RI5CY: User Manual

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

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

Pasquale Davide Schiavone (pschiavo@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
1.1	12.07.16	P.D. Schiavone	Removed pv.ball, and replaced with p.beqimm
1.2	14.11.16	P.D. Schiavone	Added register variants of clip, addnorm, and bit manipulation instructions
1.3	04.01.17	Michael Gautschi	Fixed typos, references, foot notes and date style
1.4	08.03.17	P.D. Schiavone	Updated to priv spec 1.9 and new IRQ handling
1.5	06.06.17	P.D. Schiavone	General updates
1.6	03.07.17	Michael Gautschi	Extended with optional FP support
1.7	12.07.17	P.D. Schiavone	Revised instructions added in Rev. 1.2
1.8	08.11.17	P.D. Schiavone	Add note in HW Loop
1.9	06.02.18	A. Ruospo	Fixed CSR reset values and general corrections
2.0	08.03.18	P.D. Schiavone	Fixed Documentation issue with lp.setupi instruction #29
2.1	16.05.18	P.D. Schiavone	Fixed Documentation issue in Debug
2.2	28.06.18	P.D. Schiavone	Fixed Nested Exception Support #40
3.0	19.12.18	P.D. Schiavone	PMP plus priv spec 1.10
4.0	17.04.19	P.D. Schiavone	New debug. Change HWLoop addresses
4.1	21.08.19	Robert Balas	Update PCER and PCMR addresses

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.
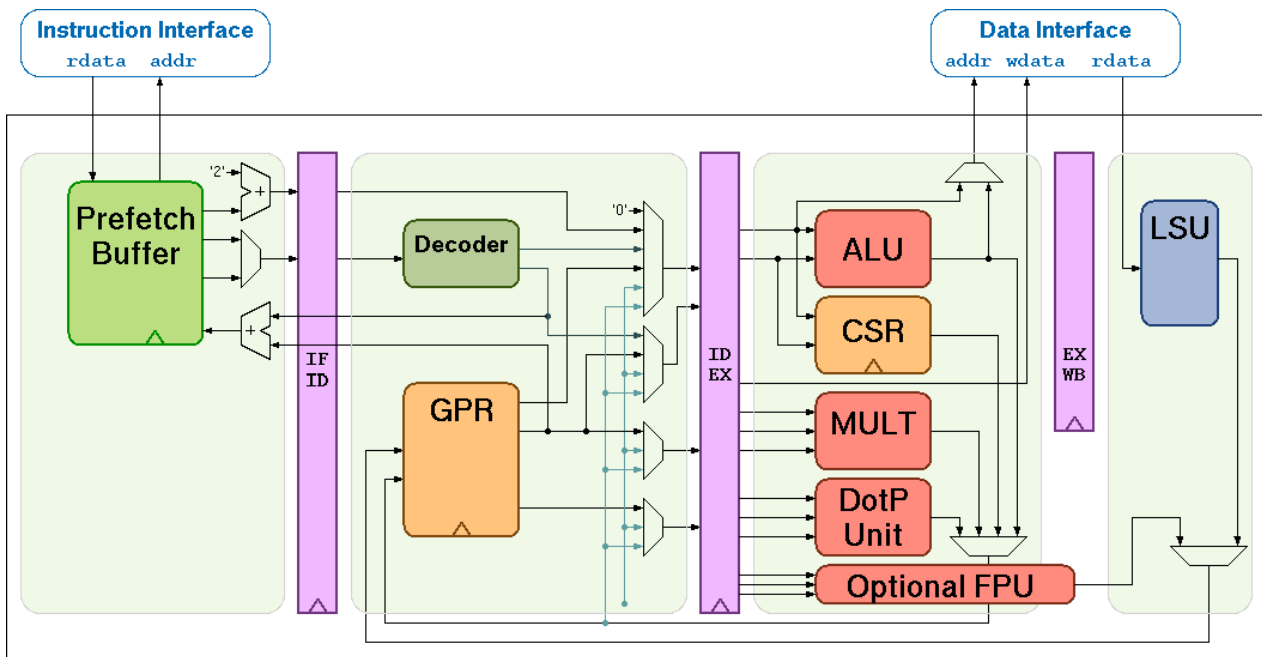
Figure 1: Block Diagram

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

- Optional full support for RV32F Single Precision Floating Point Extensions

- PULP specific extensions

  o Post-Incrementing load and stores, see Chapter 3

  o Multiply-Accumulate extensions, see Chapter 4

  o ALU extensions, see Chapter 5

  o Hardware Loops, see Chapter 7

1.2   Optional Floating Point Support

Floating-point support in the form of IEEE-754 single precision can be enabled by setting the parameter **FPU** of the toplevel file "riscv_core" to one. This will instantiate the FPU in the execution stage, and also extend the register file to host floating-point operands and extend the ALU to support the floating-point comparisons and classifications.

1.3   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. With the FPU, the core area increases to about 90 kGE (30kGE FPU, 10kGE additional register file).

1.4   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.

1.5   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 7 focuses on the hardware loop extensions and Chapter 9 explains the register file. Control and status registers are explained in Chapter 10 and Chapter 11 gives an overview of all performance counters. Chapter 12 deals with exceptions and interrupts, and Chapter 13 summarizes the accessible debug registers. Finally, Chapter 14 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 2, Figure 3 and Figure 4 show example-timing diagrams of the protocol.

