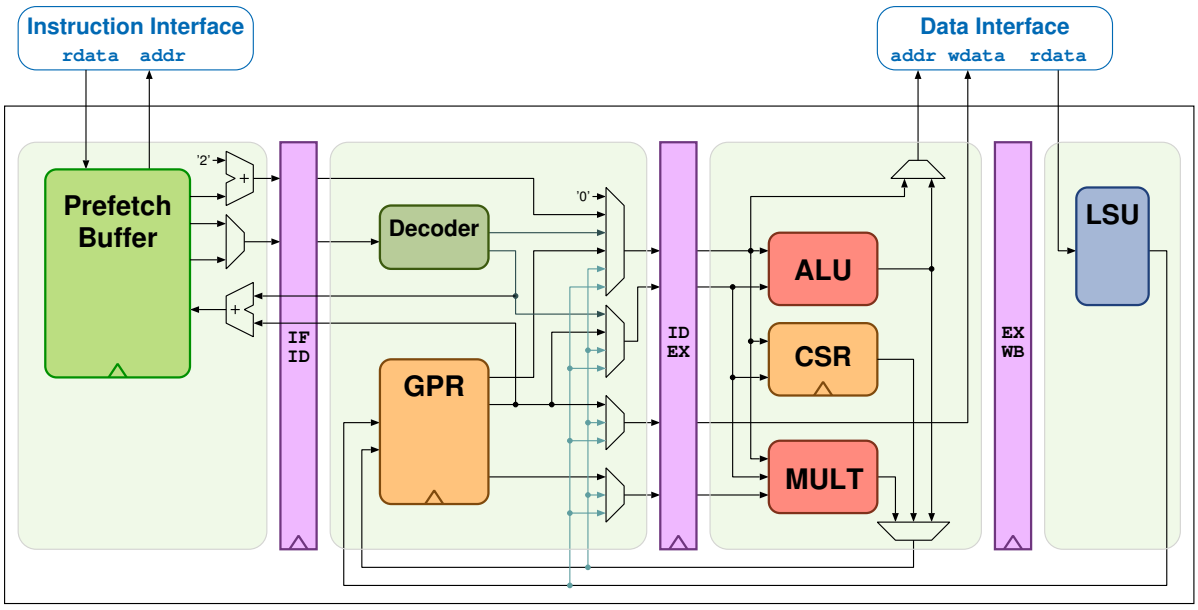
RI5CY Core: Datasheet



Andreas Traber
atraber@iis.ee.ethz.ch

February 2, 2017

Contents

1 Overview

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

Figure 1.1 shows a block diagram of the core.

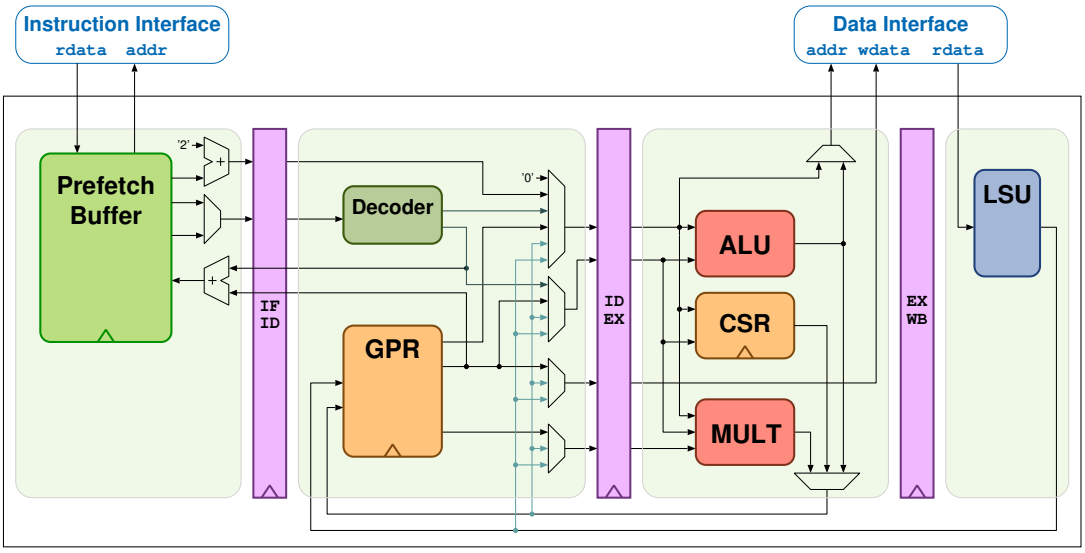


Figure 1.1: RI5CY Overview.

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
- Partial support for RV32M Standard Extension for Integer Multiplication and Division Only the `mul` instruction is supported.
- 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, where there is an option to use latches instead of flip-flops. See Chapter 8 for more details about the register file. The core occupies an area of about 35 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.

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 after one cycle. The instruction address must be half-word-aligned. 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 to fetch instructions. 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 2.1 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.

Table 2.1: Instruction Fetch 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_i` is valid now for this cycle. When `instr_rvalid_i` is high, another request can be sent.
`instr_gnt_i`	**input**	The instruction cache accepted the request. The `instr_addr_o` may be change in the next cylce.

2.1 Protocol

The protocol used to communicate with the instruction cache or the instruction memory is the same as the protocl 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 3.1 describes the signals that are used by the LSU.

Table 3.1: LSU Signals

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, sent together with `data_req_o`
`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, valid when `data_rvalid_i` is set
`data_rvalid_i`	**input**	`data_rdata_i` is valid.
`data_gnt_i`	**input**	The memory accepted the request, another request can be sent in the next cycle

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. Also 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 the grant, the memory answers with a `data_rvalid_i` set high when `data_rdata_i` is valid. This may happen one cycle after the grant was received, but may take any number of cycles after the grant was 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 3.1, Figure 3.2 and Figure 3.3 show example timing diagrams of the protocol.

