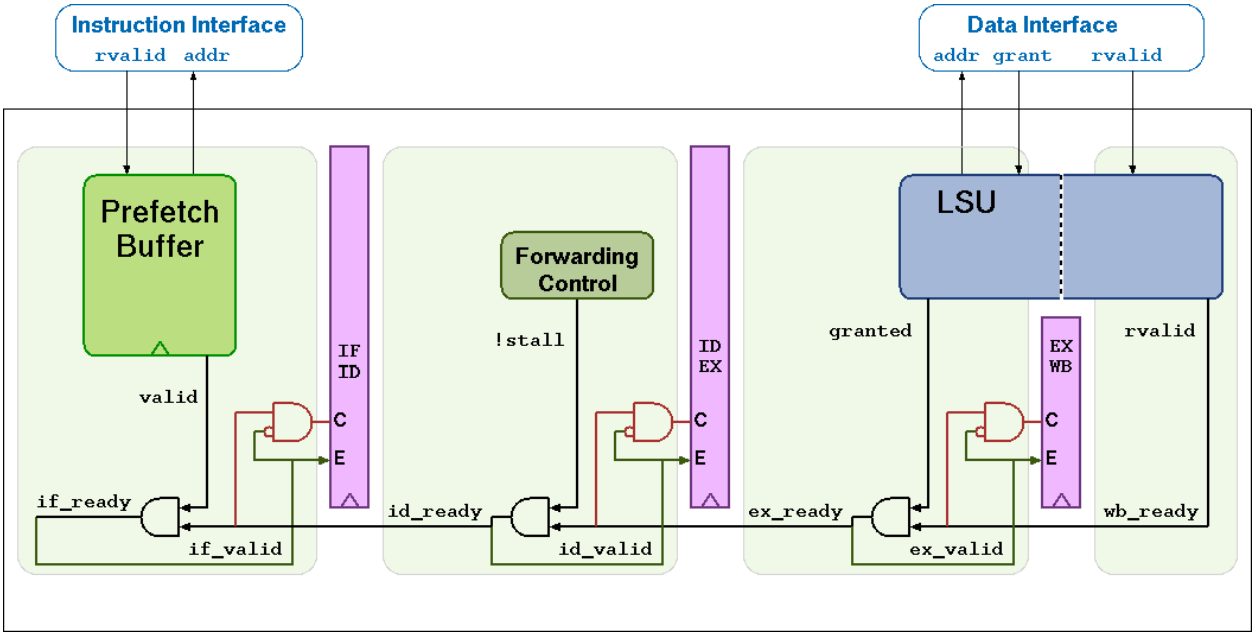

Figure 12.1: CV32E40P Pipeline

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. This prefetch buffer is able to store 2 32-b data. 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.

  The FPU writes back its result at EX stage as well through this ALU/Mult/Div register file write port when FPU_*_LAT is either 0 cycle or greater than 1 cycle. When FPU_*_LAT > 1, FPU write-back has the highest priority so it will stall EX stage if there is a conflict. There are few exceptions to this FPU priority over ALU/Mult/Div.

They are:

- There is a multi-cycle MULH in EX.
- There is a Misaligned LOAD/STORE in EX.
- There is a Post-Increment LOAD/STORE in EX.

In those 3 exceptions, EX will not be stalled, FPU result (and flags) are memorized and will be written back in the register file (and FPU CSR) as soon as there is no conflict anymore.

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

The FPU writes back its result from WB stage as well when FPU_*_LAT is 1 cycle. It is reusing register file LSU write port but LSU has the highest priority over FPU if there is a conflict.

12.1 Hazards

There is a forwarding path betwen ALU, Multiplier and Divider result in EX stage and ID stage flip-flops to avoid the need of a write-through register file. This allows to have 0-cycle penalty between those instructions and immediately following one when using result. This is the same with 0-cycle latency FPU instructions.

But 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
- FPU data hazard when FPU_*_LAT = 1 in case the instruction immediately following a FPU one (except FDIV/FSQRT) uses the result of the FPU

More than 1-cycle penalty will happen when:

- FPU data hazard of FPU_*_LAT cycles (FPU_*_LAT > 1) in case the instruction immediately following a FPU one (except FDIV/FSQRT) uses the result of the FPU
- FPU data hazard in case the instruction immediately following FDIV/FSQRT uses the result of those instructions

Those cycles penalty can be hidden if the compiler is able to add instructions between the instructions causing this data hazard.