clk
data_addr_o   Address
data_wdata_o   WData
data_req_o
data_gnt_i
data_rvalid_i
data_rdata_i   RData
data_we_o   WE
data_be_o   BE
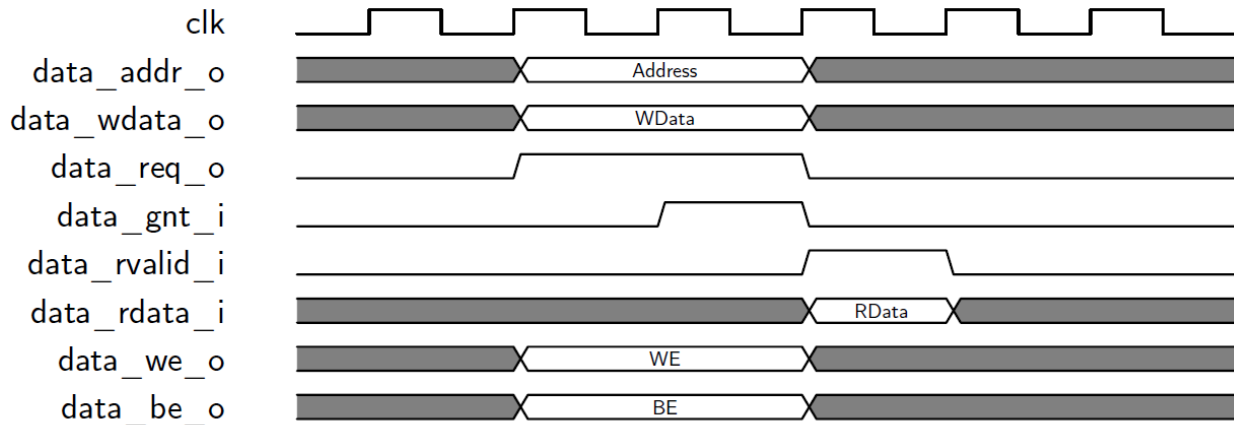
Figure 2: Basic Memory Transaction

clk
data_addr_o   Addr1   Addr2
data_wdata_o   WData1   Wdata2
data_req_o
data_gnt_i
data_rvalid_i
data_rdata_i   RData1   RData2
data_we_o   WE1   WE2
data_be_o   BE1   BE2
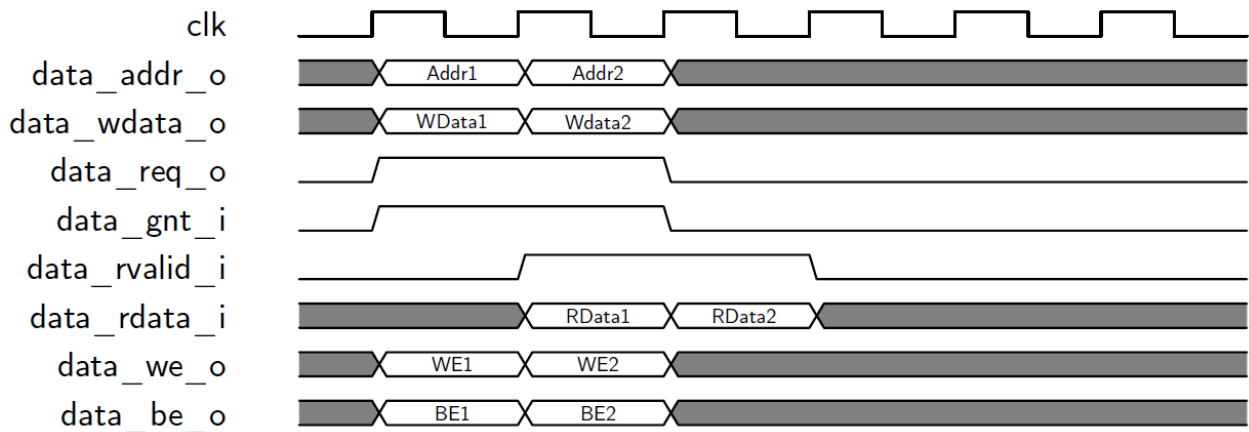
Figure 3: Back-to-back Memory Transaction

clk
data_addr_o   Address
data_wdata_o   WData
data_req_o
data_gnt_i
data_rvalid_i
data_rdata_i   RData
data_we_o   WE
data_be_o   BE
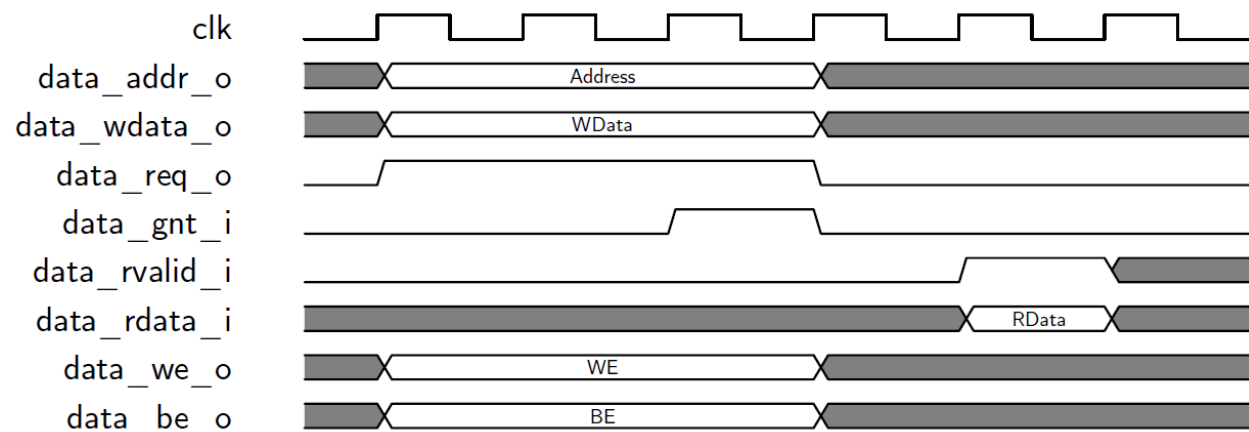
Figure 4: Slow Response Memory Transaction

3.3   Physical Memory Protection (PMP) Unit

The RI5CY core has a PMP module which can be enabled by setting the parameter PULP_SECURE=1 which also enabled the core to possibly run in USER MODE. Such unit has a configurable number of entries (up to 16) and supports all the modes as TOR, NAPOT and NA4. Every fetch, load and store access executed in USER MODE are first filtered by the PMP unit which can possibly generated exceptions. For the moment, the MPRV bit in MSTATUS as well as the LOCK mechanism in the PMP are not supported.

3.4   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 Optional private Floating Point Unit (FPU)

It is possible to extend the core with a private FPU, which is capable of performing all RISC-V floating-point operations that are defined in the RV32F ISA extensions. The latency of the individual instructions and information where they are computed are summarized in Table 3. FP extensions can be enabled by setting the parameter of the toplevel file "riscv_core.sv" to one.