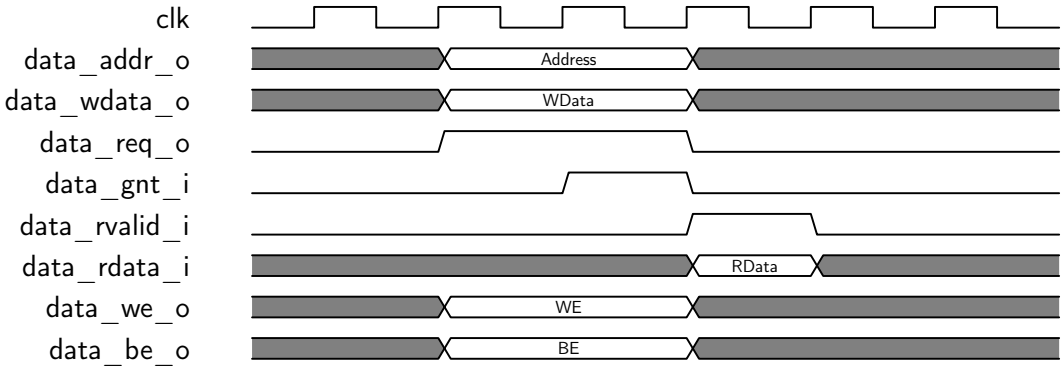


Figure 3.1: Basic Memory Transaction

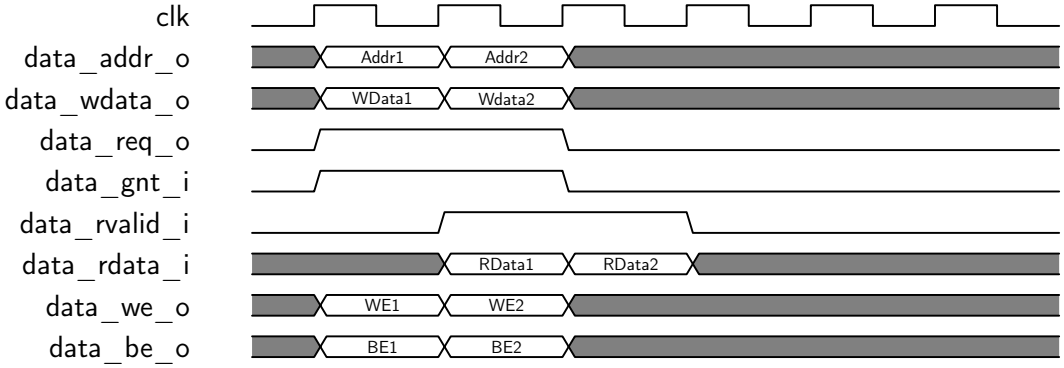


Figure 3.2: Back to Back Memory Transaction

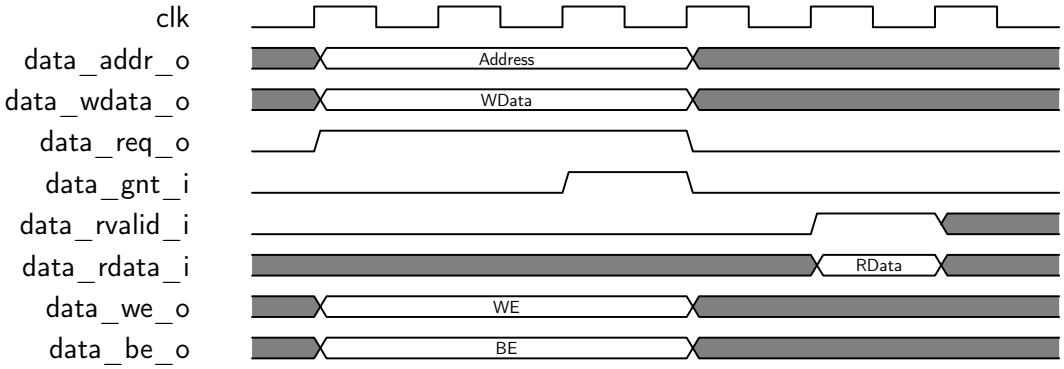


Figure 3.3: Slow Answer Memory Transaction

3.3 Post-Incrementing Load and Store Instructions

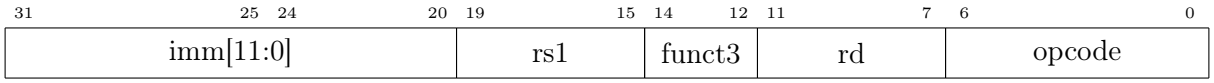
Post-incrementing load and store instructions perform a load/store operation from/to the data memory while at the same time increasing the base address by the specified offset. For the memory access the base address without offset is used.

Post-incrementing load and stores reduce the number of instructions necessary to execute when running in a loop, i.e. the address increment can be embedded in the post-increment instructions. Coupled with the hardware loop extension a significant reduction in the number of instructions necessary to execute small loops can be achieved.

A description of the individual instructions that were added can be found below.

3.3.1 lb rD, imm(rs1!)

Loads a byte from data memory on the address `rs1` and saves `(rs1 + I)` in register `rs1` after the load has completed.