12.2 Single- and Multi-Cycle Instructions

Table 12.1 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 12.1: Cycle counts per instruction type

Instruction Type	Cycles	Description
Integer Computational	1	Integer Computational Instructions are defined in the RISCV-V RV32I Base Integer Instruction Set.
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.
Load/Store	1 2 (non-word aligned word transfer) 2 (halfword transfer crossing word boundary) 4 (cv.elw)	Load/Store is handled in 1 bus transaction using both EX and WB stages for 1 cycle each. For misaligned word transfers and for halfword transfers that cross a word boundary 2 bus transactions are performed using EX and WB stages for 2 cycles each. A **cv.elw** takes 4 cycles.
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.
CSR Access	4 (mstatus, mepc, mtvec, mcause, mcycle, minstret, mhpmcounter*, mcycleh, minstreth, mhpmcounter*h, mcountinhibit, mhpmevent*, dscr, dpc, dscratch0, dscratch1) 1 (all the other CSRs)	CSR Access Instruction are defined in 'Zicsr' of the RISC-V specification.
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.
Floating-Point Addition or Multiplication	1..FPU_ADDMUL_LAT + 1	Floating-Point instructions are dispatched to the FPU. Following instructions can be executed by the Core as long as they are not FPU ones and there are no Read-After-Write or Write-After-Write data hazard between them and the destination register of the outstanding FPU instruction. If there are enough instructions between FPU one and the instruction using the result then cycle number is 1. "Enough instruction" number is either FPU_ADDMUL_LAT, FPU_OTHERS_LAT or 19. If there are no instruction in between then cycle number is the maximum value for each category.
Floating-Point Comparison, Conversion or Classify	1..FPU_OTHERS_LAT + 1	
Single Precision Floating-Point Division and Square-Root	1..19	

CHAPTER

THIRTEEN

INSTRUCTION FETCH

The Instruction Fetch (IF) stage of the CV32E40P is able to supply one instruction per cycle to the Instruction Decode (ID) stage if the external bus interface is able to serve one fetch request per cycle. In case of executing compressed instructions, on average less than one 32-bit fetch request 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 buffer performs word-aligned 32-bit prefetches and stores the fetched words in a FIFO with a number of entries depending of a local parameter. It is called `DEPTH` and can be found in `cv32e40p_prefetch_buffer.sv` (default value of 2). As a result of this (speculative) prefetch, CV32E40P can fetch up to `DEPTH` 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 13.1 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 13.1: Instruction Fetch interface signals

Signal	Direction	Description
`instr_addr_o[31:0]`	output	Address, word aligned
`instr_req_o`	output	Request valid, will stay high until `instr_gnt_i` is high for one cycle
`instr_gnt_i`	input	The other side accepted the request. `instr_addr_o` may change in the next cycle.
`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_rdata_i[31:0]`	input	Data read from memory

13.1 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.

13.2 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.

Note: Transactions Ordering As mentioned above, instruction fetch interface can generate up to `DEPTH` outstanding transactions. OBI specification states that links are always in-order from master point of view. So as the fetch interface does not generate transaction id (aid), interconnect infrastructure should ensure that transaction responses come back in the same order they were sent by adding its own additional information.

Figure 13.1 and Figure 13.2 show example timing diagrams of the protocol.