The FPU is divided into three parts:

1. A *simple FPU* of ~10kGE complexity, which computes FP-ADD, FP-SUB and FP-casts.

2. An *iterative FP-DIV/SQRT unit* of ~7 kGE complexity, which computes FP-DIV/SQRT operations.

3. An *FP-FMA unit* which takes care of all fused operations. This unit is currently only supported through a Synopsys Design Ware instantiation, or a Xilinx block for FPGA targets.

FP-Operation	Executed in:	Latency	Operation:	Information
flw	LSU	2	Loads 32 to FP-RF	Mapped to lw
fsw	LSU	2	Stores FP-operand to memory	Mapped to sw
fmadd	FPU	3	rd = rs1 * rs2 + rs3	
fmsub	FPU	3	rd = rs1 * rs2– rs3	
fnmadd	FPU	3	rd = – (rs1 * rs2+ rs3)	
fnmsub	FPU	3	rd = –(rs1 * rs2 – rs3)	
fadd.s	FPU	2	rd = rs1 + rs2	
fsub.s	FPU	2	rd = rs1 – rs2	
fmul.s	FPU	2	rd = rs1 * rs2	
fdiv.s	FPU	5 – 8	rd = rs1 / rs2	According to precision specified in CSR see Table 5: Custom CSR to control the precision of FP DIV/SQRT operationsTable 5
fsqrt.s	FPU	5 – 8	rd = sqrt(rs1)	
fclass.s	ALU	1	See specification	
fmv.s.w	ALU	1	Move from int-RF to FP-RF	Mapped to mv
fmv.w.s	ALU	1	Move from FP-RF to int-RF	
fsgnj.s	ALU	1	Inserts sign of rs2	
fsgnjn.s	ALU	1	Inserts negative sign of rs2	
fsgnjx.s	ALU	1	Inserts xor of the two signs	
feq.s	ALU	1	(rs1 == rs2)	Reuses integer comparator

FP-Operation	Executed in:	Latency	Operation:	Information
flt.s	ALU	1	(rs1 < rs2)	
fle.s	ALU	1	(rs1 <= rs2)	
fmin	ALU	1	rd = min(rs1, rs2)	
fmax	ALU	1	rd = max(rs1, rs2)	
fcvt.x.w	FPU	2	Int to FP cast	
fcvt.x.wu	FPU	2	Unsigned int to FP cast	
fcvt.w.x	FPU	2	FP to int cast	
fcvt.wu.x	FPU	2	FP to unsigned int cast	

Table 3: Overview of FP-operations

6.1   FP CSR

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

Since RISCY includes an iterative div/sqrt unit, its precision and latency can be controlled over a custom csr (fprec). This allows faster division / square-root operations at the lower precision. By default, the single-precision equivalents are computed with a latency of 8 cycles.

CSR Address				Hex	Name	Acc.	Description
11:10	**9:8**	**7:6**	**5:0**				
00	00	00	00001	0x001	fflags	R/W	Floating-point accrued exceptions
00	00	00	00010	0x002	frm	R/W	Floating-point dynamic rounding mode
00	00	00	00011	0x003	fcsr	R/W	Floating-point control and status register
00	00	00	00110	0x006	fprec	R/W	Custom flag which controls the precision and latency of the iterative div/sqrt unit

Table 4: FP related CSRs

fprec value	Precision	Latency
0	Default value: single precision	8
8 – 11	Computes as many mantissa bits as specified in "fprec value"	5
12 – 15		6
16 – 19		7
20 – 23		8

Table 5: Custom CSR to control the precision of FP DIV/SQRT operations

6.2   Floating-point Performance Counters:

Some specific performance counters have been implemented to profile FP-kernels.

6.3   Some hints on synthesizing the FPU

The pipeline of the FPU is not balanced but it includes one pipeline register in front of the *simple FPU* which is intended to be moved in to the pipeline with automatic retiming commands. The same holds for the *FP-FMA unit* which contains two pipeline registers (one in front, and one after the unit).

Optimal performance is only achieved with retiming these two blocks. This can for example be achieved with the "optimize_register" command of the Synopsys Design Compiler.

7 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 and the last instruction cannot be any jump or branch instruction.

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

7.1 CSR Mapping

CSR Address				Hex	Name	Acc.	Description
11:10	9:8	7:6	5:0				
01	11	10	110000	0x7C0	lpstart[0]	R/W	Hardware Loop 0 Start
01	11	10	110001	0x7C1	lpendt[0]	R/W	Hardware Loop 0 End
01	11	10	110010	0x7C2	lpcount[0]	R/W	Hardware Loop 0 Counter
01	11	10	110000	0x7C4	lpstart[1]	R/W	Hardware Loop 0 Start
01	11	10	110001	0x7C5	lpend[1]	R/W	Hardware Loop 1 End
01	11	10	110010	0x7C6	lpcount[1]	R/W	Hardware Loop 1 Counter