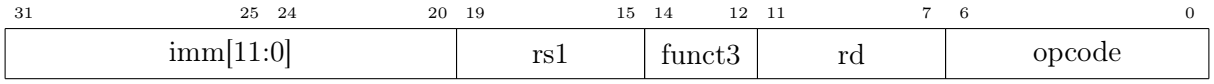


imm[11:0]	src1	000	dest	000 1011

Operation: `rD = Sext(mem[rs1]); rs1 += Sext(imm)`

3.3.2 lh rD, imm(rs1!)

Loads a half-word from data memory on the address `rs1` and saves `(rs1 + I)` in register `rs1` after the load has completed.

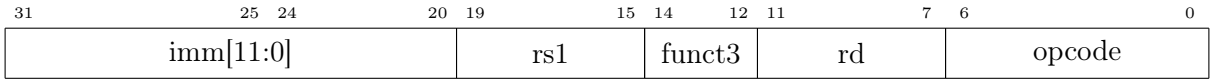


imm[11:0]	src1	001	dest	000 1011

Operation: `rD = Sext(mem[rs1]); rs1 += Sext(imm)`

3.3.3 lw rD, imm(rs1!)

Loads a word from data memory on the address `rs1` and saves `(rs1 + I)` in register `rs1` after the load has completed.

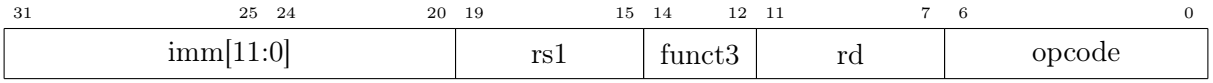


imm[11:0]	src1	010	dest	000 1011

Operation: `rD = mem[rs1]; rs1 += Sext(imm)`

3.3.4 lbu rD, imm(rs1!)

Loads a byte from data memory on the address `rs1` and saves `(rs1 + I)` in register `rs1` after the load has completed.

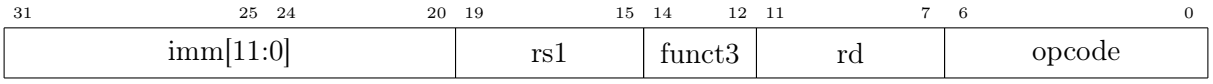


imm[11:0]	src1	100	dest	000 1011

Operation: `rD = Zext(mem[rs1]); rs1 += Sext(imm)`

3.3.5 lhu rD, imm(rs1!)

Loads a half-word from data memory on the address `rs1` and saves `(rs1 + I)` in register `rs1` after the load has completed.

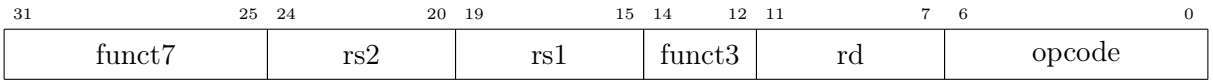


imm[11:0]	src1	101	dest	000 1011

Operation: `rD = Zext(mem[rs1]); rs1 += Sext(imm)`

3.3.6 lb rD, rs2(rs1!)

Loads a byte from data memory on the address `rs1` and saves `(rs1 + rs2)` in register `rs1` after the load has completed.

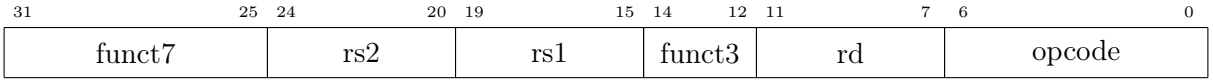


000 0000	src2	src1	111	dest	000 1011

Operation: `rD = Sext(mem[rs1]); rs1 += rs2`

3.3.7 lh rD, rs2(rs1!)

Loads a half-word from data memory on the address `rs1` and saves `(rs1 + rs2)` in register `rs1` after the load has completed.

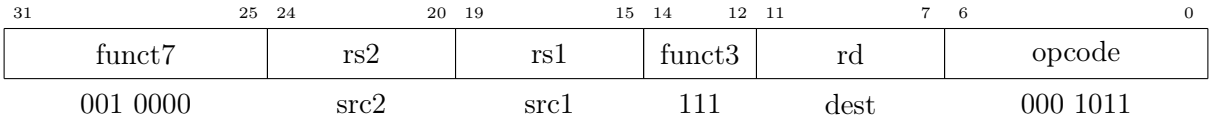


000 1000	src2	src1	111	dest	000 1011

Operation: `rD = Sext(mem[rs1]); rs1 += rs2`

3.3.8 lw rD, rs2(rs1!)

Loads a word from data memory on the address `rs1` and saves `(rs1 + I)` in register `rs1` after the load has completed.

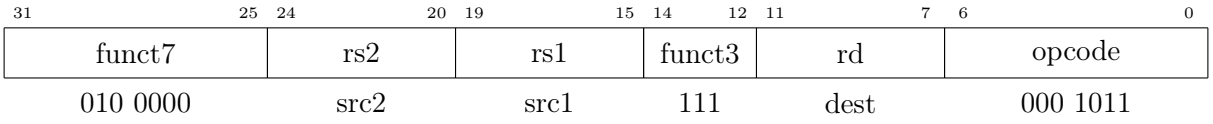


31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
001 0000	src2	src1	111	dest	000 1011

Operation: `rD = mem[rs1]; rs1 += rs2`

3.3.9 lbu rD, rs2(rs1!)

Loads a byte from data memory on the address `rs1` and saves `(rs1 + rs2)` in register `rs1` after the load has completed.

