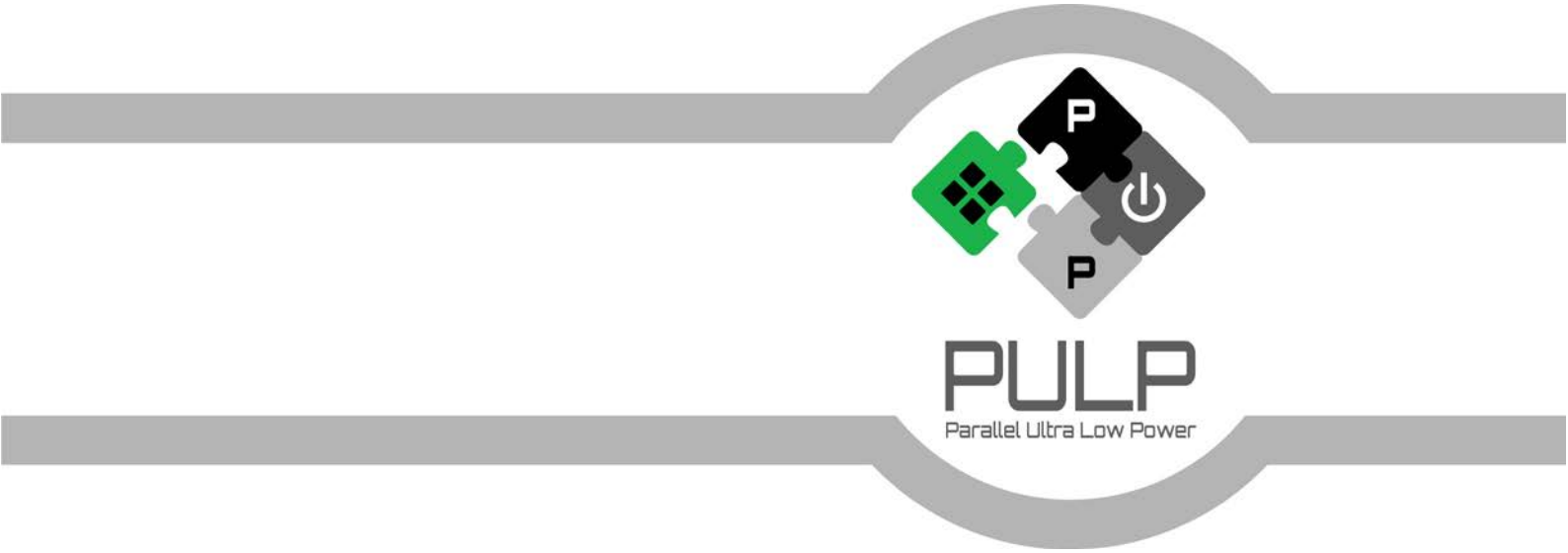


RI5CY: User Manual

May 2016
Revision 0.9

Andreas Traber (atraber@iis.ee.ethz.ch)

Michael Gautschi (gautschi@iis.ee.ethz.ch)

Micrel Lab and Multitherman Lab
University of Bologna, Italy

Integrated Systems Lab
ETH Zürich, Switzerland

Document Revisions

Rev.	Date	Author	Description
0.1	25/02/16	Andreas Traber	First Draft
0.8	13/05/16	Andreas Traber	Added instruction encoding
0.9	19/05/16	Michael Gautschi	Typos and general corrections

Table of Contents

1 Introduction

RI5CY is a 4-stage in-order 32b RISC-V processor core. The ISA of RI5CY was extended to support multiple additional instructions including hardware loops, post-increment load and store instructions and additional ALU instructions that are not part of the standard RISC-V ISA.

Figure 1 shows a block diagram of the core.

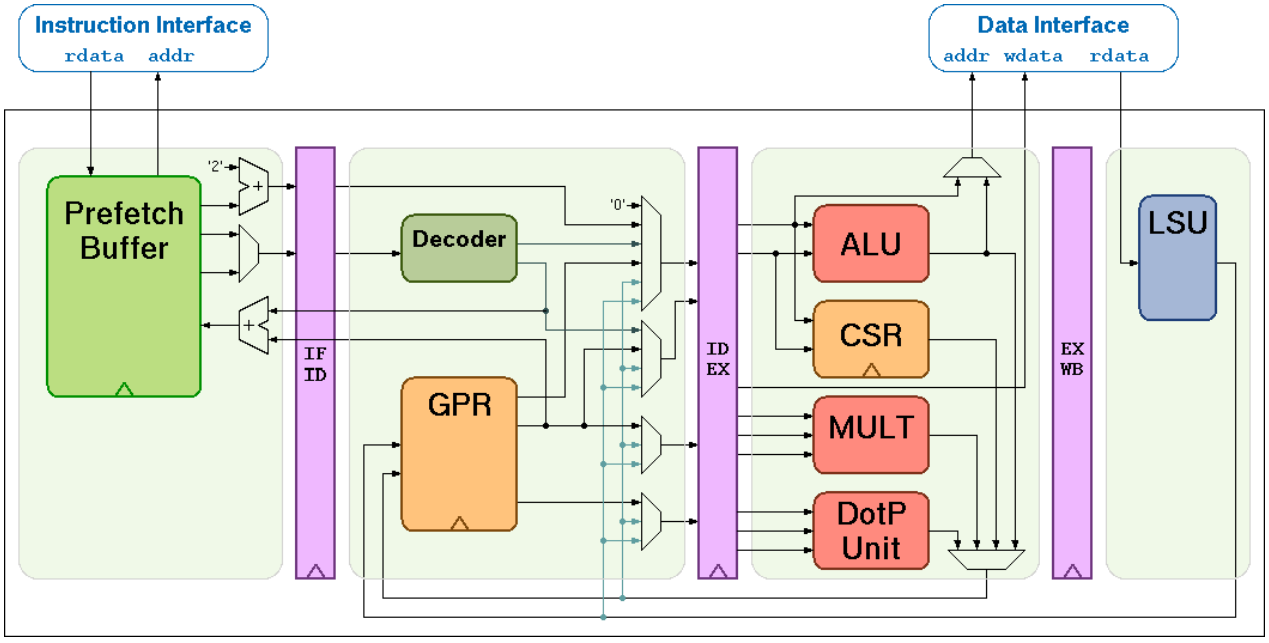


Figure 1: Block Diagram

1.1 Supported Instruction Set

RI5CY supports the following instructions:

- Full support for RV32I Base Integer Instruction Set
- Full support for RV32C Standard Extension for Compressed Instructions
- Full support for RV32M Integer Multiplication and Division Instruction Set Extension
- PULP specific extensions
    - Post-Incrementing load and stores, see Chapter 3
    - Multiply-Accumulate extensions, see Chapter 4
    - ALU extensions, see Chapter 5
    - Hardware Loops, see Chapter 6

1.2 ASIC Synthesis

ASIC synthesis is supported for RI5CY. The whole design is completely synchronous and uses positive-edge triggered flip-flops, except for the register file, which can be implemented either with latches or with flip-flops. See Chapter 8 for more details about the register file. The core occupies an area of about 50 kGE when the latch based register file is used.

1.3 FPGA Synthesis

FPGA synthesis is supported for RI5CY when the flip-flop based register file is used. Since latches are not well supported on FPGAs, it is crucial to select the flip-flop based register file.

1.4 Outline

This document summarizes all the functionality of the Ri5CY core in more detail. First, the instruction and data interfaces are explained in Chapter 2 and 3. The multiplier as well as the ALU are then explained in Chapter 4 and 5. Chapter 6 focuses on the hardware loop extensions and Chapter 8 explains the register file. Control and status registers are explained in Chapter 9 and Chapter 10 gives an overview of all performance counters. Chapter 11 deals with exceptions and interrupts, and Chapter 12 summarizes the accessible debug registers. Finally, Chapter 13 gives an overview of all instruction-extensions, its encodings and meanings.

2 Instruction Fetch

The instruction fetcher of the core is able to supply one instruction to the ID stage per cycle if the instruction cache or the instruction memory is able to serve one instruction per cycle. The instruction address must be half-word-aligned due to the support of compressed instructions. It is not possible to jump to instruction addresses that have the LSB bit set.

For optimal performance and timing closure reasons, a prefetcher is used which fetches instruction from the instruction memory, or instruction cache.
There are two prefetch flavors available:

- 32-Bit word prefetcher. It stores the fetched words in a FIFO with three entries.
- 128-Bit cache line prefetcher. It stores one 128-bit wide cache line plus 32-bit to allow for cross-cache line misaligned instructions.

Table 1 describes the signals that are used to fetch instructions. This interface is a simplified version that is used by the LSU that is described in Chapter 3. The difference is that no writes are possible and thus it needs less signals.

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

Table 1: Instruction Fetch Signals

2.1 Protocol

The protocol used to communicate with the instruction cache or the instruction memory is the same as the protocol used by the LSU. See the description of the LSU in Chapter 3.2 for details about the protocol.

3 Load-Store-Unit (LSU)

The LSU of the core takes care of accessing the data memory. Load and stores on words (32 bit), half words (16 bit) and bytes (8 bit) are supported.
Table 2 describes the signals that are used by the LSU.

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

Table 2: LSU Signals

3.1 Misaligned Accesses

The LSU is able to perform misaligned accesses, meaning accesses that are not aligned on natural word boundaries. However, it needs to perform two separate word-aligned accesses internally. This means that at least two cycles are needed for misaligned loads and stores.

3.2 Protocol

The protocol that is used by the LSU to communicate with a memory works as follows:
The LSU provides a valid address in data_addr_o and sets data_req_o high. The memory then answers with a data_gnt_i set high as soon as it is ready to serve the request. This may happen in the same cycle as the request was sent or any number of cycles later. After a grant was received, the address may be changed in the next cycle by the LSU. In addition, the data_wdata_o, data_we_o and data_be_o signals may be changed as it is assumed that the memory has already processed and stored that information. After receiving a grant, 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 grant has been received. Note that data_rvalid_i must also be set when a write was performed, although the data_rdata_i has no meaning in this case.

Figure 3, Figure 4 and Figure 5 show example-timing diagrams of the protocol.