Table 6: Hardware-Loop CSR Mapping

8   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 5 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.
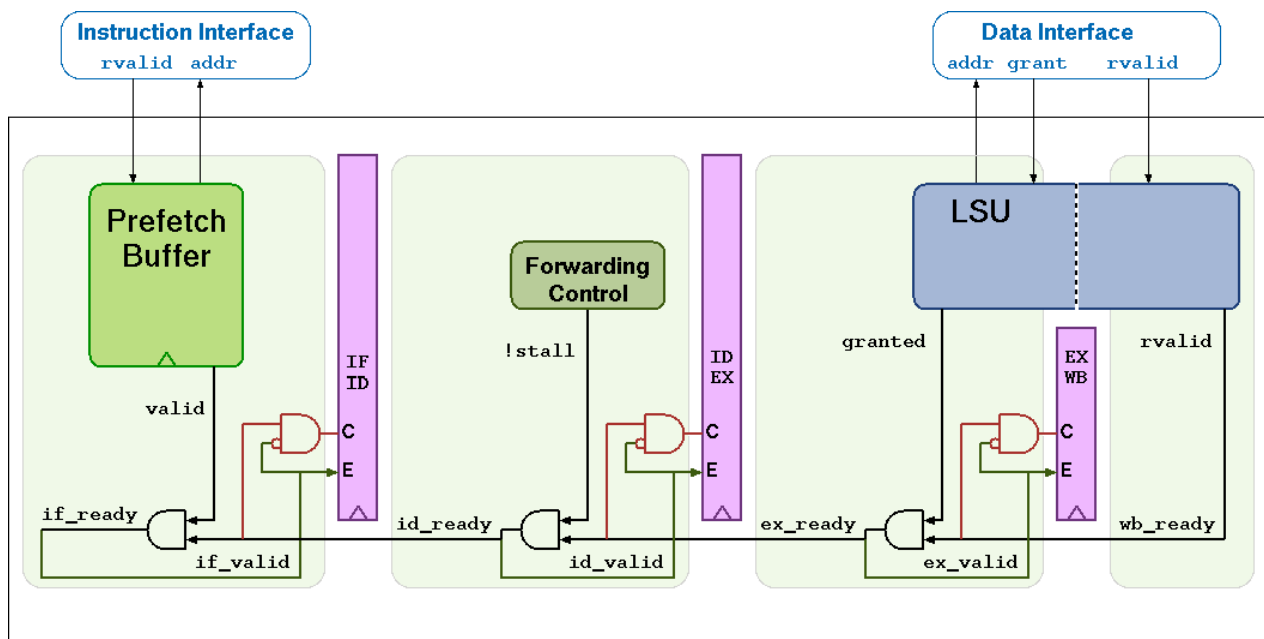
Figure 5: RI5CY Pipeline

9   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.

9.1   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

9.2   FPU Register File

In case the optional FPU is instantiated, 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 fp_reg_sel 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.

10 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	00	000101	0x305	MTVEC	R	Machine Trap-Vector Base Address
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	0x7E0, 0xCC0	PCER	R/W	Performance Counter Enable
01	11	10	100001	0x7E1, 0xCC1	PCMR	R/W	Performance Counter Mode
01	11	10	110xxx	0x7B0-0x7B7	HWLP	R/W	Hardware Loop Registers
11	00	00	010000	0xC10	PRIVLV	R	Privilege Level
00	00	00	010100	0x014	UHARTID	R	Hardware Thread ID
11	11	00	010100	0xF14	MHARTID	R	Hardware Thread ID
01	11	10	110000	0x7B0	DCSR	R/W	Debug Control and Status
01	11	10	110001	0x7B1	DPC	R/W	Debug PC
01	11	10	110010	0x7B2	DSCRATCH0	R/W	Debug Scratch Register 0
01	11	10	110011	0x7B3	DSCRATCH1	R/W	Debug Scratch Register 1

Table 7: Control and Status Register Map

10.1 Machine Status (MSTATUS)

CSR Address: 0x300

Reset Value: 0x0000_1800
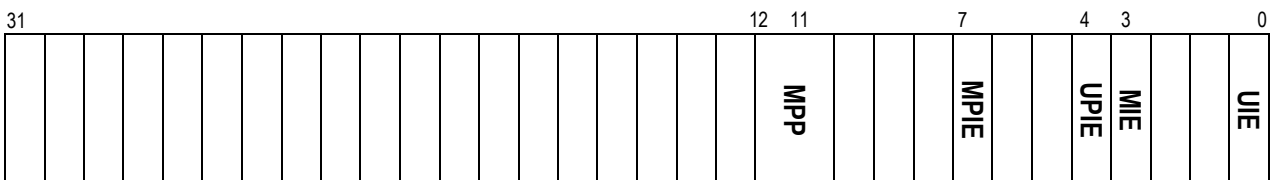
Detailed:

Bit #	R/W	Description
12:11	R/W	**MPP:** Machine Previous Priviledge mode, hardwired to 11 when the user mode is not enabled.
7	R/W	**Previous Machine Interrupt Enable:** When an exception is encountered, MPIE will be set to MIE. When the mret instruction is executed, the value of MPIE will be stored to MIE.
4	R/W	**Previous User Interrupt Enable:** If user mode is enabled, when an exception is encountered, UPIE will be set to UIE. When the uret instruction is executed, the value of UPIE will be stored to UIE. *Note that PULP/issimo does not support USER interrupts.*
3	R/W	**Machine Interrupt Enable:** If you want to enable interrupt handling in your exception handler, set the Interrupt Enable MIE to 1'b1 inside your handler code.
0	R/W	**User Interrupt Enable:** If you want to enable user level interrupt handling in your exception handler, set the Interrupt Enable UIE to 1'b1 inside your handler code. *Note that PULP/issimo does not support USER interrupts.*

10.2 User Status (USTATUS)

CSR Address: 0x000

Reset Value: 0x0000_0000
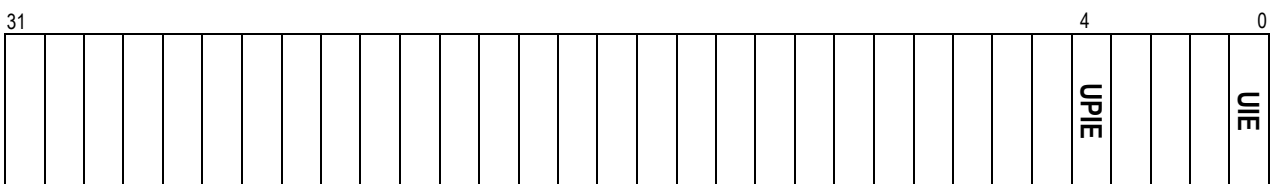
Detailed:

Bit #	R/W	Description
4	R/W	**Previous User Interrupt Enable:** If user mode is enabled, when an exception is encountered, UPIE will be set to UIE. When the uret instruction is executed, the value of UPIE will be stored to UIE. *Note that PULP/issimo does not support USER interrupts.*
0	R/W	**User Interrupt Enable:** If you want to enable user level interrupt handling in your exception handler, set the Interrupt Enable UIE to 1'b1 inside your handler code. *Note that PULP/issimo does not support USER interrupts.*

10.3 Machine Trap-Vector Base Address (MTVEC)

CSR Address: 0x305

When an exception 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. The only mode supported is vectorized interrupt, thus the bits 1:0 are hardwired to 01.