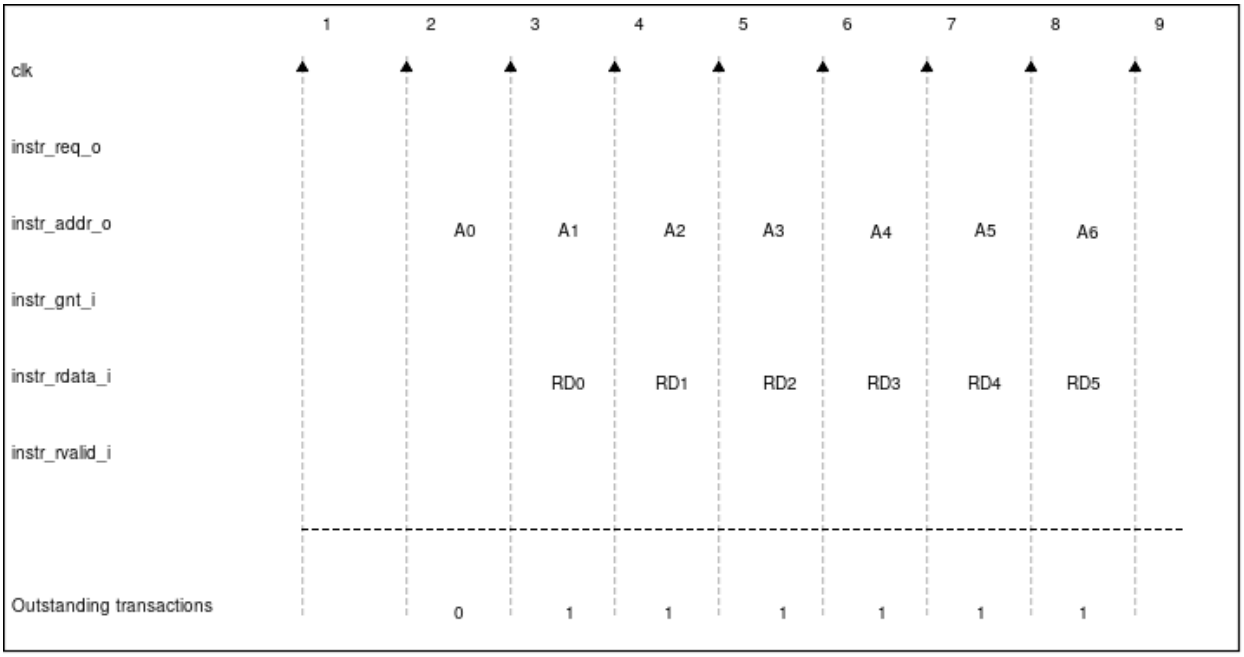

Figure 13.1: Back-to-back Memory Transactions

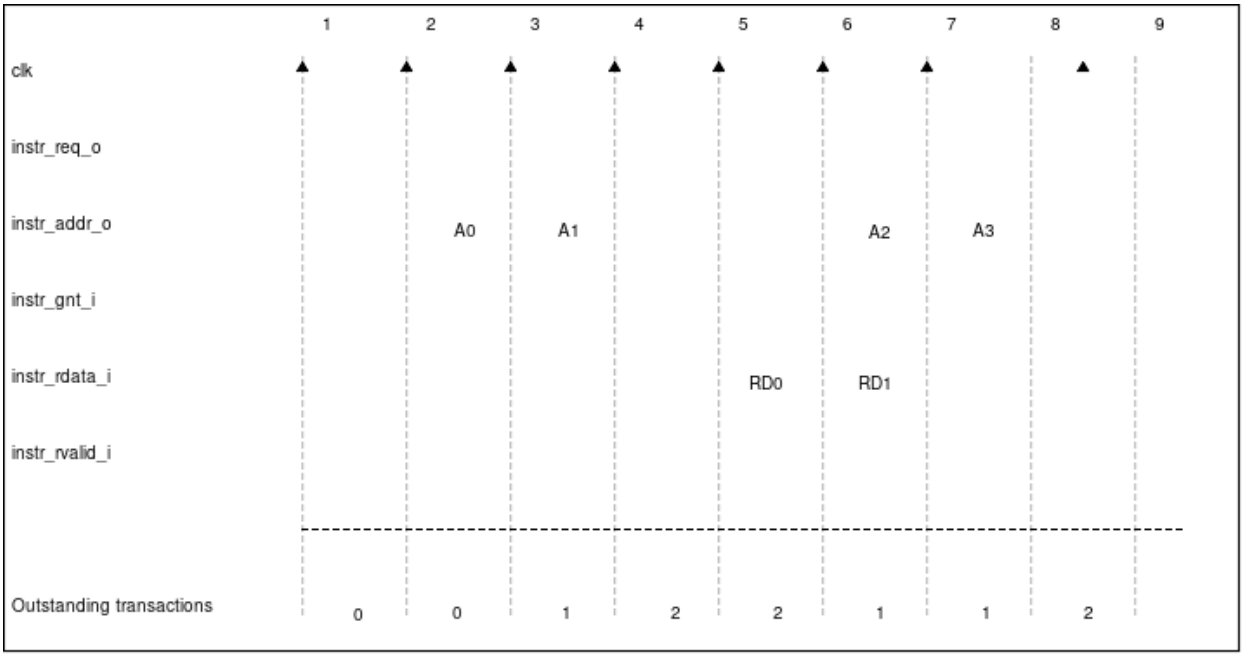

Figure 13.2: Multiple Outstanding Memory Transactions

CHAPTER

FOURTEEN

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. The CV32E40P data interface can cause up to 2 outstanding transactions and there is no FIFO to allow more outstanding requests.

Table 14.1 describes the signals that are used by the LSU.

Table 14.1: LSU interface signals

Signal	Direction	Description
`data_addr_o[31:0]`	output	Address
`data_req_o`	output	Request valid, will stay high until `data_gnt_i` is high for one cycle
`data_gnt_i`	input	The other side accepted the request. `data_addr_o` may change in the next cycle.
`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_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_rdata_i[31:0]`	input	Data read from memory

14.1 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.

14.2 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.

Note: Transactions Ordering As mentioned above, data interface can generate up to 2 outstanding transactions. OBI specification states that links are always in-order from master point of view. So as the data interface does not generate transaction id (aid), interconnect infrastructure should ensure that transaction responses come back in the same order they were sent by adding its own additional information.

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 14.1, Figure 14.2, Figure 14.3 and Figure 14.4 show example timing diagrams of the protocol.