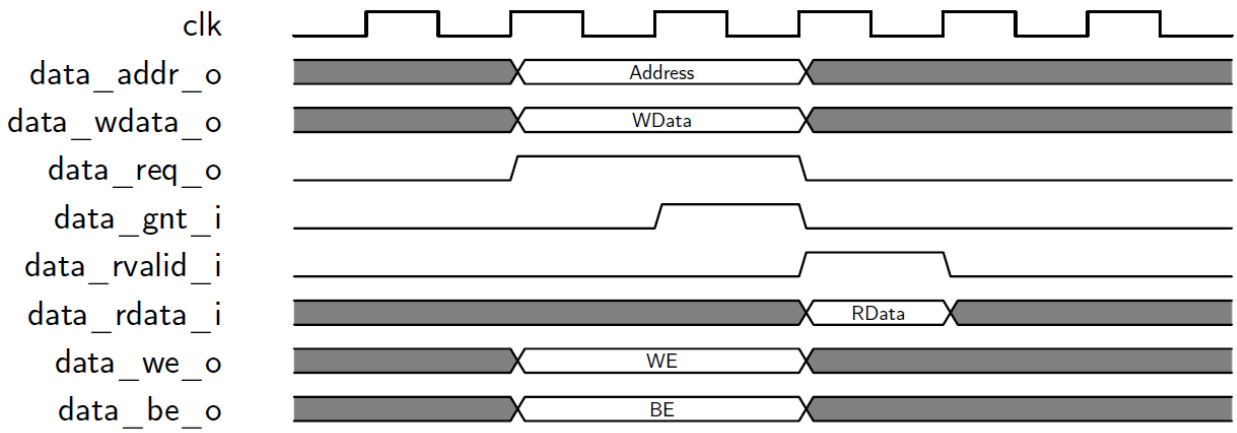


Figure 2: Basic Memory Transaction

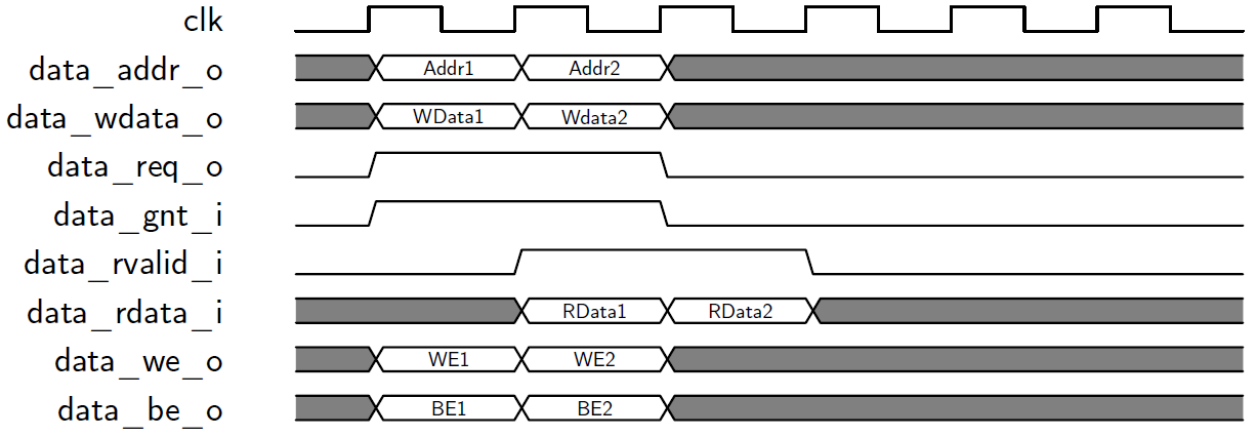


Figure 3: Back-to-back Memory Transaction

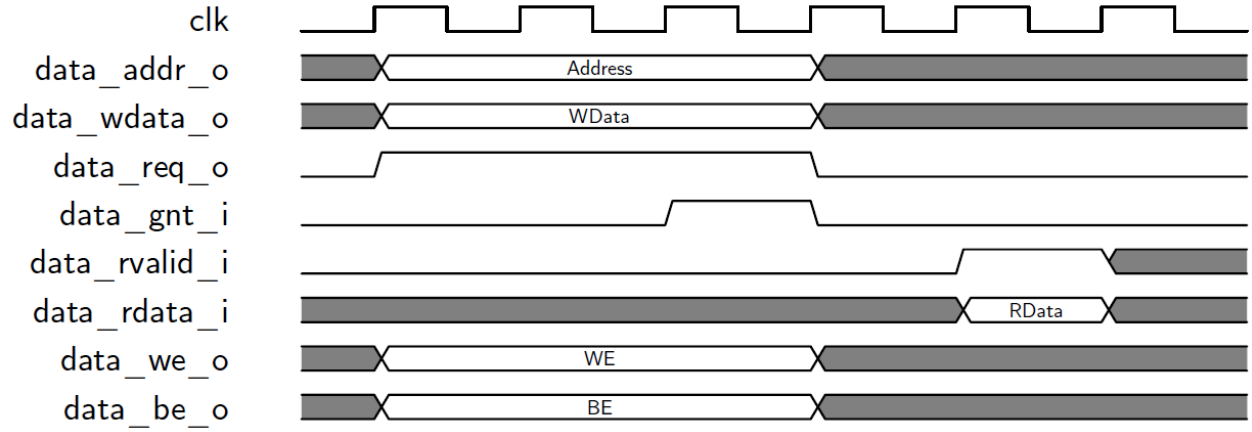


Figure 4: Slow Response Memory Transaction

3.3 Post-Incrementing Load and Store Instructions

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, this instructions allow to reduce the loop overhead significantly.

4 Multiply-Accumulate

RI5CY uses a single-cycle 32-bit x 32-bit multiplier with a 32-bit result. All instructions of the RISC-V M instruction set extension are supported.
The multiplications with upper-word result (MSP of 32-bit x 32-bit multiplication), take 4 cycles to compute. The division and remainder instructions take between 2 and 32 cycles. The number of cycles depends on the operand values.

Additionally, RI5CY supports non-standard extensions for multiply-accumulate and half-word multiplications with an optional post-multiplication shift.

5 PULP ALU Extensions

RI5CY 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.

6 PULP Hardware Loop Extensions

To increase the efficiency of small loops, RI5CY supports hardware loops. Hardware loops make it possible to execute a piece of code multiple times, without the overhead of branches or updating a counter. Hardware loops involve zero stall cycles for jumping to the first instruction of a loop.

A hardware loop is defined by its start address (pointing to the first instruction in the loop), its end address (pointing to the instruction that will be executed last in the loop) and a counter that is decremented every time the loop body is executed. RI5CY 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.
If the end address of the two hardware loops is identical, loop 0 has higher priority and only the loop counter for hardware loop 0 is decremented. As soon as the counter of loop 0 reaches 1 at an end address, meaning it is decremented to 0 now, loop 1 gets active too. In this case, both counters will be decremented and the core jumps to the start of loop 1.

In order to use hardware loops, the compiler needs to setup the loop beforehand with the following instructions. Note that the minimum loop size is two instructions.

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

6.1 CSR Mapping

CSR Address				Hex	Name	Acc.	Description
11:10	9:8	7:6	5:0				
01	11	10	110000	0x7B0	lpstart[0]	R/W	Hardware Loop 0 Start
01	11	10	110001	0x7B1	lpendt[0]	R/W	Hardware Loop 0 End
01	11	10	110010	0x7B2	lpcount[0]	R/W	Hardware Loop 0 Counter
01	11	10	110000	0x7B4	lpstart[1]	R/W	Hardware Loop 0 Start
01	11	10	110001	0x7B5	lpend[1]	R/W	Hardware Loop 1 End
01	11	10	110010	0x7B6	lpcount[1]	R/W	Hardware Loop 1 Counter

Table 3: Hardware-Loop CSR Mapping

Pipeline

RI5CY has a fully independent pipeline, meaning that whenever possible data will propagate through the pipeline and therefor does not suffer from any unneeded stalls.

The pipeline design is easily extendable to incorporate out-of-order completion. E.g., it would be possible to complete an instruction that only needs the EX stage before the WB stage, that is currently blocked waiting for an rvalid, is ready. Currently this is not done in RI5CY, but might be added in the future.

Figure 6 shows the relevant control signals for the pipeline operation. The main control signals, the ready signals of each pipeline stage, are propagating from right to left. Each pipeline stage has two control inputs: an enable and a clear. The enable activates the pipeline stage and the core moves forward by one instruction. The clear removes the instruction from the pipeline stage as it is completed. Every pipeline stage is cleared if the ready coming from the stage to the right is high, and the valid signal of the stage is low. If the valid signal is high, it is enabled.

Every pipeline stage is independent of its left neighbor, meaning that it can finish its execution no matter if a stage to its left is currently stalled or not. On the other hand, an instruction can only propagate to the next stage if the stage to its right is ready to receive a new instruction. This means that in order to process an instruction in a stage, its own stage needs to be ready and so does its right neighbor.

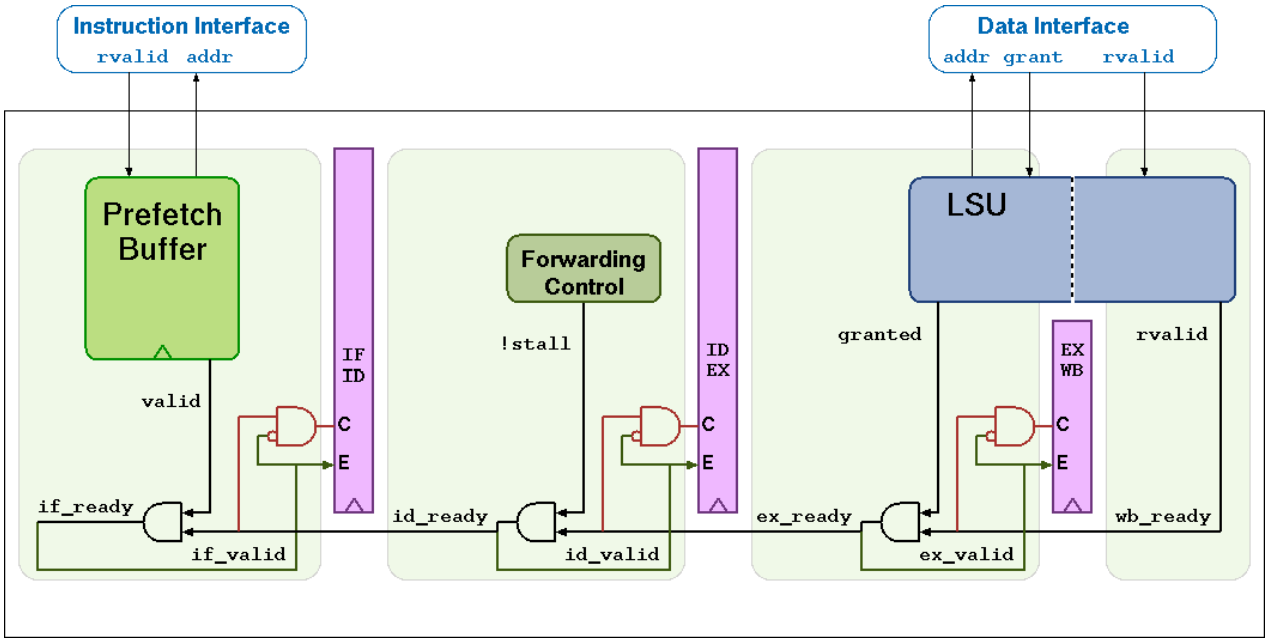


Figure 5: RI5CY Pipeline

7 Register File

RI5CY 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.

There are two flavors of register file available:

1. Latch-based
2. Flip-flop based

While the latch-based register file is recommended for ASICs, the flip-flop based register file is recommended for FPGA synthesis, although both are compatible with either synthesis target. Note the flip-flop based register file is significantly larger than the latch-based register-file for an ASIC implementation.

7.1 Latch-based Register File

The latch based register file contains manually instantiated clock gating cells to keep the clock inactive when the latches are not written.

It is assumed that there is a clock gating cell for the target technology that is wrapped in a module called cluster_clock_gating and has the following ports:

- clk_i: Clock Input
- en_i: Clock Enable Input
- test_en_i: Test Enable Input (activates the clock even though en_i is not set)
- clk_o: Gated Clock Output

8 Control and Status Registers

RI5CY 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.