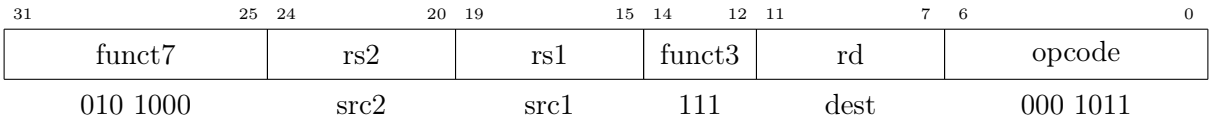


31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
010 0000	src2	src1	111	dest	000 1011

Operation: `rD = Zext(mem[rs1]); rs1 += rs2`

3.3.10 lhu rD, rs2(rs1!)

Loads a half-word from data memory on the address `rs1` and saves `(rs1 + rs2)` in register `rs1` after the load has completed.

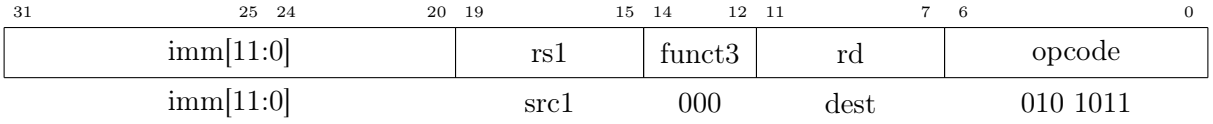


31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
010 1000	src2	src1	111	dest	000 1011

Operation: `rD = Zext(mem[rs1]); rs1 += rs2`

3.3.11 sb rs2, imm(rs1!)

Store a byte to data memory on the address `rs1` and saves `(rs1 + imm)` in register `rs1` after the store has completed.

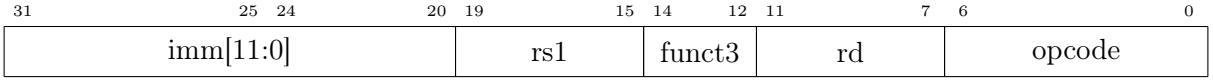


31 25	24 20	19 15	14 12	11 7	6 0
imm[11:0]		rs1	funct3	rd	opcode
imm[11:0]		src1	000	dest	010 1011

Operation: `mem[rs1] = rs2[7:0]; rs1 += Sext(imm)`

3.3.12 sh rs2, imm(rs1!)

Store a half-word to data memory on the address `rs1` and saves `(rs1 + imm)` in register `rs1` after the store has completed.

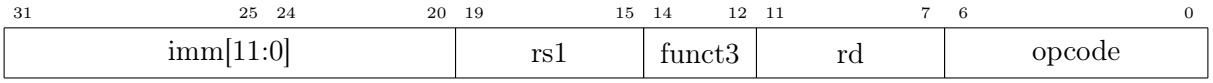


imm[11:0]	src1	001	dest	010 1011

Operation: `mem[rs1] = rs2[15:0]; rs1 += Sext(imm)`

3.3.13 sw rs2, imm(rs1!)

Store a word to data memory on the address `rs1` and saves `(rs1 + imm)` in register `rs1` after the store has completed.

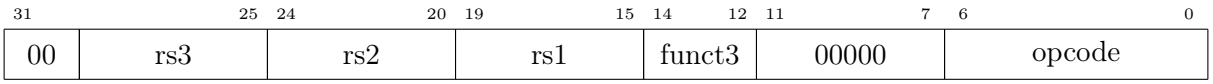


imm[11:0]	src1	010	dest	010 1011

Operation: `mem[rs1] = rs2[31:0]; rs1 += Sext(imm)`

3.3.14 sb rs2, rs3(rs1!)

Store a byte to data memory on the address `rs1` and saves `(rs1 + rs3)` in register `rs1` after the store has completed.

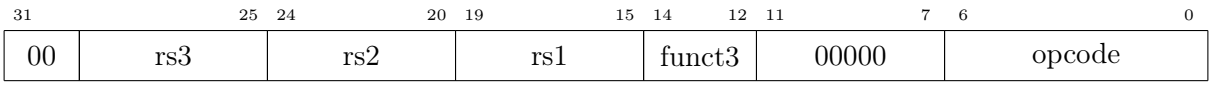


00	src3	src2	src1	100	dest	000 1011

Operation: `mem[rs1] = rs2[7:0]; rs1 += rs3`

3.3.15 sh rs2, rs3(rs1!)

Store a half-word to data memory on the address `rs1` and saves `(rs1 + rs3)` in register `rs1` after the store has completed.

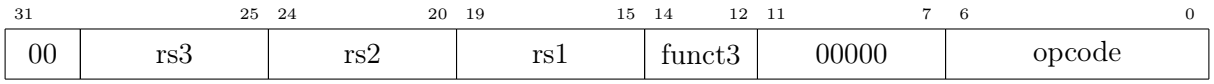


00	src3	src2	src1	101	dest	000 1011

Operation: `mem[rs1] = rs2[15:0]; rs1 += rs3`

3.3.16 sw rs2, rs3(rs1!)

Store a word to data memory on the address `rs1` and saves `(rs1 + rs3)` in register `rs1` after the store has completed.



00	src3	src2	src1	110	dest	000 1011

Operation: `mem[rs1] = rs2[31:0]; rs1 += rs3`

4 Multiply-Accumulate

RI5CY uses a single-cycle 32-bit × 32-bit multiplier with a 32-bit result. Only a subset of the standard M extension is implemented, i.e. the `mul` instruction. Divisions and multiplications that return the upper 32-bit of the result are not supported.