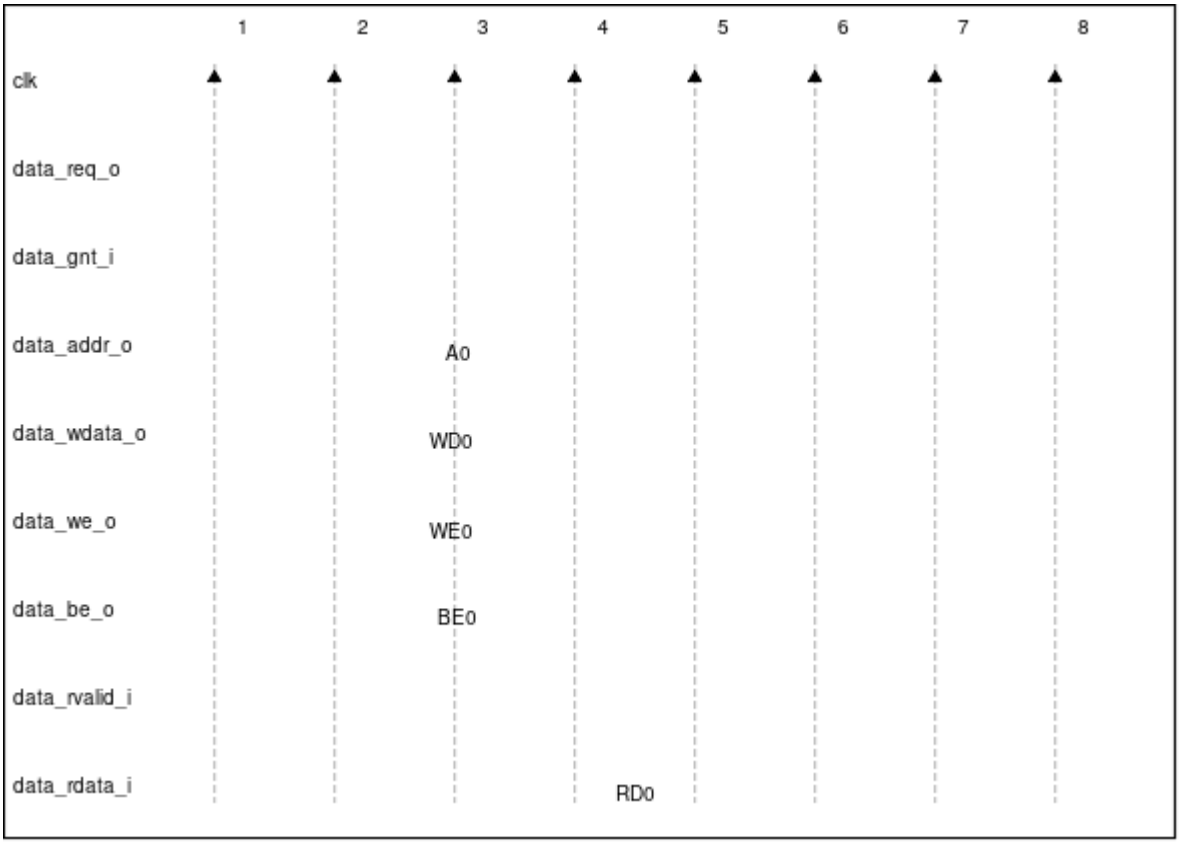

Figure 14.1: Basic Memory Transaction

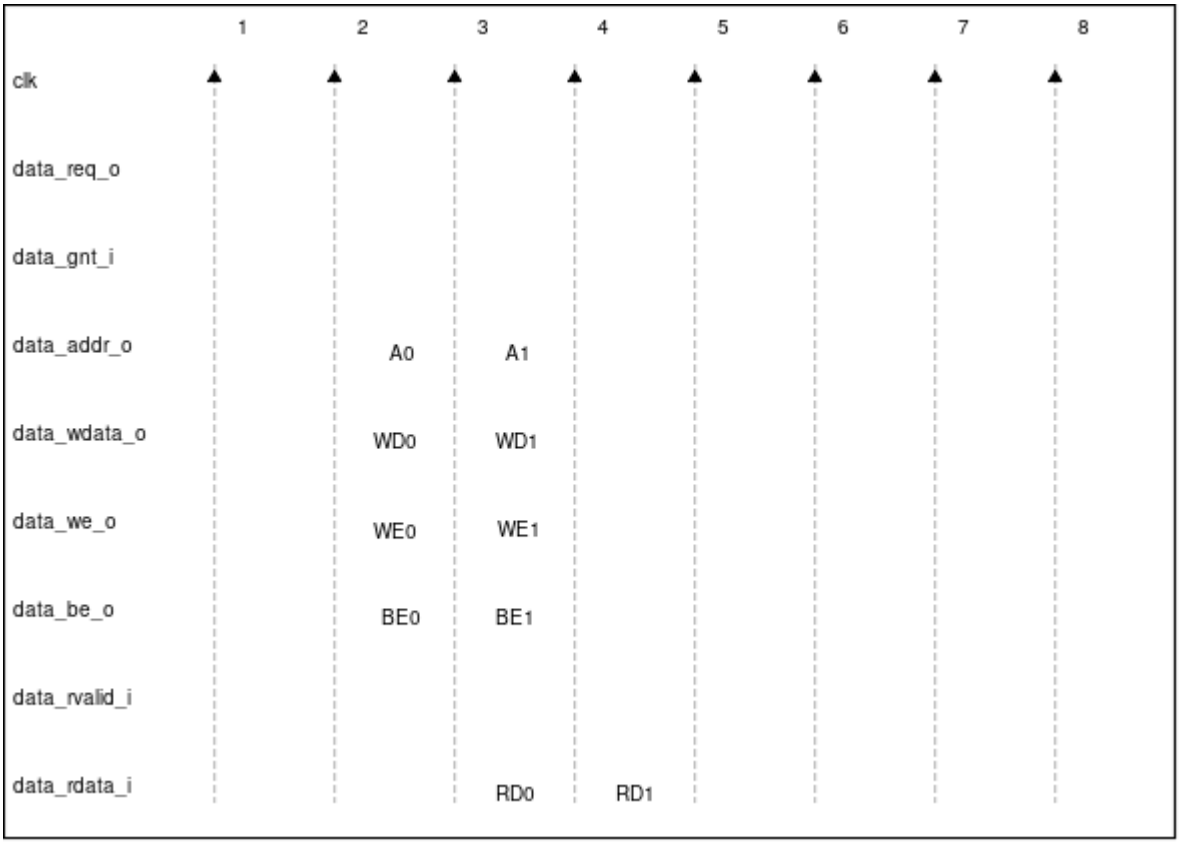

Figure 14.2: Back-to-back Memory Transactions

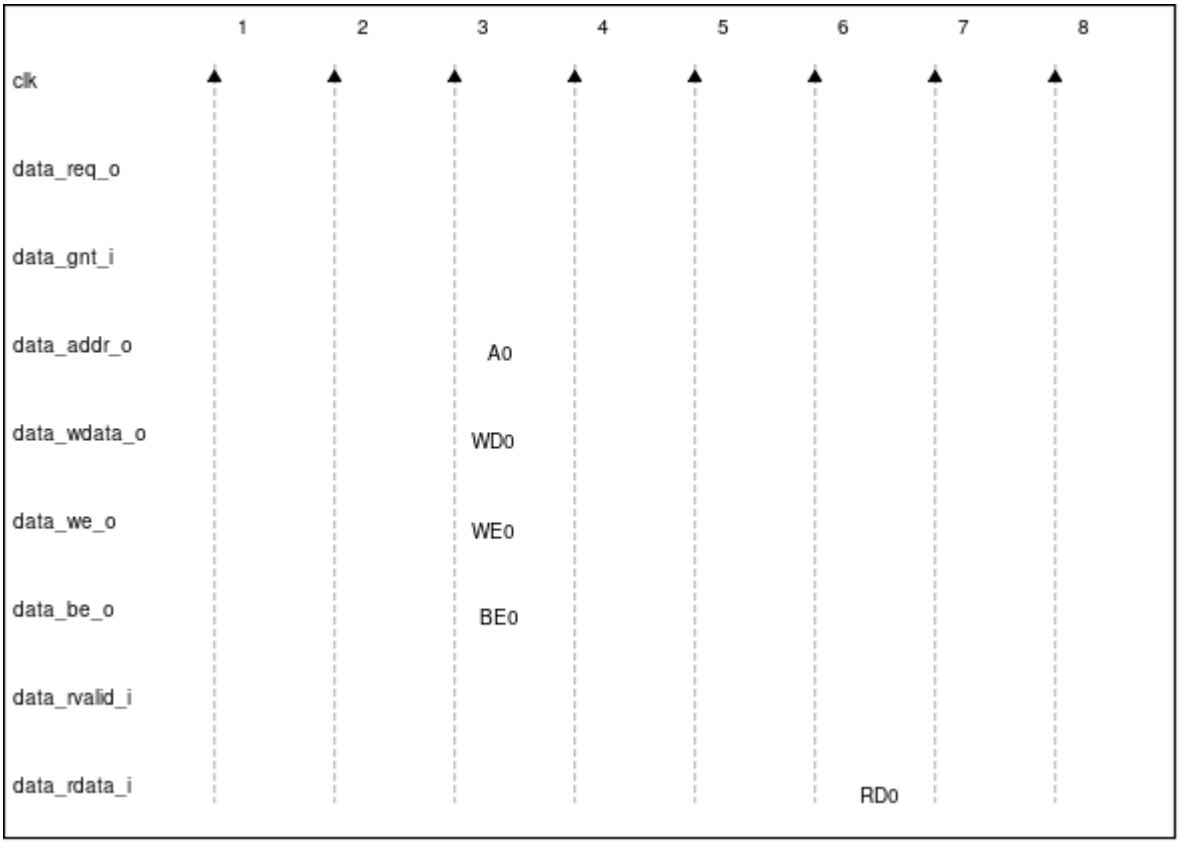

Figure 14.3: Slow Response Memory Transaction

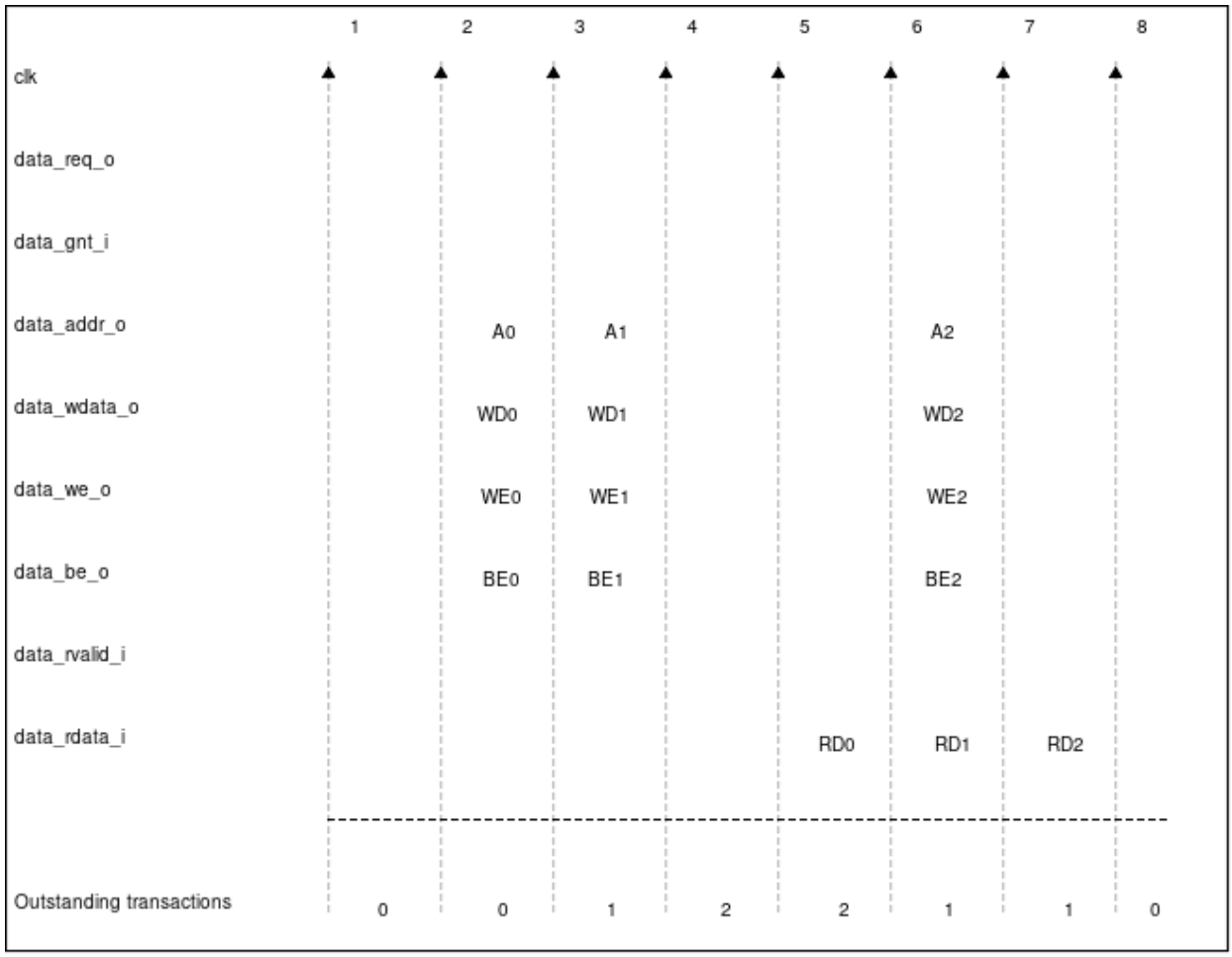

Figure 14.4: Multiple Outstanding Memory Transactions

14.3 Post-Incrementing Load and Store Instructions

This section is only valid if `COREV_PULP = 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.

CHAPTER

FIFTEEN

REGISTER FILE

Source files: `rtl/cv32e40p_register_file_ff.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.

15.1 Floating-Point Register File

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

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

If `ZFINX` parameter is set, there is no additional register bank and FPU instructions are using the same register file than for integer instructions.

CHAPTER

SIXTEEN

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 `COREV_CLUSTER` = 0)
- **cv.elw** instruction (only when `COREV_CLUSTER` = 1)
- `pulp_clock_en_i` (only when `COREV_CLUSTER` = 1)

Table 16.1 describes the Sleep Unit interface.

Table 16.1: Sleep Unit interface signals

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

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

16.1 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.

16.2 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)
- `COREV_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 16.1 shows an example waveform for sleep mode entry because of a **wfi** instruction.