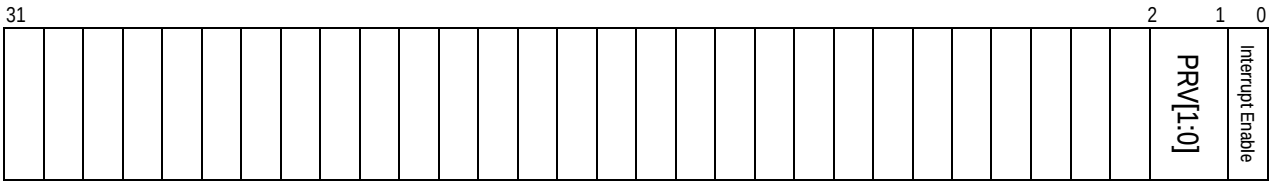
CSR Address				Hex	Name	Acc.	Description
11:10	9:8	7:6	5:0				
00	11	00	000000	0x300	MSTATUS	R/W	Machine Status
00	11	01	000001	0x341	MEPC	R/W	Machine Exception Program Counter
00	11	01	000010	0x342	MCAUSE	R/W	Machine Trap Cause
01	11	00	0xxxxx	0x780-0x79F	PCCRs	R/W	Performance Counter Counter Registers
01	11	10	100000	0x7A0	PCER	R/W	Performance Counter Enable
01	11	10	100001	0x7A1	PCMR	R/W	Performance Counter Mode
01	11	10	110xxx	0x7B0-0x7B7	HWLP	R/W	Hardware Loop Registers
01	11	10	111000	0x7C0	MESTATUS	R/W	Machine Exception Status
11	11	00	000000	0xF00	MCPUID	R	CPU Description
11	11	00	000001	0xF01	MIMPID	R	Vendor ID and version number
11	11	00	010000	0xF10	MHARTID	R	Hardware Thread ID

Table 4: Control and Status Register Map

8.1 Machine Status (MSTATUS)

CSR Address: 0x300

Reset Value: 0x0000_0006



Detailed:

Bit #	R/W	Description
2:1	R	PRV: Statically 2'b11 and cannot be altered (read-only).

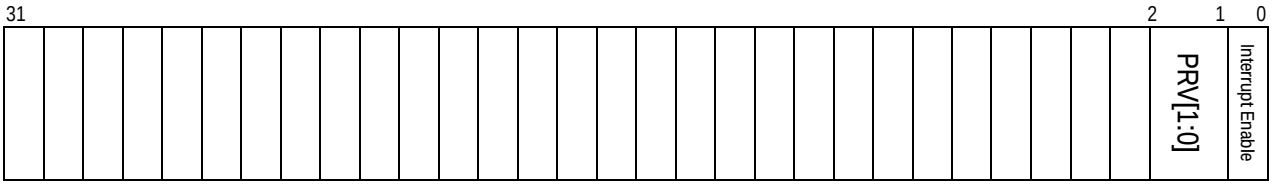
Bit #	R/W	Description
0	R/W	Interrupt Enable: When an exception is encountered, Interrupt Enable will be set to 1'b0. When the eret instruction is executed, the original value of Interrupt Enable will be restored, as MESTATUS will replace MSTATUS. If you want to enable interrupt handling in your exception handler, set the Interrupt Enable to 1'b1 inside your handler code.

Table 5: MSTATUS

8.2 Machine Exception Status (MESTATUS)

CSR Address: 0x7C0

Reset Value: 0x0000_0006



Detailed:

Bit #	R/W	Description
2:1	R	PRV: Statically 2'b11 and cannot be altered (read-only).
0	R/W	Interrupt Enable: When an exception is encountered, the current value of MSTATUS is saved in MESTATUS. When an eret instruction is executed, the value from MESTATUS replaces the MSTATUS register.

Table 6: MESTATUS

8.3 Machine Exception PC (MEPC)

CSR Address: 0x341

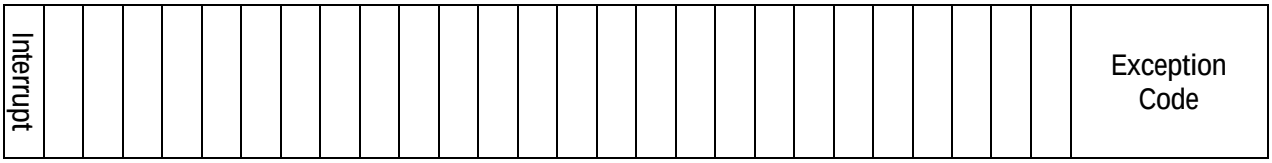
Reset Value: 0x0000_0000

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

8.4 Machine Cause (MCAUSE)

CSR Address: 0x342

Reset Value: 0x0000_0000



Detailed:

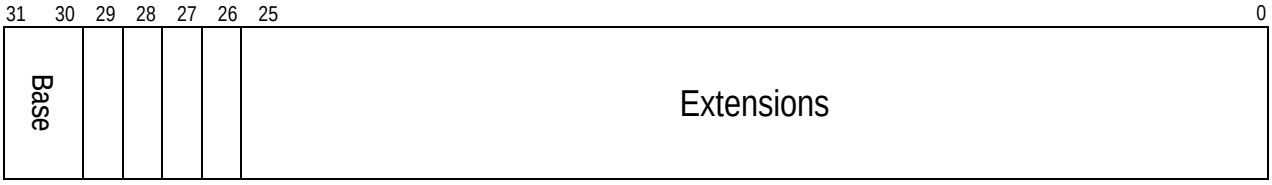
Bit #	R/W	Description
31	R	**Interrupt:** This bit is set when the exception was triggered by an interrupt.
4:0	R	**Exception Code**

Table 7: MCAUSE

8.5 MCPUID

CSR Address: 0xF00

Reset Value: 0x0080_1100



Detailed:

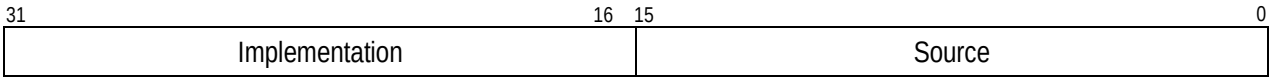
Bit #	R/W	Description
31:30	R	**Base:** Reads as 0 which means RV32I
25:0	R	**Extensions:** RI5CY only supports the I and M extensions, plus the RI5CY non-standard extensions. This means bits 8 (I), 12 (M) and 23 (X) are 1, the rest is 0.

Table 8: MCPUID

8.6 MIMPID

CSR Address: 0xF01

Reset Value: 0x0000_8000



Detailed:

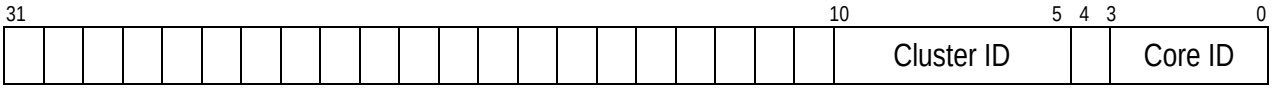
Bit #	R/W	Description
31:16	R	**Implementation:** Reads as 0
15:0	R	**Source:** Reads as 0x8000

Table 9: MIMPID

8.7 MHARTID

CSR Address: 0xF10

Reset Value: Defined



Detailed:

Bit #	R/W	Description
10:5	R	**Cluster ID:** ID of the cluster
3:0	R	**Core ID:** ID of the core within the cluster

Table 10: MHARTID

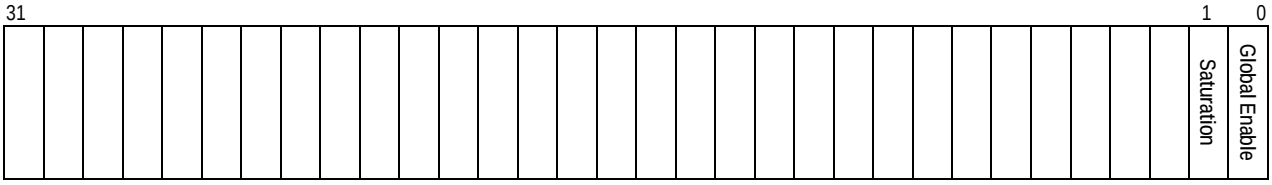
9 Performance Counters

Performance Counters in RI5CY are placed inside the Control and Status Registers and can be accessed with csrr and csrw instructions. See Table 9.1 for the address map of the performance counter registers

9.1 Performance Counter Mode Register (PCMR)

CSR Address: 0x7A1

Reset Value: 0x0000_0003



Detailed:

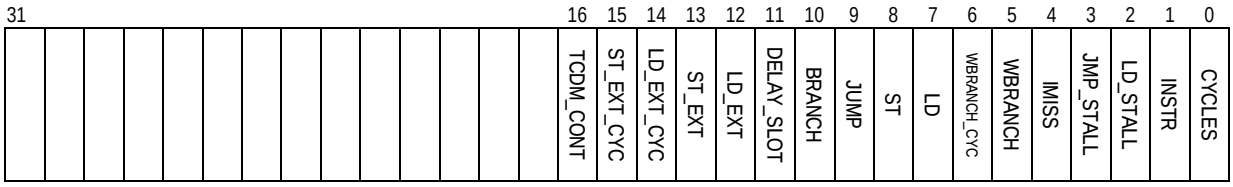
Bit #	R/W	Description
1	R/W	Global Enable: Activate/deactivate all performance counters. If this bit is 0, all performance counters are disabled. After reset, this bit is set.
0	R/W	Saturation: If this bit is set, saturating arithmetic is used in the performance counter counters. After reset, this bit is set.

Table 11: PCMR

9.2 Performance Counter Event Register (PCER)

CSR Address: 0x7A0

Reset Value: 0x0000_0000



Detailed:

Bit #	R/W	Description
16	R/W	TCDM_CONT
15	R/W	ST_EXT_CYC
14	R/W	LD_EXT_CYC

Bit #	R/W	Description
13	R/W	ST_EXT
12	R/W	LD_EXT
11	R/W	DELAY_SLOT
10	R/W	BRANCH
9	R/W	JUMP
8	R/W	ST
7	R/W	LD
6	R/W	WBRANCH_CYC
5	R/W	WBRANCH
4	R/W	IMISS
3	R/W	JMP_STALL
2	R/W	LD_STALL
1	R/W	INSTR
0	R/W	CYCLES

Table 12: PCER

Each bit in the PCER register controls one performance counter. If the bit is 1, the counter is enabled and starts counting events. If it is 0, the counter is disabled and its value won't change.

In the ASIC there is only one counter register, thus all counter events are masked by PCER and ORed together, i.e. if one of the enabled event happens, the counter will be increased. If multiple non-masked events happen at the same time, the counter will only be increased by one.
In order to be able to count separate events on the ASIC, the program can be executed in a loop with different events configured.

In the FPGA or RTL simulation version, each event has its own counter and can be accessed separately.

9.3 Performance Counter Counter Register (PCCR0-31)

CSR Address: 0x780 - 0x79F

Reset Value: 0x0000_0000

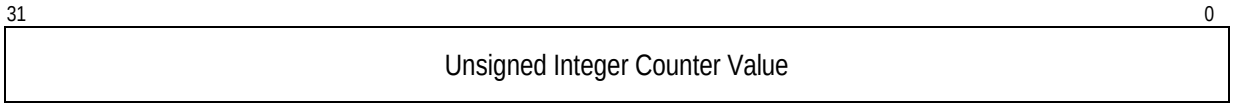


Table 13: PCCR0-31

PCCR registers support both saturating and wrap-around arithmetic. This is controlled by the saturation bit in PCMR.