Table 6: MTVEC

10.4 User Trap-Vector Base Address (UTVEC)

CSR Address: 0x005

31																								7							0
																								0	0	0	0	0	0	0	1
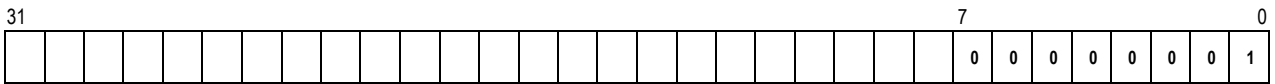
When an exception is encountered in user-mode, the core jumps to the corresponding handler using the content of the UTVEC[31:8] as base address. Only 8-byte aligned addresses are allowed. The only mode supported is vectorized interrupt, thus the bits 1:0 are hardwired to 01. *Note that PULP/issimo does not support USER interrupts.*

Table 6: UTVEC

10.5 Machine Exception PC (MEPC)

CSR Address: 0x341

Reset Value: 0x0000_0000

31	0
MEPC	

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.

10.6 User Exception PC (UEPC)

CSR Address: 0x041

Reset Value: 0x0000_0000

31	0
UEPC	

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

10.7 Machine Cause (MCAUSE)

CSR Address: 0x342

Reset Value: 0x0000_0000

31																										4		0
Interrupt																										**Exception Code**		

Detailed:

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

Table 7: MCAUSE

10.8 User Cause (UCAUSE)

CSR Address: 0x042

Reset Value: 0x0000_0000

31																										4		0
Interrupt																										**Exception Code**		

Detailed:

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

Table 8: MCAUSE

10.9 Privilege Level

CSR Address: 0xC10

Reset Value: 0x0000_0003

31

	PRV LVL

Detailed:

Bit #	R/W	Description
1:0	R	**PRV LVL**: It contains the current privilege level the core is executing.

Table 9: PRIVILEGE LEVEL

10.10 MHARTID/UHARTID

CSR Address: 0xF14/0x014

Reset Value: Defined

31	10	5 4 3	0
	Cluster ID		**Core ID**

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

10.11 PMP Configuration (PMPCFGx)

CSR Address: 0x3A{0,1,2,3}

Reset Value: 0x0000_0000

31	0
PMPCFGx	

If the PMP is enabled, these four registers contain the configuration of the PMP as specified by the official privileged spec 1.10.

10.12 PMP Address (PMPADDRx)

CSR Address: 0x3B{0x0, 0x1, …. 0xF}

Reset Value: 0x0000_0000

31	0
PMPADDRx	

If the PMP is enabled, these sixteen registers contain the addresses of the PMP as specified by the official privileged spec 1.10.

10.13 Debug Control and Status (DCSR)

CSR Address: 0x7B0

Reset Value: 0x0000_0003
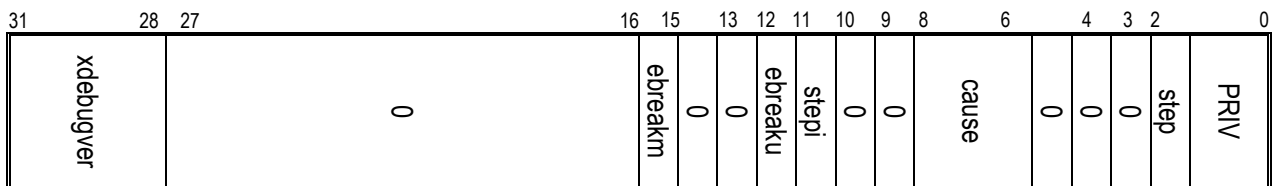
Bit #	R/W	Description
31:28	R	**xdebugver:** returns 4 - External debug support exists as it is described in this document.
15	R/W	**ebreakm**
12	R/W	**ebreaku**
11	R/W	**stepi**
8:6	R/W	**cause**
2	R/W	**step**
1:0	R	**priv:** returns the current priviledge mode

10.14 Debug PC (DPC)

CSR Address: 0x7B1

Reset Value: 0x0000_0000
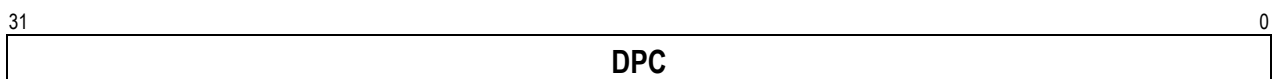
When the core enters in Debug Mode, DPC contains the virtual address of the next instruction to be executed.

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

CSR Address: 0x7B2/0x7B3

Reset Value: 0x0000_0000

31	0
DSCRATCH0/1	

Scratch register that can be used by implementations that need it.

11 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

11.1 Performance Counter Mode Register (PCMR)

CSR Address: 0x7A1

Reset Value: 0x0000_0003

31																														1	0
																														Saturation	Global Enable

Detailed:

Bit #	R/W	Description
0	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.
1	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

11.2 Performance Counter Event Register (PCER)

CSR Address: 0x7A0

Reset Value: 0x0000_0000

31						16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					FP_WB	FP_DEP	FP_CONT	FP_TYPE	CSR_HAZARD	TCDM_CONT	ST_EXT_CYC	LD_EXT_CYC	ST_EXT	LD_EXT	COMP_INSTR	BRANCH_TAKEN	BRANCH	JUMP	ST	LD	IMISS	JMP_STALL	LD_STALL	INSTR	CYCLES

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
20	R/W	**FP_WB**
19	R/W	**FP_DEP**
18	R/W	**FP_CONT**
17	R/W	**FP_TYPE**
16	R/W	**CSR_HAZARD**
15	R/W	**TCDM_CONT**
14	R/W	**ST_EXT_CYC**
13	R/W	**LD_EXT_CYC**
12	R/W	**ST_EXT**
11	R/W	**LD_EXT**
10	R/W	**COMP_INSTR**
9	R/W	**BRANCH_TAKEN**
8	R/W	**BRANCH**
7	R/W	**JUMP**
6	R/W	**ST**
5	R/W	**LD**
4	R/W	**IMISS**
3	R/W	**JMP_STALL**
2	R/W	**LD_STALL**
1	R/W	**INSTR**
0	R/W	**CYCLES**

Table 7: 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.