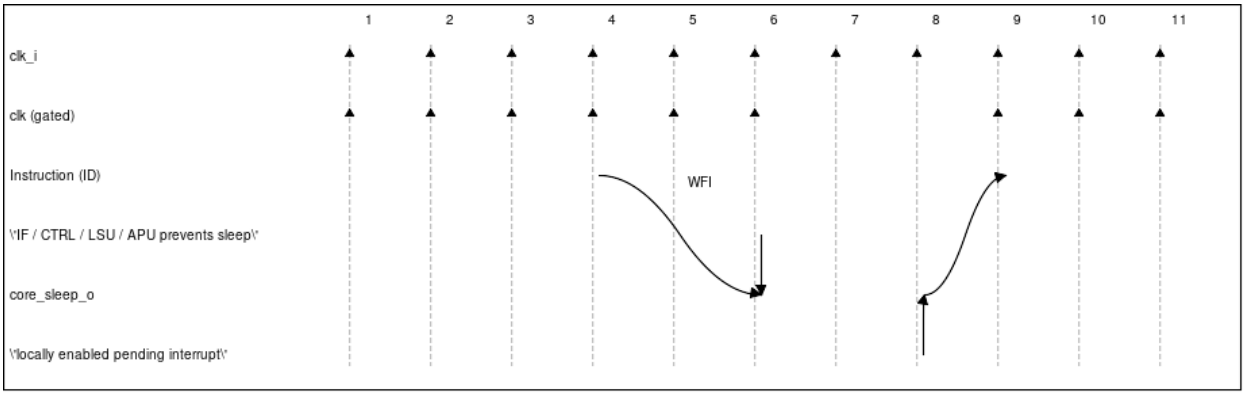

Figure 16.1: **wfi** example

16.3 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 `COREV_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 (`COREV_CLUSTER` = 0), the `pulp_clock_en_i` signal is not used and should be tied to 0.

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

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

As `pulp_clock_en_i` can directly impact the internal clock gate, certain requirements are imposed on the environment of CV32E40P in case `COREV_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 16.2 shows an example waveform for sleep mode entry because of a **cv.elw** instruction.

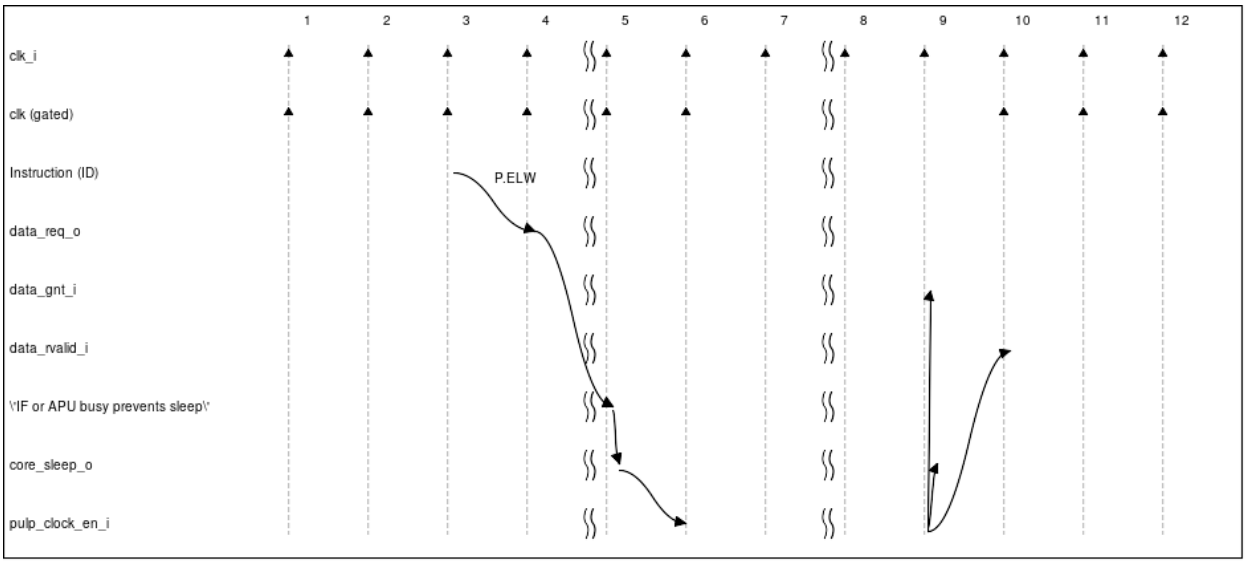

Figure 16.2: **cv.elw** example

CHAPTER

SEVENTEEN

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 indentified 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.

17.1 What happens after RTL Freeze?

17.1.1 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.

17.1.2 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.

17.1.3 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.

17.1.4 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 17.1 shows the aforementioned rules.