Register	Name	Description
PCCR0	CYCLES	Counts the number of cycles the core was active (not sleeping)
PCCR1	INSTR	Counts the number of instructions executed
PCCR2	LD_STALL	Number of load data hazards
PCCR3	JR_STALL	Number of jump register data hazards
PCCR4	IMISS	Cycles waiting for instruction fetches, i.e. number of instructions wasted due to non-ideal caching
PCCR5	LD	Number of data memory loads executed. Misaligned accesses are counted twice
PCCR6	ST	Number of data memory stores executed. Misaligned accesses are counted twice
PCCR7	JUMP	Number of unconditional jumps (j, jal, jr, jalr)
PCCR8	BRANCH	Number of branches. Counts taken and not taken branches
PCCR9	BTAKEN	Number of taken branches.
PCCR10	RVC	Number of compressed instructions executed
PCCR11	LD_EXT	Number of memory loads to EXT executed. Misaligned accesses are counted twice. Every non-TCDM access is considered external (PULP only)
PCCR12	ST_EXT	Number of memory stores to EXT executed. Misaligned accesses are counted twice. Every non-TCDM access is considered external (PULP only)
PCCR13	LD_EXT_CYC	Cycles used for memory loads to EXT. Every non-TCDM access is considered external (PULPY only)
PCCR14	ST_EXT_CYC	Cycles used for memory stores to EXT. Every non-TCDM access is considered external (PULPY only)
PCCR15	TCDM_CONT	Cycles wasted due to TCDM/log-interconnect contention (PULPY only)
PCCR31	ALL	Special Register, a write to this register will set all counters to the supplied value

Table 14: PCCR Definitions

In the FPGA, RTL simulation and Virtual-Platform there are individual counters for each event type, i.e. PCCR0-30 each represent a separate register. To save area in the ASIC, there is only one counter and one counter register. Accessing PCCR0-30 will access the same counter register in the ASIC. Reading/writing from/to PCCR31 in the ASIC will access the same register as PCCR0-30.

Figure 7 shows how events are first masked with the PCER register and then ORed together to increase the one performance counter PCCR.

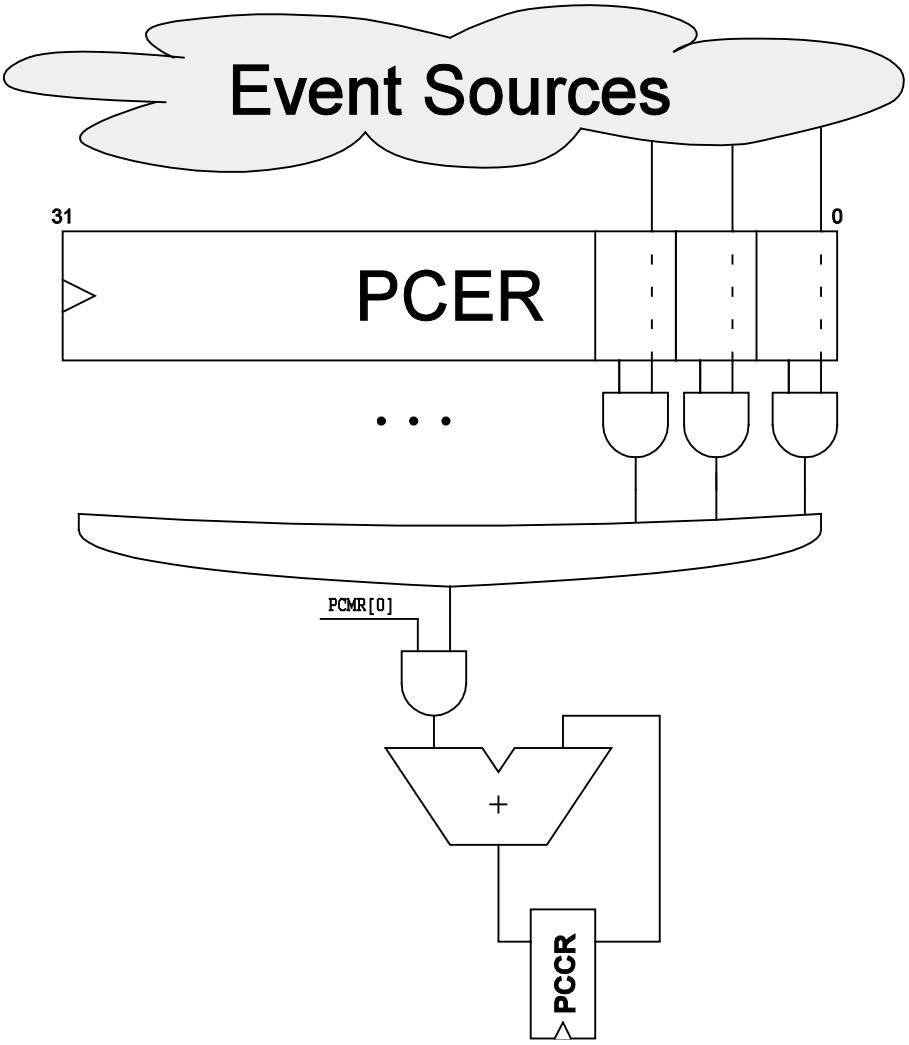


Figure 6: Events and PCCR, PCMR and PCER on the ASIC.

10 Exceptions and Interrupts

RI5CY supports vectorized interrupts, exceptions on illegal instructions and exceptions on load and store instructions to invalid addresses.

Address	Description
0x00-0x7C	Interrupts 0 - 31
0x80	Reset
0x84	Illegal Instruction
0x88	ECALL Instruction Executed
0x8C	LSU Error (Invalid Memory Access)

Table 15: Interrupt/Exception Offset Vector Table

The base address of the interrupt vector table is given by the boot address. The boot address given to the core is used for the first instruction fetch of the core and as the basis of the interrupt vector table. The boot address can be changed after the first instruction was fetched to change the interrupt vector table address. It is assumed that the boot address is supplied via a register to avoid long paths to the instruction fetch unit.

10.1 Interrupts

RI5CY uses vectorized interrupts; specifically it provides 32 separate interrupt lines. Interrupts can only be enabled/disabled on a global basis and not individually. It is assumed that there is an event/interrupt controller outside of the core that performs masking and buffering of the interrupt lines. The global interrupt enable is done via the CSR register MSTATUS.

If multiple interrupts arrive at the same cycle, the interrupt with the lowest number will be executed first. As soon as IRQs are re-enabled, either after an eret or after an explicit enable, the interrupt with lowest number will be executed. This means that it is important to clear the interrupt line before re-enabling interrupts, as otherwise the same interrupt handler could be called over and over again.

10.2 Exceptions

The illegal instruction exception, the load and store invalid memory access exceptions and ecall instruction exceptions cannot be disabled and are always active.
The illegal instruction exception and the load and store invalid memory access exceptions are precise exceptions, i.e. the value of MEPC will be the instruction address that caused it.

10.3 Handling

RI5CY does support nested interrupt/exception handling. Exceptions inside interrupt/exception handlers cause another exception, thus exceptions during the critical part of your exception
handlers, i.e. before having saved the MEPC and MESTATUS registers, will cause those register to be overwritten. Interrupts during interrupt/exception handlers are disabled by default, but can be explicitly enabled if desired.

Upon executing an eret instruction, the core jumps to the program counter saved in the CSR register MEPC and restores the value of register MESTATUS to MSTATUS. When entering an interrupt/exception handler, the core sets MEPC to the current program counter and saves the current value of MSTATUS in MESTATUS.

11 Debug Unit

11.1 Address Map

Address	Name	Description
0x0000-0x007F	Debug Registers	Always accessible, even when the core is running
0x400-0x47F	GPR (x0-x31)	General Purpose Registers Only accessible if the core is halted
0x500-0x5FF	FPR (f0-f31)	Reserved. Not used in the RI5CY core. First LSP from 0x500-0x57F, then MSP from 0x580-0x5FF
0x2000-0x20FF	Debug Registers	Only accessible if the core is halted
0x4000-0x7FFF	CSR	Control and Status Registers Only accessible if the core is halted

Table 16: Debug Unit Address Map

Addresses are intended for a bus system with 32-bit wide words.
FPR get more address space than GPR because they can be 64-bit wide even in a 32-bit system.
Addresses have to be aligned to word-boundaries.

11.2 Debug Registers

Address	Name	Description
0x00	DBG_CTRL	Debug Control
0x04	DBG_HIT	Debug Hit
0x08	DBG_IE	Debug Interrupt Enable
0x0C	DBG_CAUSE	Debug Cause (Why we entered debug state)
0x40	DBG_BPCTRL0	HW BP0 Control
0x44	DBG_BPDATA0	HW BP0 Data
0x48	DBG_BPCTRL1	HW BP1 Control
0x4C	DBG_BPDATA1	HW BP1 Data
0x50	DBG_BPCTRL2	HW BP2 Control
0x54	DBG_BPDATA2	HW BP2 Data
0x58	DBG_BPCTRL3	HW BP3 Control
0x5C	DBG_BPDATA3	HW BP3 Data
0x60	DBG_BPCTRL4	HW BP4 Control

Address	Name	Description
0x64	DBG_BPDATA4	HW BP4 Data
0x68	DBG_BPCTRL5	HW BP5 Control
0x6C	DBG_BPDATA5	HW BP5 Data
0x70	DBG_BPCTRL6	HW BP6 Control
0x74	DBG_BPDATA6	HW BP6 Data
0x78	DBG_BPCTRL7	HW BP7 Control
0x7C	DBG_BPDATA7	HW BP7 Data
0x2000	DBG_NPC	Next PC
0x2004	DBG_PPC	Previous PC

Table 17: Debug Unit Registers

11.2.1 Debug Control (DBG_CTRL)

Compact:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															HALT
															R/W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved															SSTE
															R/W

Detailed:

Bit #	R/W	Description
16	W1	HALT: When 1 written, core enters debug mode, when 0 written, core exits debug mode. When read, 1 means core is in debug modekkj
0	R/W	SSTE: Single-step enable

Table 18: DBG_CTRL register

11.2.2 Debug Hit (DBG_HIT)

Compact:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															SLEEP
															R

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved															SSTH
															R/W

Detailed:

Bit #	R/W	Description
16	R	SLEEP: Set when the core is in a sleeping state and waits for an event
0	R/W	SSTH: Single-step hit, sticky bit that must be cleared by external debugger

Table 19: DBG_HIT register

11.2.3 Debug Interrupt Enable (DBG_IE)

Compact:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TO BE DEFINED															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved				ECALL	reserved			SAF	SAM	LAF	LAM	BP	ILL	IAF	IAM
				R/W				R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Detailed:

Bit #	R/W	Description
11	R/W	ECALL: Environment call from M-Mode
7	R/W	SAF: Store Access Fault (together with LAF)
6	R/W	SAM: Store Address Misaligned (never traps)
5	R/W	LAF: Load Access Fault (together with SAF)
4	R/W	LAM: Load Address Misaligned (never traps)
3	R/W	BP: EBREAK instruction causes trap
2	R/W	ILL: Illegal Instruction
1	R/W	IAF: Instruction Access Fault (not implemented)
0	R/W	IAM: Instruction Address Misaligned (never traps)

Table 20: DBG_IE register

When ‘1’ exceptions cause traps, otherwise normal exceptions.

11.2.4 Debug Cause (DBG_CAUSE)

Compact:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IRQ	reserved														
R															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

reserved	CAUSE
	R

Detailed:

Bit #	R/W	Description
31	R	IRQ: Interrupt caused us to enter debug mode
4:0	R	CAUSE: Exception/interrupt number

Table 21: DBG_CAUSE register

11.2.5 Debug Hardware Breakpoint x Control (DBG_BPCTRLx)

Compact:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved															IMPL
															R0

Detailed:

Bit #	R/W	Description
0	R	IMPL: RI5CY does not implement hardware breakpoints. Always read as 0.

Table 22: DBG_BPCTRLx register

11.2.6 Debug Next Program Counter (DBG_NPC)

Compact:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
NPC[31:16]															
R/W															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NPC[15:0]															
R/W															

Detailed:

Bit #	R/W	Description
31:0	R/W	NPC: Next PC to be executed

Table 23: DBG_NPC register

When written core jumps to PC.