11.3 Performance Counter Counter Register (PCCR0-31)

CSR Address: 0x780 - 0x79F

Reset Value: 0x0000_0000

31   0

Unsigned Integer Counter Value
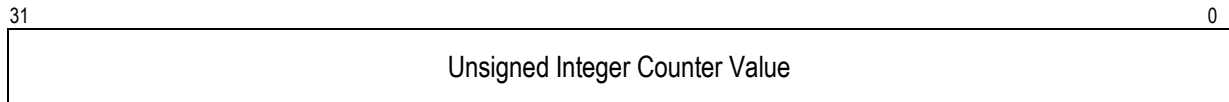
Table 8: PCCR0-31

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 (PULP only)
PCCR14	ST_EXT_CYC	Cycles used for memory stores to EXT. Every non-TCDM access is considered external (PULP only)

Register	Name	Description
PCCR15	TCDM_CONT	Cycles wasted due to TCDM/log-interconnect contention (PULP only)
PCCR16	CSR_HAZARD	Cycles wasted due to CSR access
PCCR17	FP_TYPE	Cycles wasted due to different latencies of subsequent FP-operations
PCCR18	FP_CONT	Cycles wasted due to contentions at the shared FPU (PULP only)
PCCR19	FP_DEP	Cycles wasted due to data hazards in subsequent FP instructions
PCCR20	FP_WB	Cycles wasted due to FP operations resulting in write-back contentions
PCCR31	ALL	Special Register, a write to this register will set all counters to the supplied value

Table 94: 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 6 shows how events are first masked with the PCER register and then ORed together to increase the one performance counter PCCR.
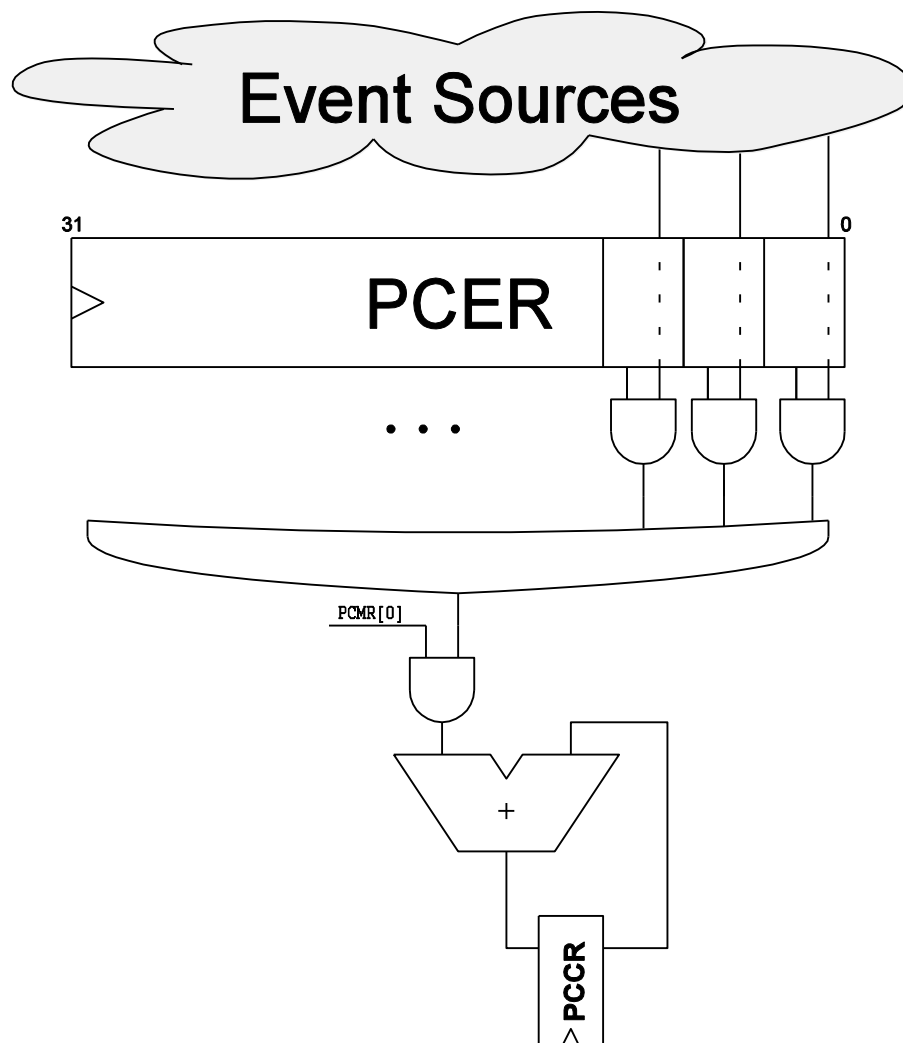
Figure 6: Events and PCCR, PCMR and PCER on the ASIC.

12 Exceptions and Interrupts

RI5CY supports interrupts, exceptions on illegal instructions and (if enabled) on PMP filtered requests on the data and instruction bus.
The base address of the interrupt vector table is given by the mtvec address. As RI5CY supports only vectorized interrupts, the interrupt 0 is reserved for exceptions as illegal instructions, ecall and instruction or data prohibited accesses.

12.1 Interrupts

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.
Multiple interrupts requests are assumed to be handled by event/interrupt controller. When an interrupt is taken, the core gives an acknowledge signal to the event/interrupt controller as well as the interrupt id taken.

12.2 Exceptions

The illegal instruction exception, ecall instruction exceptions cannot be disabled and are always active.
For PMP exceptions when enabled, every instruction or data requests is filtered by the PMP which can possibly generated LOAD, STORE or FETCH exceptions.

12.3 Handling

RI5CY supports SW-assisted 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 mret instruction, the core jumps to the program counter saved in the CSR register MEPC and restores the MPIE value of the register MSTATUS to IE. When entering an interrupt/exception handler, the core sets MEPC to the current program counter and saves the current value of MIE in MPIE of the MSTATUS register.

13 Debug

RI5CY supports the RISC-V debug specification 0.13 and it implementes the execution based to reuse the existing core pipeline.

RI5CY has a **debug_req_i** input port that is sent by the system Debug Module. Such request makes the core jumps to the a specific address location where the Debug Rom is mapped. Such address location is referred as to the parameter DM_HaltAddress. RI5CY implements the debug sets of registers as dpc, dcsr, dscratch0, dscratch1.