Specifically the following instruction is supported:

- `mul`

The following instructions are **not** supported:

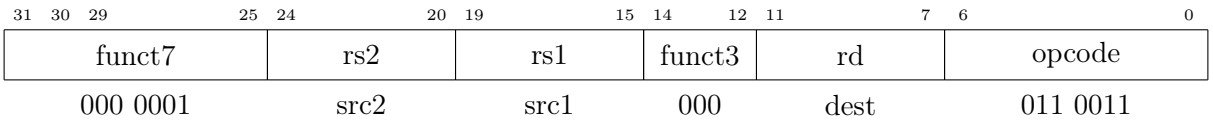
- `mulh`
- `mulhs`
- `mulhu`
- `div`
- `divu`
- `rem`
- `remu`

Instead RI5CY supports non-standard extensions for multiply-accumulate and half-word multiplications.

4.1 Instructions

4.1.1 p.mul rD, rs1, rs2

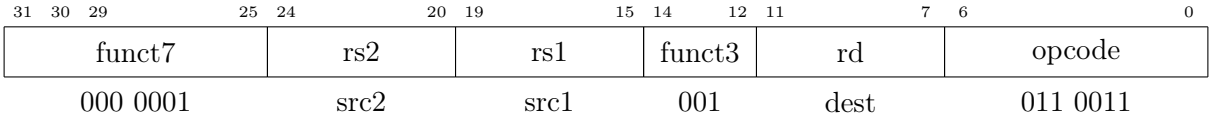
Multiply on 32-bit × 32-bit with a 32-bit result.



Operation: `rD = rs1 * rs2`

4.1.2 p.mac rD, rs1, rs2

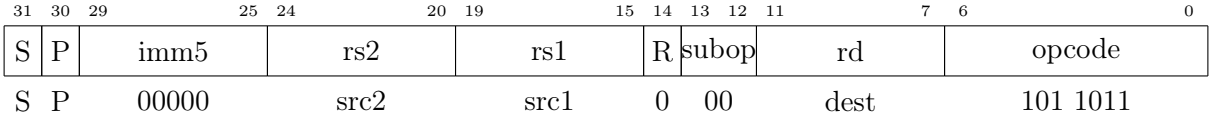
Multiply-Accumulate on 32-bit × 32-bit with a 32-bit result.



Operation: `rD += rs1 * rs2`

4.1.3 p.mul{s,hhs,u,hhu} rD, rs1, rs2

Multiply-Accumulate on 16-bit × 16-bit with a 32-bit result. The half-word and sign-mode that is used for the multiplication can be selected.

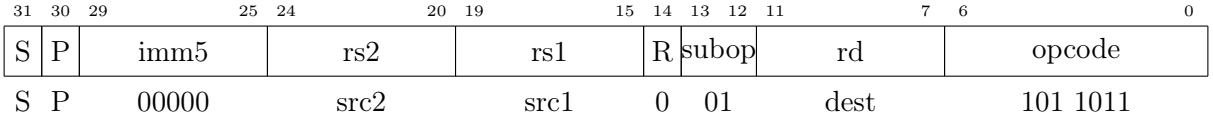


Operation: `rD = rs1[P*16+15:P*16] * rs2[P*16+15:P*16]`

S determines the zero/sign-extension of rs1 and rs2. A value of `1` means sign-extension.

4.1.4 p.mac{s,hhs,u,hhu} rD, rs1, rs2

Multiply-Accumulate on 16-bit × 16-bit with a 32-bit result. The half-word and sign-mode that is used for the multiplication can be selected.



Operation: `rD += rs1[P*16+15:P*16] * rs2[P*16+15:P*16]`

S determines the zero/sign-extension of rs1 and rs2. A value of `1` means sign-extension.

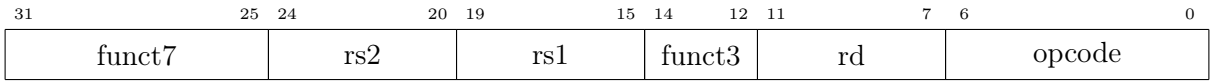
5 PULP ALU Extensions

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.

5.1 Instructions

5.1.1 p.avg rD, rs1, rs2

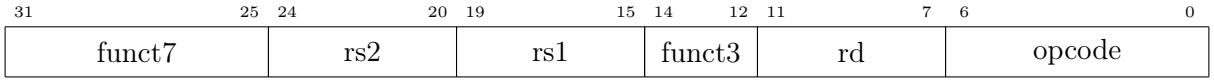
Performs an arithmetic right shift after the addition of rs1 and rs2.



Operation: `rD = (rs1 + rs2) >> 1`

5.1.2 p.avgu rD, rs1, rs2

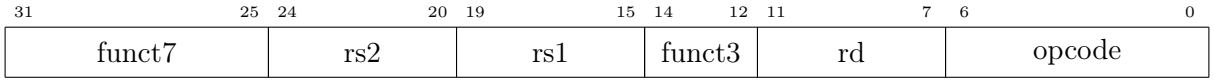
Performs a logical right shift after the addition of rs1 and rs2.



Operation: `rD = (rs1 + rs2) >> 1`

5.1.3 p.slet rD, rs1, rs2

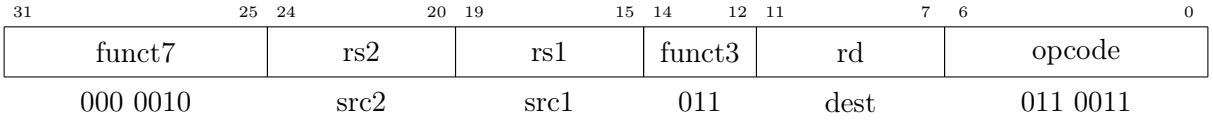
Performs an signed smaller than or equal comparison between rs1 and rs2.