11.2.7 Debug Previous Program Counter (DBG_PPC)

Compact:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PPC[31:16]															
R															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PPC[15:0]															
R															

Detailed:

Bit #	R/W	Description
31:0	W	PPC: Previous PC, already executed

Table 24: DBG_PPC register

Values of PPC and NPC when entering debug mode:

Reason	PPC	NPC	Cause	GDB Sigval
ebreak	ebreak instruction	next instruction	BP	TRAP
ecall	ecall instruction	IVT entry	ECALL	TRAP
illegal instruction	illegal instruction	IVT entry	ILL	ILL
invalid mem access	load/store instruction	IVT entry	LAF/SAF	SEGV
interrupt	last instruction	IVT entry	?	INT
halt	last instruction	next instruction	?	TRAP
single-step	last instruction	next instruction	?	TRAP

Table 25: NPC/PPC when entering Debug Mode

11.3 Control and Status Registers

Address	Name	Description
0x4000	CSR 0 = 0x000	CSR
...	...	...
0x7FFC	CSR 4095 = 0xFFF	CSR

Table 26: Debug CSR Mapping

Can only be accessed when core is in debug mode.

11.4 Interface

Signal	Direction	Description
debug_req_i	input	Request
debug_gnt_o	output	Grant
debug_rvalid_o	output	Read data valid
debug_addr_i[14:0]	input	Address for write/read
debug_we_i	input	Write Enable
debug_wdata_i[31:0]	input	Write data
debug_rdata_o[31:0]	output	Read data
debug_halted_o	output	Is high when core is in debug mode
debug_halt_i	input	Set high when core should enter debug mode
debug_resume_i	input	Set high when core should exit debug mode

Table 27: Debug Interface

debug_halted_o, debug_halt_i and debug_resume_i are intended for cross-triggering between multiple cores. They are not required for single-core debug, thus debug_halt_i and debug-resume_i can be tied to 0.

debug_halt_i and debug_resume_i should be high for only one single cycle to avoid deadlock issues.

12 Instruction Set Extensions

12.1 Post-Incrementing Load & Store Instructions

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

12.1.1.1Load Operations

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

Mnemonic	Description
p.lh rD, rs2(rs1)	rD = Sext(Mem16(rs1 + rs2))
p.lhu rD, rs2(rs1)	rD = Zext(Mem16(rs1 + rs2))
p.lw rD, rs2(rs1)	rD = Mem32(rs1 + rs2)

12.1.1.2Store Operations

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

12.1.2 Encoding

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

offset	base	010	dest	000 1011	p.lw rD, Imm(rs1!)

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	
000 0000	offset	base	111	dest	000 1011	p.lb rD, rs2(rs1!)
010 0000	offset	base	111	dest	000 1011	p.lbu rD, rs2(rs1!)
000 1000	offset	base	111	dest	000 1011	p.lh rD, rs2(rs1!)
010 1000	offset	base	111	dest	000 1011	p.lhu rD, rs2(rs1!)
001 0000	offset	base	111	dest	000 1011	p.lw rD, rs2(rs1!)

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	
000 0000	offset	base	111	dest	000 0011	p.lb rD, rs2(rs1)
010 0000	offset	base	111	dest	000 0011	p.lbu rD, rs2(rs1)
000 1000	offset	base	111	dest	000 0011	p.lh rD, rs2(rs1)
010 1000	offset	base	111	dest	000 0011	p.lhu rD, rs2(rs1)
001 0000	offset	base	111	dest	000 0011	p.lw rD, rs2(rs1)

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

31	20 19	15 14	12 11	7 6	0	
funct7	rs2	rs1	funct3	rs3	opcode	
000 0000	src	base	100	offset	010 1011	p.sb rs2, rs3(rs1!)
000 0000	src	base	101	offset	010 1011	p.sh rs2, rs3(rs1!)
000 0000	src	base	110	offset	010 1011	p.sw rs2, rs3(rs1!)

31	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rs3	opcode

000 0000	src	base	100	offset	010 0011	p.sb rs2, rs3(rs1)
000 0000	src	base	101	offset	010 0011	p.sh rs2, rs3(rs1)
000 0000	src	base	110	offset	010 0011	p.sw rs2, rs3(rs1)

12.2 Hardware Loops

RI5CY supports 2 levels of nested hardware loops. The loop has to be setup before entering the loop body. For this purpose, there are two methods, either the long commands that separately set start- and end-addresses of the loop and the number of iterations, or the short command that does all of this in a single instruction. The short command has a limited range for the number of instructions contained in the loop and the loop must start in the next instruction after the setup instruction.

Loop number 0 has higher priority than loop number 1 in a nested loop configuration, meaning that loop 0 represents the inner loop.

A hardware loop is subject to the following constraints:

- Minimum of 2 instructions in the loop body.
- Loop counter has to be bigger than 0, since the loop body is always entered at least once.

12.2.1 Operations

Mnemonic		Description
Long Hardware Loop Setup instructions		
lp.starti	L, uimmL	lpstart[L] = PC + (uimmL << 1)
lp.endi	L, uimmL	lpend[L] = PC + (uimmL << 1)
lp.count	L, rs1	lpcount[L] = rs1
lp.counti	L, uimmL	lpcount[L] = uimmL
Short Hardware Loop Setup Instructions		
lp.setup	L, rs1, uimmL	lpstart[L] = pc + 4 lpend[L] = pc + (uimmL << 1) lpcount[L] = rs1
lp.setupi	L, uimmS, uimmL	lpstart[L] = pc + 4 lpend[L] = pc + (uimmS << 1) lpcount[L] = uimmL

12.2.2 Encoding

31 20	19 15	14 12	11 10	7	6 0	
uimmL[11:0]	rs1	funct3	0000	L	opcode	
uimmL[11:0]	00000	000	0000	L	111 1011	lp.starti L, uimmL
uimmL[11:0]	00000	001	0000	L	111 1011	lp.endi L, uimmL
0000 0000 0000	src1	010	0000	L	111 1011	lp.count L, rs1
uimmL[11:0]	00000	011	0000	L	111 1011	lp.counti L, uimmL
uimmL[11:0]	src1	100	0000	L	111 1011	lp.setup L, rs1, uimmL
uimmL[11:0]	uimmS[4:0]	101	0000		111 1011	lp.setupi L, uimmS, uimmL

12.3 ALU

The ALU extensions 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 Section 13.3.1.
- 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 Section 13.3.3

12.3.1 Bit Manipulation Operations

Mnemonic		Description
p.extract	rD, rs1, ls3, ls2	rD = Sext((rs1 & ((1 << ls3) – 1) << ls2) >> ls2) Note: ls3 + ls2 must be <= 32
p.extractu	rD, rs1, ls3, ls2	rD = Zext((rs1 & ((1 << ls3) – 1) << ls2) >> ls2) Note: ls3 + ls2 must be <= 32
p.insert	rD, rs1, ls3, ls2	rD[2*L:L] = rs1[L:0] Note: ls3 + ls2 must be <= 32, the rest of the bits of rD are passed through and are not modified
p.bclr	rD, rs1, ls3, ls2	rD = rs1 & (~((1 << ls3) – 1) << ls2)) Note: ls3 + ls2 must be <= 32
p.bset	rD, rs1, ls3, ls2	rD = rs1 | ((1 << ls3) – 1) << ls2) Note: ls3 + ls2 must be <= 32
p.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.
p.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.
p.clb	rD, rs1	rD = count leading bits of rs1 Note: This is the number of consecutive 1's or 0's from MSB. Note: If rs1 is 0, rD will be 0.
p.cnt	rD, rs1	rD = Population count of rs1, i.e. number of bits set in rs1
p.ror	rD, rs1, rs2	rD = RotateRight(rs1, rs2)

12.3.2 Bit Manipulation Encoding

31	30 29	25 24	20 19	15 14	12 11	7 6	0	
f2	ls3[4:0]	ls2[4:0]	rs1	funct3	rD	opcode		
11	Luimm5[4:0]	luimm5[4:0]	src	000	dest	011 0011		p.extract rD, rs1, ls3, ls2
11	Luimm5[4:0]	luimm5[4:0]	src	001	dest	011 0011		p.extractu rD, rs1, ls3, ls2
11	Luimm5[4:0]	luimm5[4:0]	src	010	dest	011 0011		p.insert rD, rs1, ls3, ls2

11	Luimm5[4:0]	luimm5[4:0]	src	011	dest	011 0011	p.bclr	rD, rs1, ls3, ls2
11	Luimm5[4:0]	luimm5[4:0]	src	100	dest	011 0011	p.bset	rD, rs1, ls3, ls2

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

12.3.3 General ALU Operations

Mnemonic		Description
p.abs	rD, rs1	rD = rs1 < 0 ? –rs1 : rs1
p.slet	rD, rs1, rs2	rD = rs1 <= rs2 ? 1 : 0 Note: Comparison is signed
p.sletu	rD, rs1, rs2	rD = rs1 <= rs2 ? 1 : 0 Note: Comparison is unsigned
p.min	rD, rs1, rs2	rD = rs1 < rs2 ? rs1 : rs2 Note: Comparison is signed
p.minu	rD, rs1, rs2	rD = rs1 < rs2 ? rs1 : rs2 Note: Comparison is unsigned
p.max	rD, rs1, rs2	rD = rs1 < rs2 ? rs2 : rs1 Note: Comparison is signed
p.maxu	rD, rs1, rs2	rD = rs1 < rs2 ? rs2 : rs1 Note: Comparison is unsigned
p.exths	rD, rs1	rD = Sext(rs1[15:0])
p.exthz	rD, rs1	rD = Zext(rs1[15:0])
p.extbs	rD, rs1	rD = Sext(rs1[7:0])
p.extbz	rD, rs1	rD = Zext(rs1[7:0])
p.clip	rD, rs1, ls2	if rs1 <= -2^ls2, rD = -2^ls2, else if rs1 >= 2^ls2 – 1, rD = 2^ls2-1, else rD = rs1
p.clipu	rD, rs1, ls2	if rs1 <= 0, rD = 0, else if rs1 >= 2^ls2 – 1, rD = 2^ls2-1, else rD = rs1

Mnemonic	Description
p.addN rD, rs1, rs2, Is3	rD = (rs1 + rs2) >>> Is3 Note: Arithmetic shift right. Setting Is3 to 2 replaces former p.avg
p.adduN rD, rs1, rs2, Is3	rD = (rs1 + rs2) >> Is3 Note: Logical shift right. Setting Is3 to 2 replaces former p.avg
p.addRN rD, rs1, rs2, Is3	rD = (rs1 + rs2 + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right.
p.adduRN rD, rs1, rs2, Is3	rD = (rs1 + rs2 + 2^(Is3-1))) >> Is3 Note: Logical shift right.
p.subN rD, rs1, rs2, Is3	rD = (rs1 - rs2) >>> Is3 Note: Arithmetic shift right.
p.subuN rD, rs1, rs2, Is3	rD = (rs1 - rs2) >> Is3 Note: Logical shift right.
p.subRN rD, rs1, rs2, Is3	rD = (rs1 - rs2 + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right.
p.subuRN rD, rs1, rs2, Is3	rD = (rs1 - rs2 + 2^(Is3-1))) >> Is3 Note: Logical shift right.