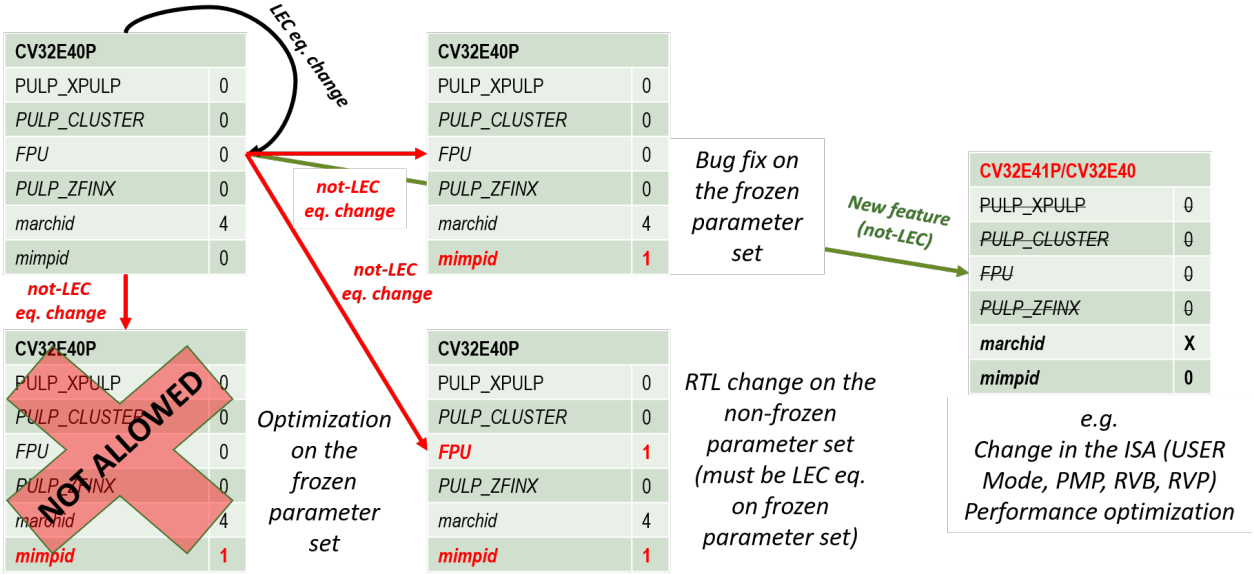

Figure 17.1: Versions control of CV32E40P

17.2 Non-backward compatibility

For cv32e40p_v2.0.0, some modifications have been done on `cv32e40p_top` and `cv32e40p_core` parameters names.

It is worth mentioning that if the core in its v1 version was/is instantiated without parameters setting, backward compatibility is still correct as all parameters default values are set to v1 values.

17.2.1 Parameters

As RTL has been updated to fully support ratified RISC-V Zfinx, old PULP_ZFINX parameter has been renamed ZFINX in all design and verification files.

To differentiate v1 to v2 encoding of PULP instructions, old PULP_XPULP and PULP_CLUSTER parameters have been renamed COREV_PULP and COREV_CLUSTER in all design and verification files.

To easily change FPU instructions latencies, 2 new parameters have been added, FPU_ADDMUL_LAT for Addition/Multiplication lane and FPU_OTHERS_LAT for the other instructions (move, conversion, comparison…).

17.3 Released core versions

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

17.3.1 cv32e40p_v1.0.0

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

For this release `mimpid` value is fixed and is equal to `0`.

It refers to the CV32E40P core verified with the following parameters:

Name	Value
`FPU`	0
`PULP_ZFINX`	0
`PULP_XPULP`	0
`PULP_CLUSTER`	0

Verification of cv32e40p_v1.0.0 has been done with only following value for `NUM_MHPMCOUNTERS` parameter: `NUM_MHPMCOUNTERS` == 1.

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/programs/blob/7a72508c90484a7835590a97038eb9dd53bd8c32/milestones/CV32E40P/RTL_Freeze_v1.0.0/Design_openissues.md
- https://github.com/openhwgroup/programs/blob/7a72508c90484a7835590a97038eb9dd53bd8c32/milestones/CV32E40P/RTL_Freeze_v1.0.0/Verification_openissues.md
- https://github.com/openhwgroup/programs/blob/7a72508c90484a7835590a97038eb9dd53bd8c32/milestones/CV32E40P/RTL_Freeze_v1.0.0/Documentation_openissues.md

17.3.2 cv32e40p_v2.0.0

Git Tag	Tagged By	Date	Reason for Release	Comment
cv32e40p_v2.0.0			RTL Freeze	

For this release `mimpid` value is depending of parameters value.

mimpid = 0

When parameters are set with the exact same values than for cv32e40p_v1.0.0 release then `mimpid` value is equal to `0`.

Name	Value
FPU	0
ZFINX	0
COREV_PULP	0
COREV_CLUSTER	0

mimpid = 1

When one parameter is set with a different value than for cv32e40p_v1.0.0 release then `mimpid` value is equal to `1`.

This means either `FPU`, `COREV_PULP` or `COREV_CLUSTER` is set to 1.

CHAPTER

EIGHTEEN

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)
- **EX**: 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