Operation: `rD = (rs1 <= rs2) ?  -1 :  0`

5.1.4 p.sletu rD, rs1, rs2

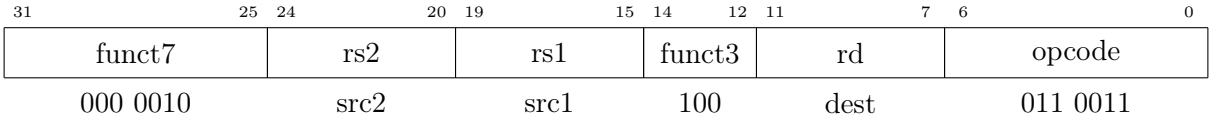
Performs an unsigned smaller than or equal comparison between rs1 and rs2.



Operation: `rD = (rs1 <= rs2) ? -1 : 0`

5.1.5 p.min rD, rs1, rs2

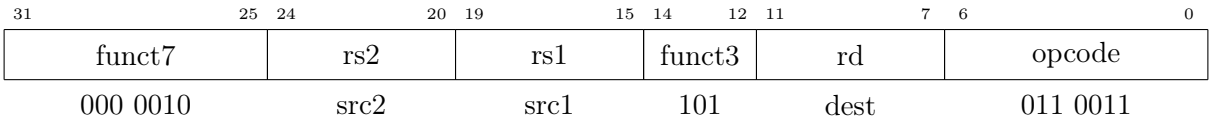
Sets rD to the minimum of rs1 and rs2, assuming both are signed 32-bit values.



Operation: `rD = rs1 < rs2 ? rs1 : rs2`

5.1.6 p.minu rD, rs1, rs2

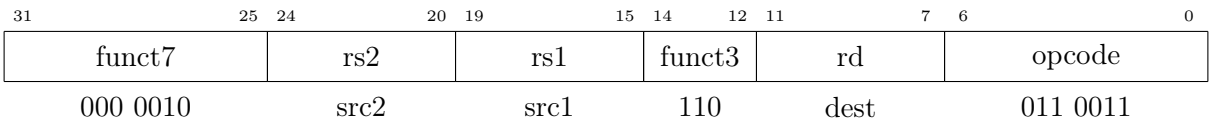
Sets rD to the minimum of rs1 and rs2, assuming both are unsigned 32-bit values.



Operation: `rD = rs1 < rs2 ? rs1 : rs2`

5.1.7 p.max rD, rs1, rs2

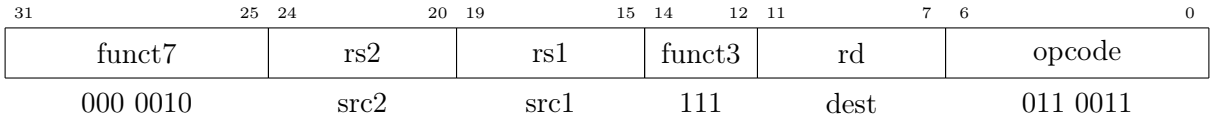
Sets rD to the maximum of rs1 and rs2, assuming both are signed 32-bit values.



Operation: `rD = rs1 > rs2 ? rs1 : rs2`

5.1.8 p.maxu rD, rs1, rs2

Sets rD to the maximum of rs1 and rs2, assuming both are unsigned 32-bit values.

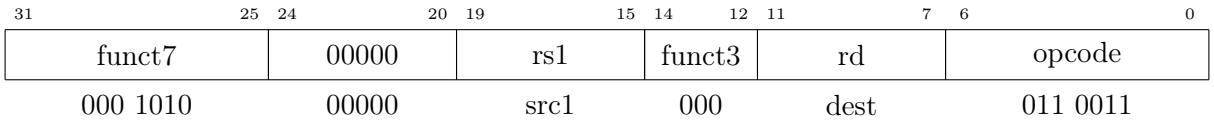


31–25	24–20	19–15	14–12	11–7	6–0
funct7	rs2	rs1	funct3	rd	opcode
000 0010	src2	src1	111	dest	011 0011

Operation: `rD = rs1 > rs2 ? rs1 : rs2`

5.1.9 p.abs rD, rs1

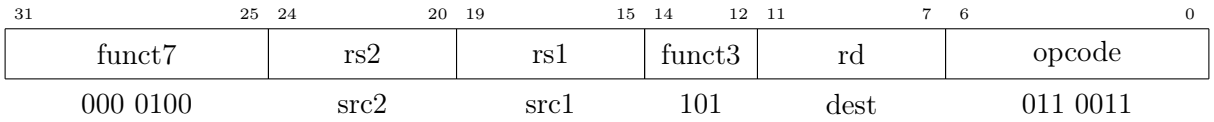
Computes the absolute value of the signed 32-bit operand rs1.



31–25	24–20	19–15	14–12	11–7	6–0
funct7	00000	rs1	funct3	rd	opcode
000 1010	00000	src1	000	dest	011 0011

Operation: `rD = rs1 < 0 ? -rs1 : rs1`

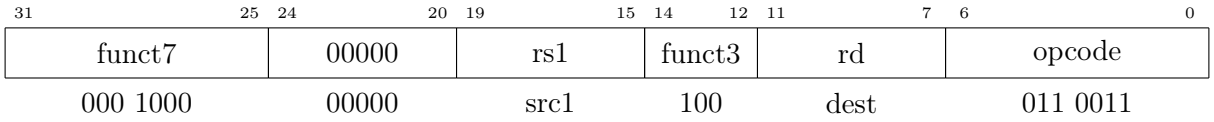
5.1.10 p.ror rD, rs1, rs2



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

Operation: `rD = RotateRight(rs1, rs2)`

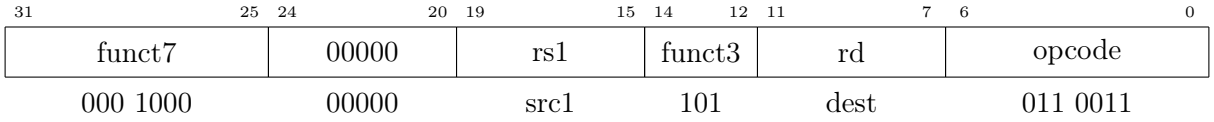
5.1.11 p.exths rD, rs1



31–25	24–20	19–15	14–12	11–7	6–0
funct7	00000	rs1	funct3	rd	opcode
000 1000	00000	src1	100	dest	011 0011