12.3.4 General ALU Encoding

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rD	opcode	
000 0010	00000	src1	000	dest	011 0011	p.abs rD, rs1
000 0010	src2	src1	010	dest	011 0011	p.slet rD, rs1, rs2
000 0010	src2	src1	011	dest	011 0011	p.sletu rD, rs1, rs2
000 0010	src2	src1	100	dest	011 0011	p.min rD, rs1, rs2
000 0010	src2	src1	101	dest	011 0011	p.minu rD, rs1, rs2
000 0010	src2	src1	110	dest	011 0011	p.max rD, rs1, rs2
000 0010	src2	src1	111	dest	011 0011	p.maxu rD, rs1, rs2
000 1000	00000	src1	100	dest	011 0011	p.exths rD, rs1
000 1000	00000	src1	101	dest	011 0011	p.exthz rD, rs1
000 1000	00000	src1	110	dest	011 0011	p.extbs rD, rs1
000 1000	00000	src1	111	dest	011 0011	p.extbz rD, rs1

31 25	24 20	19 15	14 12	11 7	6 0
funct7	Is2[4:0]	rs1	funct3	rD	opcode

000 1010	Iuimm5[4:0]	src1	001	dest	011 0011	p.clip rD, rs1, Is2
000 1010	Iuimm5[4:0]	src1	010	dest	011 0011	p.clipu rD, rs1, Is2

31 30 29 25 24 20 19 15 14 12 11 7 6 0							
f2	Is3[4:0]	rs2	rs1	funct3	rD	opcode	
00	Luimm5[4:0]	src2	src1	010	dest	101 1011	p.addN rD, rs1, rs2, Is3
10	Luimm5[4:0]	src2	src1	010	dest	101 1011	p.adduN rD, rs1, rs2, Is3
00	Luimm5[4:0]	src2	src1	110	dest	101 1011	p.addRN rD, rs1, rs2, Is3
10	Luimm5[4:0]	src2	src1	110	dest	101 1011	p.adduRN rD, rs1, rs2, Is3
00	Luimm5[4:0]	src2	src1	011	dest	101 1011	p.subN rD, rs1, rs2, Is3
10	Luimm5[4:0]	src2	src1	011	dest	101 1011	p.subuN rD, rs1, rs2, Is3
00	Luimm5[4:0]	src2	src1	111	dest	101 1011	p.subRN rD, rs1, rs2, Is3
10	Luimm5[4:0]	src2	src1	111	dest	101 1011	p.subuRN rD, rs1, rs2, Is3

12.4 Multiply-Accumulate

12.4.1 MAC Operations

Mnemonic		Description
32-Bit x 32-Bit Multiplication Operations		
p.mac	rD, rs1, rs2	rD = rD + rs1 * rs2
p.msu	rD, rs1, rs2	rD = rD - rs1 * rs2
16-Bit x 16-Bit Multiplication		
p.muls	rD, rs1, rs2	rD[31:0] = Sext(rs1[15:0]) * Sext(rs2[15:0])
p.mulhhs	rD, rs1, rs2	rD[31:0] = Sext(rs1[31:15]) * Sext(rs2[31:15])
p.mulsN	rD, rs1, rs2, Is3	rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0])) >>> Is3 Note: Arithmetic shift right
p.mulhhsN	rD, rs1, rs2, Is3	rD[31:0] = (Sext(rs1[31:15]) * Sext(rs2[31:15])) >>> Is3 Note: Arithmetic shift right
p.mulsRN	rD, rs1, rs2, Is3	rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0]) + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right
p.mulhhsRN	rD, rs1, rs2, Is3	rD[31:0] = (Sext(rs1[31:15]) * Sext(rs2[31:15]) + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right
p.mulu	rD, rs1, rs2	rD[31:0] = Zext(rs1[15:0]) * Zext(rs2[15:0])
p.mulhhu	rD, rs1, rs2	rD[31:0] = Zext(rs1[31:15]) * Zext(rs2[31:15])
p.muluN	rD, rs1, rs2, Is3	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0])) >>> Is3 Note: Logical shift right

Mnemonic	Description
p.mulhhuN rD, rs1, rs2, Is3	rD[31:0] = (Zext(rs1[31:15]) * Zext(rs2[31:15])) >>> Is3 Note: Logical shift right
p.muluRN rD, rs1, rs2, Is3	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0]) + 2^(Is3-1)) >>> Is3 Note: Logical shift right
p.mulhhuRN rD, rs1, rs2, Is3	rD[31:0] = (Zext(rs1[31:15]) * Zext(rs2[31:15]) + 2^(Is3-1)) >>> Is3 Note: Logical shift right
16-Bit x 16-Bit Multiply-Accumulate	
p.macsN rD, rs1, rs2, Is3	rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0]) + rD) >>> Is3 Note: Arithmetic shift right
p.machhsN rD, rs1, rs2, Is3	rD[31:0] = (Sext(rs1[31:15]) * Sext(rs2[31:15]) + rD) >>> Is3 Note: Arithmetic shift right
p.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
p.machhsRN , rD, rs1, rs2, Is3	rD[31:0] = (Sext(rs1[31:15]) * Sext(rs2[31:15]) + rD + 2^(Is3-1)) >>> Is3 Note: Arithmetic shift right
p.macuN rD, rs1, rs2, Is3	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0]) + rD) >>> Is3 Note: Logical shift right
p.machhuN rD, rs1, rs2, Is3	rD[31:0] = (Zext(rs1[31:15]) * Zext(rs2[31:15]) + rD) >>> Is3 Note: Logical shift right
p.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
p.machhuRN rD, rs1, rs2, Is3	rD[31:0] = (Zext(rs1[31:15]) * Zext(rs2[31:15]) + rD + 2^(Is3-1)) >>> Is3 Note: Logical shift right

12.4.2 MAC Encoding

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

31 30	29 25	24 20	19 15	14 12	11 7	6 0	
f2	Is3[4:0]	rs2	rs1	funct3	rD	opcode	
10	00000	src2	src1	000	dest	101 1011	p.mulsrD, rs1, rs2
11	00000	src2	src1	000	dest	101 1011	p.mulhhsrD, rs1, rs2
10	Luimm5[4:0]	src2	src1	000	dest	101 1011	p.mulsNrD, rs1, rs2, Is3
11	Luimm5[4:0]	src2	src1	000	dest	101 1011	p.mulhhsNrD, rs1, rs2, Is3

10	Luimm5[4:0]	src2	src1	100	dest	101 1011	p.mulsRN rD, rs1, rs2, ls3
11	Luimm5[4:0]	src2	src1	100	dest	101 1011	p.mulhhsRN rD, rs1, rs2, ls3
00	00000	src2	src1	000	dest	101 1011	p.mulu rD, rs1, rs2
01	00000	src2	src1	000	dest	101 1011	p.mulhhu rD, rs1, rs2
00	Luimm5[4:0]	src2	src1	000	dest	101 1011	p.muluN rD, rs1, rs2, ls3
01	Luimm5[4:0]	src2	src1	000	dest	101 1011	p.mulhhuN rD, rs1, rs2, ls3
00	Luimm5[4:0]	src2	src1	100	dest	101 1011	p.muluRN rD, rs1, rs2, ls3
01	Luimm5[4:0]	src2	src1	100	dest	101 1011	p.mulhhuRN rD, rs1, rs2, ls3
10	Luimm5[4:0]	src2	src1	001	dest	101 1011	p.macsN rD, rs1, rs2, ls3
11	Luimm5[4:0]	src2	src1	001	dest	101 1011	p.machhsN rD, rs1, rs2, ls3
10	Luimm5[4:0]	src2	src1	101	dest	101 1011	p.macsRN rD, rs1, rs2, ls3
11	Luimm5[4:0]	src2	src1	101	dest	101 1011	p.machhsRN rD, rs1, rs2, ls3
00	Luimm5[4:0]	src2	src1	001	dest	101 1011	p.macuN rD, rs1, rs2, ls3
01	Luimm5[4:0]	src2	src1	001	dest	101 1011	p.machhuN rD, rs1, rs2, ls3
00	Luimm5[4:0]	src2	src1	101	dest	101 1011	p.macuRN rD, rs1, rs2, ls3
01	Luimm5[4:0]	src2	src1	101	dest	101 1011	p.machhuRN rD, rs1, rs2, ls3

12.5 Vectorial

Vectorial instructions perform operations in a SIMD-like manner 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.

Vectorial 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
- 16-Bit, to perform two operations on the 2 half-words inside a 32-bit word at the same time

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.
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.

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

12.5.1 Vectorial ALU Operations

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

Mnemonic	Description
pv.dotup[.sc,.sci].h	rD = rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operations are unsigned
pv.dotup[.sc,.sci].b	rD = rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operations are unsigned
pv.dotusp[.sc,.sci].h	rD = rs1[0] * op2[0] + rs1[1] * op2[1] Note: rs1 is treated as unsigned, while rs2 is treated as signed
pv.dotusp[.sc,.sci].b	rD = rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: rs1 is treated as unsigned, while rs2 is treated as signed
pv.dotsp[.sc,.sci].h	rD = rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operations are signed
pv.dotsp[.sc,.sci].b	rD = rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operations are signed
pv.sdotup[.sc,.sci].h	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operations are unsigned
pv.sdotup[.sc,.sci].b	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operations are unsigned
pv.sdotusp[.sc,.sci].h	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] Note: rs1 is treated as unsigned, while rs2 is treated as signed
pv.sdotusp[.sc,.sci].b	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: rs1 is treated as unsigned, while rs2 is treated as signed
pv.sdotsp[.sc,.sci].h	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] Note: All operations are signed
pv.sdotsp[.sc,.sci].b	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1] + rs1[2] * op2[2] + rs1[3] * op2[3] Note: All operations are signed
Shuffle and Pack Instructions	
pv.shuffle.h	rD[31:16] = rs1[rs2[16]*16+15:rs2[16]*16] rD[15:0] = rs1[rs2[0]*16+15:rs2[0]*16]
pv.shuffle.sci.h	rD[31:16] = rs1[I1*16+15:I1*16] rD[15:0] = rs1[I0*16+15:I0*16] Note: I1 and I0 represent bits 1 and 0 of the immediate
pv.shuffle.b	rD[31:24] = rs1[rs2[25:24]*8+7:rs2[25:24]*8] rD[23:16] = rs1[rs2[17:16]*8+7:rs2[17:16]*8] rD[15:8] = rs1[rs2[9:8]*8+7:rs2[9:8]*8] rD[7:0] = rs1[rs2[1:0]*8+7:rs2[1:0]*8]
pv.shuffleI0.sci.b	rD[31:24] = rs1[7:0] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]

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

12.5.2 Vectorial ALU Encoding

31 27	26	25 24	20 19	15 14	12 11	7 6	0	
funct5	F		rs2	rs1	funct3	rD	opcode	
0 0000	0	0	src2	src1	000	dest	101 0111	pv.add.h rD, rs1, rs2
0 0000	0	0	src2	src1	100	dest	101 0111	pv.add.sc.h rD, rs1, rs2
0 0000	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.add.sci.h rD, rs1, Imm6
0 0000	0	0	src2	src1	001	dest	101 0111	pv.add.b rD, rs1, rs2
0 0000	0	0	src2	src1	101	dest	101 0111	pv.add.sc.b rD, rs1, rs2
0 0000	0	Imm6[5:0]		src1	111	dest	101 0111	pv.add.sci.b rD, rs1, Imm6
0 0001	0	0	src2	src1	000	dest	101 0111	pv.sub.h rD, rs1, rs2