14 Instruction Set Extensions

14.1 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.

14.1.1.1 Load 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)

14.1.1.2 Store 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

14.1.2 Encoding

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

14.2 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.

14.2.1 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, uimmL, uimmS	lpstart[L] = pc + 4 lpend[L] = pc + (uimmS << 1) lpcount[L] = uimmL

14.2.2 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**

14.3 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 14.3.1.

- 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 14.3.3.

- Immediate branching instructions are useful to compare a register with an immediate value before taking or not a branch, see Section 13.3.5.

14.3.1 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.extractr	**rD, rs1, rs2**	rD = Sext((rs1 & ((1 << rs2[9:5]) – 1) << rs2[4:0]) >> rs2[4:0]) Note: rs2[9:5]+ rs2[4:0] must be <= 32
p.extractur	**rD, rs1, rs2**	rD = Zext((rs1 & ((1 << rs2[9:5]) – 1) << rs2[4:0]) >> rs2[4:0]) Note: rs2[9:5]+ rs2[4:0] must be <= 32
p.insert	**rD, rs1, ls3, ls2**	rD = rD | (rs1[ls3:0] << ls2) Note: ls3 + ls2 must be <= 32, the rest of the bits of rD are passed through and are not modified
p.insertr	**rD, rs1, rs2**	rD = rD | (rs1[ls3:0] << rs2[4:0]) Note: rs2[9:5]+ rs2[4:0] 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)) – 1) << ls2) Note: ls3 + ls2 must be <= 32
p.bclrr	**rD, rs1, rs2**	rD = rs1 & ~(((1 << (rs2[9:5]+1)) – 1) << rs2[4:0]) Note: rs2[9:5]+ rs2[4:0] must be <= 32
p.bset	**rD, rs1, ls3, ls2**	rD = rs1 | (((1 << (ls3+1)) – 1) << ls2) Note: ls3 + ls2 must be <= 32
p.bsetr	**rD, rs1, rs2**	rD = rs1 | (((1 << (rs2[9:5]+1)) – 1) << rs2[4:0]) Note: rs2[9:5]+ rs2[4:0] 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.

Mnemonic		Description
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)

14.3.2 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**
10	00000	src2	src1	000	dest	011 0011	**p.extractr rD, rs1, rs2**
10	00000	src2	src1	001	dest	011 0011	**p.extractur rD, rs1, rs2**
10	00000	src2	src1	010	dest	011 0011	**p.insertr rD, rs1, rs2**
10	00000	src2	src1	011	dest	011 0011	**p.bclrr rD, rs1, rs2**
10	00000	src2	scr1	100	dest	011 0011	**p.bsetr rD, rs1, rs2**

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**	

14.3.3 General ALU Operations

Mnemonic		Description
p.abs	**rD, rs1**	rD = rs1 < 0 ? –rs1 : rs1

Mnemonic		Description
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, 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, $-2^{(Is2-1)}= -1$ while $(2^{(Is2-1)}-1)=0$;
p.clipr	**rD, rs1, rs2**	if rs1 <= -(rs2+1), rD = -(rs2+1), else if rs1 >=rs2, rD = rs2, else rD = rs1
p.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, $(2^{(Is2-1)}-1)=0$;
p.clipur	**rD, rs1, rs2**	if rs1 <= 0, rD = 0, else if rs1 >= rs2, rD = rs2, else rD = rs1
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.

Mnemonic	Description
p.adduRN rD, rs1, rs2, ls3	rD = (rs1 + rs2 + 2^(ls3-1))) >> ls3 Note: Logical shift right.
p.addNr rD, rs1, rs2	rD = (rD + rs1) >>> rs2[4:0] Note: Arithmetic shift right.
p.adduNr rD, rs1, rs2	rD = (rD + rs1) >> rs2[4:0]
p.addRNr rD, rs1, rs2	rD = (rD + rs1 + 2^(rs2[4:0]-1)) >>> rs2[4:0] Note: Arithmetic shift right.
p.adduRNr rD, rs1, rs2	rD = (rD + rs1 + 2^(rs2[4:0]-1))) >> rs2[4:0] Note: Logical shift right.
p.subN rD, rs1, rs2, ls3	rD = (rs1 - rs2) >>> ls3 Note: Arithmetic shift right.
p.subuN rD, rs1, rs2, ls3	rD = (rs1 - rs2) >> ls3 Note: Logical shift right.
p.subRN rD, rs1, rs2, ls3	rD = (rs1 - rs2 + 2^(ls3-1)) >>> ls3 Note: Arithmetic shift right.
p.subuRN rD, rs1, rs2, ls3	rD = (rs1 - rs2 + 2^(ls3-1))) >> ls3 Note: Logical shift right.
p.subNr rD, rs1, rs2	rD = (rD – rs1) >>> rs2[4:0] Note: Arithmetic shift right.
p.subuNr rD, rs1, rs2	rD = (rD – rs1) >> rs2[4:0] Note: Logical shift right.
p.subRNr rD, rs1, rs2	rD = (rD – rs1+ 2^(rs2[4:0]-1)) >>> rs2[4:0] Note: Arithmetic shift right.
p.subuRNr rD, rs1, rs2	rD = (rD – rs1+ 2^(rs2[4:0]-1))) >> rs2[4:0] Note: Logical shift right.

14.3.4 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	ls2[4:0]	rs1	funct3	rD	opcode	
000 1010	luimm5[4:0]	src1	001	dest	011 0011	**p.clip rD, rs1, ls2**
000 1010	luimm5[4:0]	src1	010	dest	011 0011	**p.clipu rD, rs1, ls2**
000 1010	src2	src1	010	dest	011 0011	**p.clipr rD, rs1, ls2**
000 1010	src2	src1	010	dest	011 0011	**p.clipur rD, rs1, ls2**