Operation: `rD = Sext(rs1[15:0])`

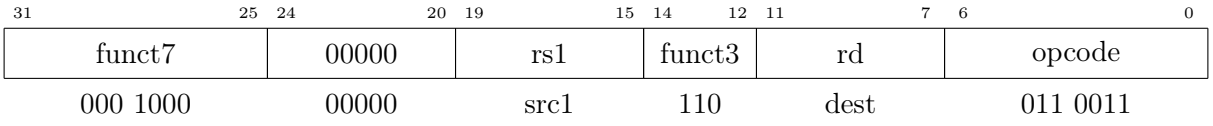
5.1.12 p.exthz rD, rs1



31–25	24–20	19–15	14–12	11–7	6–0
funct7	00000	rs1	funct3	rd	opcode
000 1000	00000	src1	101	dest	011 0011

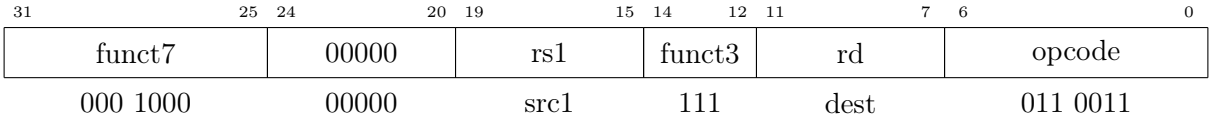
Operation: `rD = Zext(rs1[15:0])`

5.1.13 p.extbs rD, rs1



Operation: `rD = Sext(rs1[7:0])`

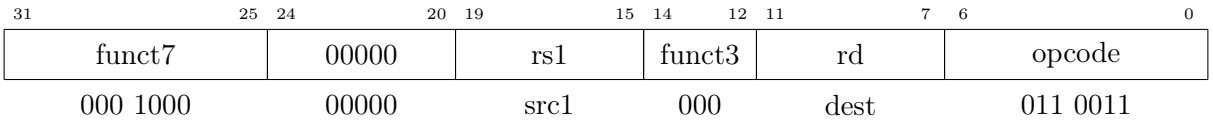
5.1.14 p.extbz rD, rs1



Operation: `rD = Zext(rs1[7:0])`

5.1.15 p.ff1 rD, rs1

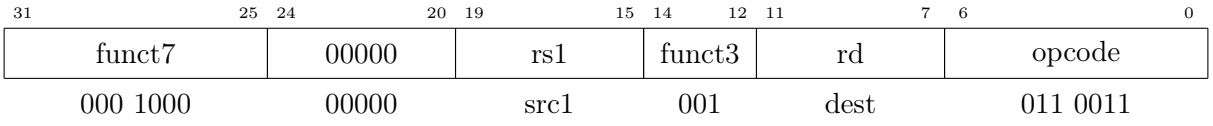
Returns position of first bit that is `1` starting from LSB, 32 if none.



Operation: `rD = FindFirst1(rs1)`

5.1.16 p.fl1 rD, rs1

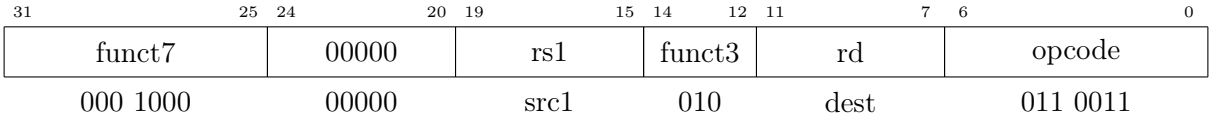
Returns position of first bit that is `1` starting from MSB, 32 if none.



Operation: `rD = FindLast1(rs1)`

5.1.17 p.clb rD, rs1

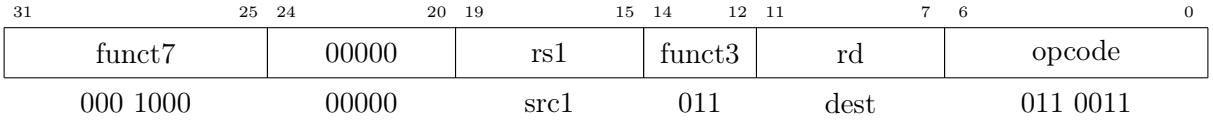
Count leading bits in rs1, i.e. the number of consecutive `1` or `0` bits from MSB.



Operation: `rD = CountLeadingBits(rs1)`

5.1.18 p.cnt rD, rs1

Count the number of bits set to `1` in rs1. This is also known as population count.



Operation: `rD = PopCount(rs1)`

6 PULP Hardware Loop Extensions

For increased 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, each of them can store these three values. 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.

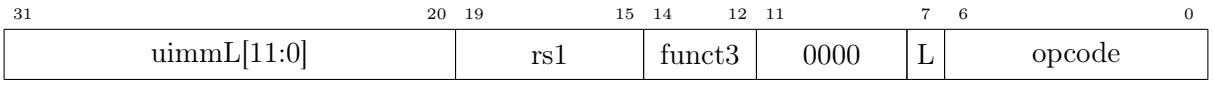
The instructions described below are used to setup the hardware loop registers. Note that the minimum loop size is two instructions.