0 0001	0	0	src2	src1	100	dest	101 0111	pv.sub.sc.h rD, rs1, rs2
0 0001	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.sub.sci.h rD, rs1, Imm6
0 0001	0	0	src2	src1	001	dest	101 0111	pv.sub.b rD, rs1, rs2
0 0001	0	0	src2	src1	101	dest	101 0111	pv.sub.sc.b rD, rs1, rs2
0 0001	0	Imm6[5:0]		src1	111	dest	101 0111	pv.sub.sci.b rD, rs1, Imm6
0 0010	0	0	src2	src1	000	dest	101 0111	pv.avg.h rD, rs1, rs2
0 0010	0	0	src2	src1	100	dest	101 0111	pv.avg.sc.h rD, rs1, rs2
0 0010	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.avg.sci.h rD, rs1, Imm6
0 0010	0	0	src2	src1	001	dest	101 0111	pv.avg.b rD, rs1, rs2
0 0010	0	0	src2	src1	101	dest	101 0111	pv.avg.sc.b rD, rs1, rs2
0 0010	0	Imm6[5:0]		src1	111	dest	101 0111	pv.avg.sci.b rD, rs1, Imm6
0 0011	0	0	src2	src1	000	dest	101 0111	pv.avgu.h rD, rs1, rs2
0 0011	0	0	src2	src1	100	dest	101 0111	pv.avgu.sc.h rD, rs1, rs2
0 0011	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.avgu.sci.h rD, rs1, Imm6
0 0011	0	0	src2	src1	001	dest	101 0111	pv.avgu.b rD, rs1, rs2
0 0011	0	0	src2	src1	101	dest	101 0111	pv.avgu.sc.b rD, rs1, rs2
0 0011	0	Imm6[5:0]		src1	111	dest	101 0111	pv.avgu.sci.b rD, rs1, Imm6
0 0100	0	0	src2	src1	000	dest	101 0111	pv.min.h rD, rs1, rs2
0 0100	0	0	src2	src1	100	dest	101 0111	pv.min.sc.h rD, rs1, rs2
0 0100	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.min.sci.h rD, rs1, Imm6
0 0100	0	0	src2	src1	001	dest	101 0111	pv.min.b rD, rs1, rs2
0 0100	0	0	src2	src1	101	dest	101 0111	pv.min.sc.b rD, rs1, rs2
0 0100	0	Imm6[5:0]		src1	111	dest	101 0111	pv.min.sci.b rD, rs1, Imm6
0 0101	0	0	src2	src1	000	dest	101 0111	pv.minu.h rD, rs1, rs2
0 0101	0	0	src2	src1	100	dest	101 0111	pv.minu.sc.h rD, rs1, rs2
0 0101	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.minu.sci.h rD, rs1, Imm6
0 0101	0	0	src2	src1	001	dest	101 0111	pv.minu.b rD, rs1, rs2
0 0101	0	0	src2	src1	101	dest	101 0111	pv.minu.sc.b rD, rs1, rs2
0 0101	0	Imm6[5:0]		src1	111	dest	101 0111	pv.minu.sci.b rD, rs1, Imm6

0 0110	0	0	src2	src1	000	dest	101 0111	pv.max.h rD, rs1, rs2
0 0110	0	0	src2	src1	100	dest	101 0111	pv.max.sc.h rD, rs1, rs2
0 0110	0		Imm6[5:0]s	src1	110	dest	101 0111	pv.max.sci.h rD, rs1, Imm6
0 0110	0	0	src2	src1	001	dest	101 0111	pv.max.b rD, rs1, rs2
0 0110	0	0	src2	src1	101	dest	101 0111	pv.max.sc.b rD, rs1, rs2
0 0110	0		Imm6[5:0]	src1	111	dest	101 0111	pv.max.sci.b rD, rs1, Imm6
0 0111	0	0	src2	src1	000	dest	101 0111	pv.maxu.h rD, rs1, rs2
0 0111	0	0	src2	src1	100	dest	101 0111	pv.maxu.sc.h rD, rs1, rs2
0 0111	0		Imm6[5:0]s	src1	110	dest	101 0111	pv.maxu.sci.h rD, rs1, Imm6
0 0111	0	0	src2	src1	001	dest	101 0111	pv.maxu.b rD, rs1, rs2
0 0111	0	0	src2	src1	101	dest	101 0111	pv.maxu.sc.b rD, rs1, rs2
0 0111	0		Imm6[5:0]	src1	111	dest	101 0111	pv.maxu.sci.b rD, rs1, Imm6
0 1000	0	0	src2	src1	000	dest	101 0111	pv.srl.h rD, rs1, rs2
0 1000	0	0	src2	src1	100	dest	101 0111	pv.srl.sc.h rD, rs1, rs2
0 1000	0		Imm6[5:0]s	src1	110	dest	101 0111	pv.srl.sci.h rD, rs1, Imm6
0 1000	0	0	src2	src1	001	dest	101 0111	pv.srl.b rD, rs1, rs2
0 1000	0	0	src2	src1	101	dest	101 0111	pv.srl.sc.b rD, rs1, rs2
0 1000	0		Imm6[5:0]	src1	111	dest	101 0111	pv.srl.sci.b rD, rs1, Imm6
0 1001	0	0	src2	src1	000	dest	101 0111	pv.sra.h rD, rs1, rs2
0 1001	0	0	src2	src1	100	dest	101 0111	pv.sra.sc.h rD, rs1, rs2
0 1001	0		Imm6[5:0]s	src1	110	dest	101 0111	pv.sra.sci.h rD, rs1, Imm6
0 1001	0	0	src2	src1	001	dest	101 0111	pv.sra.b rD, rs1, rs2
0 1001	0	0	src2	src1	101	dest	101 0111	pv.sra.sc.b rD, rs1, rs2
0 1001	0		Imm6[5:0]	src1	111	dest	101 0111	pv.sra.sci.b rD, rs1, Imm6
0 1010	0	0	src2	src1	000	dest	101 0111	pv.sll.h rD, rs1, rs2
0 1010	0	0	src2	src1	100	dest	101 0111	pv.sll.sc.h rD, rs1, rs2
0 1010	0		Imm6[5:0]s	src1	110	dest	101 0111	pv.sll.sci.h rD, rs1, Imm6
0 1010	0	0	src2	src1	001	dest	101 0111	pv.sll.b rD, rs1, rs2
0 1010	0	0	src2	src1	101	dest	101 0111	pv.sll.sc.b rD, rs1, rs2

0 1010	0	Imm6[5:0]		src1	111	dest	101 0111	pv.sll.sci.b rD, rs1, Imm6
0 1011	0	0	src2	src1	000	dest	101 0111	pv.or.h rD, rs1, rs2
0 1011	0	0	src2	src1	100	dest	101 0111	pv.or.sc.h rD, rs1, rs2
0 1011	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.or.sci.h rD, rs1, Imm6
0 1011	0	0	src2	src1	001	dest	101 0111	pv.or.b rD, rs1, rs2
0 1011	0	0	src2	src1	101	dest	101 0111	pv.or.sc.b rD, rs1, rs2
0 1011	0	Imm6[5:0]		src1	111	dest	101 0111	pv.or.sci.b rD, rs1, Imm6
0 1100	0	0	src2	src1	000	dest	101 0111	pv.xor.h rD, rs1, rs2
0 1100	0	0	src2	src1	100	dest	101 0111	pv.xor.sc.h rD, rs1, rs2
0 1100	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.xor.sci.h rD, rs1, Imm6
0 1100	0	0	src2	src1	001	dest	101 0111	pv.xor.b rD, rs1, rs2
0 1100	0	0	src2	src1	101	dest	101 0111	pv.xor.sc.b rD, rs1, rs2
0 1100	0	Imm6[5:0]		src1	111	dest	101 0111	pv.xor.sci.b rD, rs1, Imm6
0 1101	0	0	src2	src1	000	dest	101 0111	pv.and.h rD, rs1, rs2
0 1101	0	0	src2	src1	100	dest	101 0111	pv.and.sc.h rD, rs1, rs2
0 1101	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.and.sci.h rD, rs1, Imm6
0 1101	0	0	src2	src1	001	dest	101 0111	pv.and.b rD, rs1, rs2
0 1101	0	0	src2	src1	101	dest	101 0111	pv.and.sc.b rD, rs1, rs2
0 1101	0	Imm6[5:0]		src1	111	dest	101 0111	pv.and.sci.b rD, rs1, Imm6
0 1110	0	0	00000	src1	000	dest	101 0111	pv.abs.h rD, rs1
0 1110	0	0	00000	src1	001	dest	101 0111	pv.abs.b rD, rs1
0 1111	0	Imm6[5:0]		src1	110	dest	101 0111	pv.extract.h rD, Imm6
0 1111	0	Imm6[5:0]		src1	111	dest	101 0111	pv.extract.b rD, Imm6
1 0010	0	Imm6[5:0]		src1	110	dest	101 0111	pv.extractu.h rD, Imm6
1 0010	0	Imm6[5:0]		src1	111	dest	101 0111	pv.extractu.b rD, Imm6
1 0110	0	Imm6[5:0]		src1	110	dest	101 0111	pv.insert.h rD, Imm6
1 0110	0	Imm6[5:0]		src1	111	dest	101 0111	pv.insert.b rD, Imm6
1 0000	0	0	src2	src1	000	dest	101 0111	pv.dotup.h rD, rs1, rs2
1 0000	0	0	src2	src1	100	dest	101 0111	pv.dotup.sc.h rD, rs1, rs2

1 0000	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.dotup.sci.h rD, rs1, Imm6
1 0000	0	0	src2	src1	001	dest	101 0111	pv.dotup.b rD, rs1, rs2
1 0000	0	0	src2	src1	101	dest	101 0111	pv.dotup.sc.b rD, rs1, rs2
1 0000	0	Imm6[5:0]		src1	111	dest	101 0111	pv.dotup.sci.b rD, rs1, Imm6
1 0001	0	0	src2	src1	000	dest	101 0111	pv.dotusp.h rD, rs1, rs2
1 0001	0	0	src2	src1	100	dest	101 0111	pv.dotusp.sc.h rD, rs1, rs2
1 0001	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.dotusp.sci.h rD, rs1, Imm6
1 0001	0	0	src2	src1	001	dest	101 0111	pv.dotusp.b rD, rs1, rs2
1 0001	0	0	src2	src1	101	dest	101 0111	pv.dotusp.sc.b rD, rs1, rs2
1 0001	0	Imm6[5:0]		src1	111	dest	101 0111	pv.dotusp.sci.b rD, rs1, Imm6
1 0011	0	0	src2	src1	000	dest	101 0111	pv.dotsp.h rD, rs1, rs2
1 0011	0	0	src2	src1	100	dest	101 0111	pv.dotsp.sc.h rD, rs1, rs2
1 0011	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.dotsp.sci.h rD, rs1, Imm6
1 0011	0	0	src2	src1	001	dest	101 0111	pv.dotsp.b rD, rs1, rs2
1 0011	0	0	src2	src1	101	dest	101 0111	pv.dotsp.sc.b rD, rs1, rs2
1 0011	0	Imm6[5:0]		src1	111	dest	101 0111	pv.dotsp.sci.b rD, rs1, Imm6
1 0100	0	0	src2	src1	000	dest	101 0111	pv.sdotup.h rD, rs1, rs2
1 0100	0	0	src2	src1	100	dest	101 0111	pv.sdotup.sc.h rD, rs1, rs2
1 0100	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.sdotup.sci.h rD, rs1, Imm6
1 0100	0	0	src2	src1	001	dest	101 0111	pv.sdotup.b rD, rs1, rs2
1 0100	0	0	src2	src1	101	dest	101 0111	pv.sdotup.sc.b rD, rs1, rs2
1 0100	0	Imm6[5:0]		src1	111	dest	101 0111	pv.sdotup.sci.b rD, rs1, Imm6
1 0101	0	0	src2	src1	000	dest	101 0111	pv.sdotusp.h rD, rs1, rs2
1 0101	0	0	src2	src1	100	dest	101 0111	pv.sdotusp.sc.h rD, rs1, rs2
1 0101	0	Imm6[5:0]s		src1	110	dest	101 0111	pv.sdotusp.sci.h rD, rs1, Imm6
1 0101	0	0	src2	src1	001	dest	101 0111	pv.sdotusp.b rD, rs1, rs2
1 0101	0	0	src2	src1	101	dest	101 0111	pv.sdotusp.sc.b rD, rs1, rs2
1 0101	0	Imm6[5:0]		src1	111	dest	101 0111	pv.sdotusp.sci.b rD, rs1, Imm6
1 0111	0	0	src2	src1	000	dest	101 0111	pv.sdotsp.h rD, rs1, rs2