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	**p.addN rD, rs1, rs2, ls3**
10	Luimm5[4:0]	src2	src1	010	dest	101 1011	**p.adduN rD, rs1, rs2, ls3**
00	Luimm5[4:0]	src2	src1	110	dest	101 1011	**p.addRN rD, rs1, rs2, ls3**
10	Luimm5[4:0]	src2	src1	110	dest	101 1011	**p.adduRN rD, rs1, rs2, ls3**
00	Luimm5[4:0]	src2	src1	011	dest	101 1011	**p.subN rD, rs1, rs2, ls3**
10	Luimm5[4:0]	src2	src1	011	dest	101 1011	**p.subuN rD, rs1, rs2, ls3**
00	Luimm5[4:0]	src2	src1	111	dest	101 1011	**p.subRN rD, rs1, rs2, ls3**
10	Luimm5[4:0]	src2	src1	111	dest	101 1011	**p.subuRN rD, rs1, rs2, ls3**
01	Luimm5[4:0]	src2	src1	010	dest	101 1011	**p.addNr rD, rs1, rs2**
11	00000	src2	src1	010	dest	101 1011	**p.adduNr rD, rs1, rs**
01	00000	src2	src1	110	dest	101 1011	**p.addRNr rD, rs1, rs**
11	00000	src2	src1	110	dest	101 1011	**p.adduRNr rD, rs1, rs2**
01	00000	src2	src1	011	dest	101 1011	**p.subNr rD, rs1, rs2**
11	00000]	src2	src1	011	dest	101 1011	**p.subuN r rD, rs1, rs2**
01	00000	src2	src1	111	dest	101 1011	**p.subRNr rD, rs1, rs2**
11	00000	src2	src1	111	dest	101 1011	**p.subuRNr rD, rs1, rs2**

14.3.5 Immediate Branching Operations

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

14.3.6 Immediate Branching Encoding

31	25	24	20	19	15	14	12	11	7	6	0	
Imm12		Imm5		rs1		funct3		Imm12		opcode		
[12]	[10:5]	[4:0]		src1		010		[4:1]	[11]	110 0011		**p.beqimm rs1, Imm5, Imm12**
[12]	[10:5]	[4:0]		Src1		011		[4:1]	[11]	1100011		**p.bneimm rs1, Imm5, Imm12**

14.4 Multiply-Accumulate

14.4.1 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
p.mulhhuN	**rD, rs1, rs2, Is3**	rD[31:0] = (Zext(rs1[31:15]) * Zext(rs2[31:15])) >>> Is3 Note: Logical shift right

Mnemonic		Description
p.muluRN	**rD, rs1, rs2, ls3**	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0]) + 2^(ls3-1)) >>> ls3 Note: Logical shift right
p.mulhhuRN	**rD, rs1, rs2, ls3**	rD[31:0] = (Zext(rs1[31:15]) * Zext(rs2[31:15]) + 2^(ls3-1)) >>> ls3 Note: Logical shift right
	16-Bit x 16-Bit Multiply-Accumulate	
p.macsN	**rD, rs1, rs2, ls3**	rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0]) + rD) >>> ls3 Note: Arithmetic shift right
p.machhsN	**rD, rs1, rs2, ls3**	rD[31:0] = (Sext(rs1[31:15]) * Sext(rs2[31:15]) + rD) >>> ls3 Note: Arithmetic shift right
p.macsRN	**rD, rs1, rs2, ls3**	rD[31:0] = (Sext(rs1[15:0]) * Sext(rs2[15:0]) + rD + 2^(ls3-1)) >>> ls3 Note: Arithmetic shift right
p.machhsRN	**, rD, rs1, rs2, ls3**	rD[31:0] = (Sext(rs1[31:15]) * Sext(rs2[31:15]) + rD + 2^(ls3-1)) >>> ls3 Note: Arithmetic shift right
p.macuN	**rD, rs1, rs2, ls3**	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0]) + rD) >>> ls3 Note: Logical shift right
p.machhuN	**rD, rs1, rs2, ls3**	rD[31:0] = (Zext(rs1[31:15]) * Zext(rs2[31:15]) + rD) >>> ls3 Note: Logical shift right
p.macuRN	**rD, rs1, rs2, ls3**	rD[31:0] = (Zext(rs1[15:0]) * Zext(rs2[15:0]) + rD + 2^(ls3-1)) >>> ls3 Note: Logical shift right
p.machhuRN	**rD, rs1, rs2, ls3**	rD[31:0] = (Zext(rs1[31:15]) * Zext(rs2[31:15]) + rD + 2^(ls3-1)) >>> ls3 Note: Logical shift right

14.4.2 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	ls3[4:0]	rs2	rs1	funct3	rD	opcode	
10	00000	src2	src1	000	dest	101 1011	**p.muls rD, rs1, rs2**
11	00000	src2	src1	000	dest	101 1011	**p.mulhhs rD, rs1, rs2**
10	Luimm5[4:0]	src2	src1	000	dest	101 1011	**p.mulsN rD, rs1, rs2, ls3**
11	Luimm5[4:0]	src2	src1	000	dest	101 1011	**p.mulhhsN rD, rs1, rs2, ls3**
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**

14.5 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.

14.5.1 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	
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

Mnemonic	Description
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.shufflel0.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]
pv.shufflel1.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]

Mnemonic	Description
pv.shufflel2.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.shufflel3.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] == 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]
pv.shuffle2.b	rD[31:24] = ((rs2[26] == 1) ? rs1 : rD)[rs2[25:24]*8+7:rs2[25:24]*8] rD[23:16] = ((rs2[18] == 1) ? rs1 : rD)[rs2[17:16]*8+7:rs2[17:16]*8] rD[15:8] = ((rs2[10] == 1) ? rs1 : rD)[rs2[9:8]*8+7:rs2[9:8]*8] rD[7:0] = ((rs2[2] == 1) ? rs1 : rD)[rs2[1:0]*8+7:rs2[1:0]*8]
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

14.5.2 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.shufflel0.sci.b rD, rs1, Imm6**
1 1101	0		Imm6[5:0]	src1	111	dest	101 0111	**pv.shufflel1.sci.b rD, rs1, Imm6**
1 1110	0		Imm6[5:0]	src1	111	dest	101 0111	**pv.shufflel2.sci.b rD, rs1, Imm6**
1 1111	0		Imm6[5:0]	src1	111	dest	101 0111	**pv.shufflel3.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**

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

14.5.3 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
pv.cmple[.sc,.sci]{.h,.b}	**rD, rs1, {rs2, Imm6}**	rD[i] = rs1[i] <= op2 ? '1 : '0

Mnemonic	Description
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

14.5.4 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

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