1 0111	0	0	src2	src1	100	dest	101 0111	pv.sdotsp.sc.h rD, rs1, rs2
1 0111	0		Imm6[5:0]s	src1	110	dest	101 0111	pv.sdotsp.sci.h rD, rs1, Imm6
1 0111	0	0	src2	src1	001	dest	101 0111	pv.sdotsp.b rD, rs1, rs2
1 0111	0	0	src2	src1	101	dest	101 0111	pv.sdotsp.sc.b rD, rs1, rs2
1 0111	0		Imm6[5:0]	src1	111	dest	101 0111	pv.sdotsp.sci.b rD, rs1, Imm6
1 1000	0	0	src2	src1	000	dest	101 0111	pv.shuffle.h rD, rs1, rs2
1 1000	0		Imm6[5:0]	src1	110	dest	101 0111	pv.shuffle.sci.h rD, rs1, Imm6
1 1000	0	0	src2	src1	001	dest	101 0111	pv.shuffle.b rD, rs1, rs2
1 1000	0		Imm6[5:0]	src1	111	dest	101 0111	pv.shuffleI0.sci.b rD, rs1, Imm6
1 1101	0		Imm6[5:0]	src1	111	dest	101 0111	pv.shuffleI1.sci.b rD, rs1, Imm6
1 1110	0		Imm6[5:0]	src1	111	dest	101 0111	pv.shuffleI2.sci.b rD, rs1, Imm6
1 1111	0		Imm6[5:0]	src1	111	dest	101 0111	pv.shuffleI3.sci.b rD, rs1, Imm6
1 1001	0	0	src2	src1	000	dest	101 0111	pv.shuffle2.h rD, rs1, rs2
1 1001	0	0	src2	src1	001	dest	101 0111	pv.shuffle2.b rD, rs1, rs2
1 1010	0	0	src2	src1	000	dest	101 0111	pv.pack.h rD, rs1, rs2
1 1011	0	0	src2	src1	001	dest	101 0111	pv.packhi.b rD, rs1, rs2
1 1100	0	0	src2	src1	001	dest	101 0111	pv.packlo.b rD, rs1, rs2

12.5.3 Vectorial Comparison Operations

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

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

Mnemonic		Description
pv.cmpeq[.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] == op2 ? ‘1 : ‘0
pv.cmpne[.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] != op2 ? ‘1 : ‘0
pv.cmpgt[.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] > op2 ? ‘1 : ‘0
pv.cmpge[.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] >=op2 ? ‘1 : ‘0
pv.cmplt[.sc,.sci]{.h,.b}	rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] < op2 ? ‘1 : ‘0

Mnemonic	Description
pv.cmple[.sc,.sci]{.h,.b} rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] <= op2 ? '1 : '0
pv.cmpgtu[.sc,.sci]{.h,.b} rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] > op2 ? '1 : '0 Note: Unsigned comparison
pv.cmpgeu[.sc,.sci]{.h,.b} rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] >= op2 ? '1 : '0 Note: Unsigned comparison
pv.cmpltu[.sc,.sci]{.h,.b} rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] < op2 ? '1 : '0 Note: Unsigned comparison
pv.cmpleu[.sc,.sci]{.h,.b} rD, rs1, {rs2, Imm6}	rD[i] = rs1[i] <= op2 ? '1 : '0 Note: Unsigned comparison

12.5.4 Vectorial Comparison Encoding

31 27	26	25 24	20 19	15 14	12 11	7 6	0	
funct5	F		rs2	rs1	funct3	rD	opcode	
0 0000	1	0	src2	src1	000	dest	101 0111	pv.cmpeq.h rD, rs1, rs2
0 0000	1	0	src2	src1	100	dest	101 0111	pv.cmpeq.sc.h rD, rs1, rs2
0 0000	1	Imm6[5:0]		src1	110	dest	101 0111	pv.cmpeq.sci.h rD, rs1, Imm6
0 0000	1	0	src2	src1	001	dest	101 0111	pv.cmpeq.b rD, rs1, rs2
0 0000	1	0	src2	src1	101	dest	101 0111	pv.cmpeq.sc.b rD, rs1, rs2
0 0000	1	Imm6[5:0]		src1	111	dest	101 0111	pv.cmpeq.sci.b rD, rs1, Imm6
0 0001	1	0	src2	src1	000	dest	101 0111	pv.cmpne.h rD, rs1, rs2
0 0001	1	0	src2	src1	100	dest	101 0111	pv.cmpne.sc.h rD, rs1, rs2
0 0001	1	Imm6[5:0]		src1	110	dest	101 0111	pv.cmpne.sci.h rD, rs1, Imm6
0 0001	1	0	src2	src1	001	dest	101 0111	pv.cmpne.b rD, rs1, rs2
0 0001	1	0	src2	src1	101	dest	101 0111	pv.cmpne.sc.b rD, rs1, rs2
0 0001	1	Imm6[5:0]		src1	111	dest	101 0111	pv.cmpne.sci.b rD, rs1, Imm6
0 0010	1	0	src2	src1	000	dest	101 0111	pv.cmpgt.h rD, rs1, rs2
0 0010	1	0	src2	src1	100	dest	101 0111	pv.cmpgt.sc.h rD, rs1, rs2
0 0010	1	Imm6[5:0]		src1	110	dest	101 0111	pv.cmpgt.sci.h rD, rs1, Imm6
0 0010	1	0	src2	src1	001	dest	101 0111	pv.cmpgt.b rD, rs1, rs2
0 0010	1	0	src2	src1	101	dest	101 0111	pv.cmpgt.sc.b rD, rs1, rs2
0 0010	1	Imm6[5:0]		src1	111	dest	101 0111	pv.cmpgt.sci.b rD, rs1, Imm6
0 0011	1	0	src2	src1	000	dest	101 0111	pv.cmpge.h rD, rs1, rs2

0 0011	1	0	src2	src1	100	dest	101 0111	pv.cmpge.sc.h rD, rs1, rs2
0 0011	1		Imm6[5:0]	src1	110	dest	101 0111	pv.cmpge.sci.h rD, rs1, Imm6
0 0011	1	0	src2	src1	001	dest	101 0111	pv.cmpge.b rD, rs1, rs2
0 0011	1	0	src2	src1	101	dest	101 0111	pv.cmpge.sc.b rD, rs1, rs2
0 0011	1		Imm6[5:0]	src1	111	dest	101 0111	pv.cmpge.sci.b rD, rs1, Imm6
0 0100	1	0	src2	src1	000	dest	101 0111	pv.cmplt.h rD, rs1, rs2
0 0100	1	0	src2	src1	100	dest	101 0111	pv.cmplt.sc.h rD, rs1, rs2
0 0100	1		Imm6[5:0]	src1	110	dest	101 0111	pv.cmplt.sci.h rD, rs1, Imm6
0 0100	1	0	src2	src1	001	dest	101 0111	pv.cmplt.b rD, rs1, rs2
0 0100	1	0	src2	src1	101	dest	101 0111	pv.cmplt.sc.b rD, rs1, rs2
0 0100	1		Imm6[5:0]	src1	111	dest	101 0111	pv.cmplt.sci.b rD, rs1, Imm6
0 0101	1	0	src2	src1	000	dest	101 0111	pv.cmple.h rD, rs1, rs2
0 0101	1	0	src2	src1	100	dest	101 0111	pv.cmple.sc.h rD, rs1, rs2
0 0101	1		Imm6[5:0]	src1	110	dest	101 0111	pv.cmple.sci.h rD, rs1, Imm6
0 0101	1	0	src2	src1	001	dest	101 0111	pv.cmple.b rD, rs1, rs2
0 0101	1	0	src2	src1	101	dest	101 0111	pv.cmple.sc.b rD, rs1, rs2
0 0101	1		Imm6[5:0]	src1	111	dest	101 0111	pv.cmple.sci.b rD, rs1, Imm6
0 0110	1	0	src2	src1	000	dest	101 0111	pv.cmpgtu.h rD, rs1, rs2
0 0110	1	0	src2	src1	100	dest	101 0111	pv.cmpgtu.sc.h rD, rs1, rs2
0 0110	1		Imm6[5:0]	src1	110	dest	101 0111	pv.cmpgtu.sci.h rD, rs1, Imm6
0 0110	1	0	src2	src1	001	dest	101 0111	pv.cmpgtu.b rD, rs1, rs2
0 0110	1	0	src2	src1	101	dest	101 0111	pv.cmpgtu.sc.b rD, rs1, rs2
0 0110	1		Imm6[5:0]	src1	111	dest	101 0111	pv.cmpgtu.sci.b rD, rs1, Imm6
0 0111	1	0	src2	src1	000	dest	101 0111	pv.cmpgeu.h rD, rs1, rs2
0 0111	1	0	src2	src1	100	dest	101 0111	pv.cmpgeu.sc.h rD, rs1, rs2
0 0111	1		Imm6[5:0]	src1	110	dest	101 0111	pv.cmpgeu.sci.h rD, rs1, Imm6
0 0111	1	0	src2	src1	001	dest	101 0111	pv.cmpgeu.b rD, rs1, rs2
0 0111	1	0	src2	src1	101	dest	101 0111	pv.cmpgeu.sc.b rD, rs1, rs2
0 0111	1		Imm6[5:0]	src1	111	dest	101 0111	pv.cmpgeu.sci.b rD, rs1, Imm6

0 1000	1	0	src2	src1	000	dest	101 0111	pv.cmpltu.h rD, rs1, rs2
0 1000	1	0	src2	src1	100	dest	101 0111	pv.cmpltu.sc.h rD, rs1, rs2
0 1000	1	Imm6[5:0]		src1	110	dest	101 0111	pv.cmpltu.sci.h rD, rs1, Imm6
0 1000	1	0	src2	src1	001	dest	101 0111	pv.cmpltu.b rD, rs1, rs2
0 1000	1	0	src2	src1	101	dest	101 0111	pv.cmpltu.sc.b rD, rs1, rs2
0 1000	1	Imm6[5:0]		src1	111	dest	101 0111	pv.cmpltu.sci.b rD, rs1, Imm6
0 1001	1	0	src2	src1	000	dest	101 0111	pv.cmpleu.h rD, rs1, rs2
0 1001	1	0	src2	src1	100	dest	101 0111	pv.cmpleu.sc.h rD, rs1, rs2
0 1001	1	Imm6[5:0]		src1	110	dest	101 0111	pv.cmpleu.sci.h rD, rs1, Imm6
0 1001	1	0	src2	src1	001	dest	101 0111	pv.cmpleu.b rD, rs1, rs2
0 1001	1	0	src2	src1	101	dest	101 0111	pv.cmpleu.sc.b rD, rs1, rs2
0 1001	1	Imm6[5:0]		src1	111	dest	101 0111	pv.cmpleu.sci.b rD, rs1, Imm6

12.5.5 Vectorial Branching Operations

Mnemonic		Description
pv.ball	rs1, Imm12	Branch to PC + (Imm12 << 1) if rs1 is equal to 0xFFFFFFFF. This corresponds to -1, or all bits set to 1. All bits set is the result of a vectorial comparison where all sub-words comparisons were true.

12.5.6 Vectorial Branching Encoding

31	25	24	20	19	15	14	12	11	7	6	0	
Imm		rs2		rs1	funct3		Imm				opcode	
[12]	[10:5]	00000		src1	010		[4:1]	[11]			110 0011	pv.ball rs1, Imm