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.

6.1 Instructions

6.1.1 lp.starti L, uimmL

Sets the start address of a hardware loop.

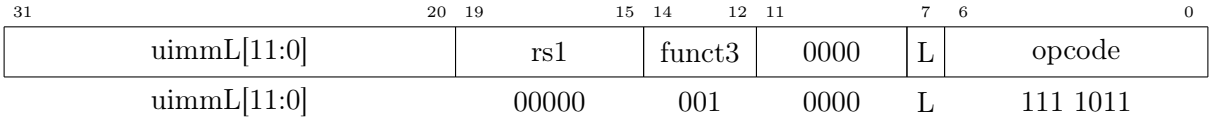


uimmL[11:0]	00000	000	0000	L	111 1011

Operation: `lpstart[L] = pc + (Zext(uimmL[11:0]) « 1)`

6.1.2 lp.endi L, uimmL

Sets the end address of a hardware loop.

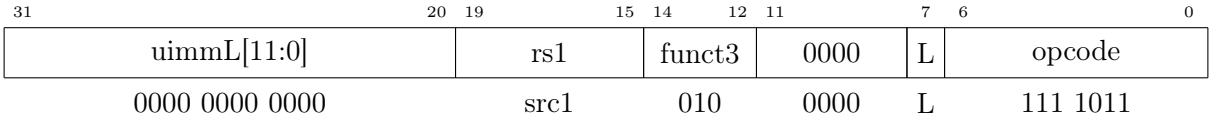


uimmL[11:0]	00000	001	0000	L	111 1011

Operation: `lpend[L] = pc + (Zext(uimmL[11:0]) « 1)`

6.1.3 lp.count L, rs1

Sets the number of iterations of a hardware loop.

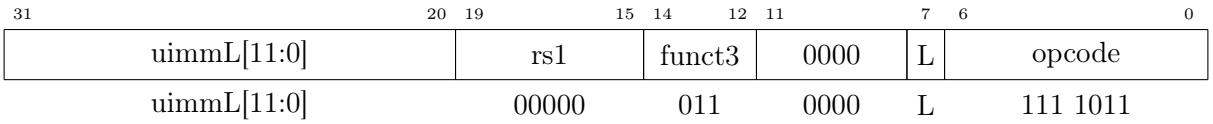


0000 0000 0000	src1	010	0000	L	111 1011

Operation: `lpcount[L] = rs1`

6.1.4 lp.counti L, uimmL

Sets the number of iterations of a hardware loop.

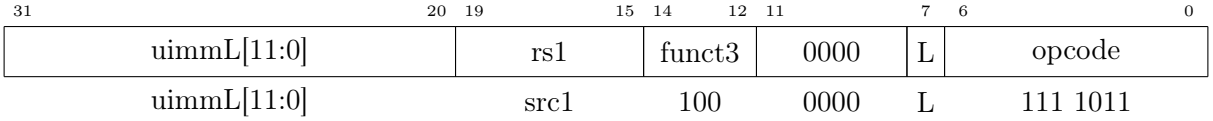


uimmL[11:0]	00000	011	0000	L	111 1011

Operation: `lpcount[L] = Zext(uimmL[11:0])`

6.1.5 lp.setup L, rs1, uimmL

Sets up a hardware loop in one instruction. This instruction assumes that the next instruction is the start address of the loop.

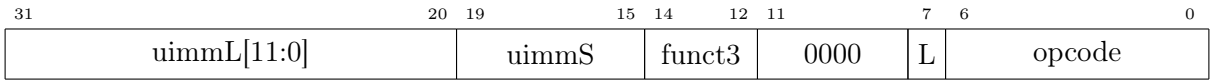


uimmL[11:0]	src1	100	0000	L	111 1011

Operation: `lpstart[L] = pc + 4; lpend[L] = pc + Zext(uimmL[11:0]); lpcount[L] = rs1`

6.1.6 lp.setupi L, uimmS, uimmL

Sets up a hardware loop in one instruction. This instruction assumes that the next instruction is the start address of the loop. The number of iterations is given as an immediate.



uimmL[11:0]	uimmS	101	0000	L	111 1011

Operation: `lpstart[L] = pc + 4; lpend[L] = pc + Zext(uimmL[11:0]); lpcount[L] = Zext(uimmS)`

6.2 CSR Mapping

Table 6.1: Control and Status Register Map

CSR Address		Hex	Name	Access	Description
[11:6]	[5:0]				
011110	110000	0x7B0	lpstart[0]	R/W	Hardware Loop 0 Start
011110	110001	0x7B1	lpend[0]	R/W	Hardware Loop 0 End
011110	110010	0x7B2	lpcount[0]	R/W	Hardware Loop 0 Counter
011110	110100	0x7B4	lpstart[1]	R/W	Hardware Loop 1 Start
011110	110101	0x7B5	lpend[1]	R/W	Hardware Loop 1 End
011110	110110	0x7B6	lpcount[1]	R/W	Hardware Loop 1 Counter

7 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 7.1 shows the control signals relevant for the pipeline operation. Running from right to left are the main control signals, the `ready` signals of each pipeline stage. 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. At every pipeline stage, when the `ready` coming from the stage to the right is high, but the valid signal of the stage is low, the stage is cleared. If the valid signal is high, it is enabled.

Going from right to left every stage is independent, no stage depends on the stage on its left. Going from left to right every stage depends on its right neighbor and can only continue when it is ready. This means that in addition that a single stage has to be ready, also the stage on its right has to be ready to move on.

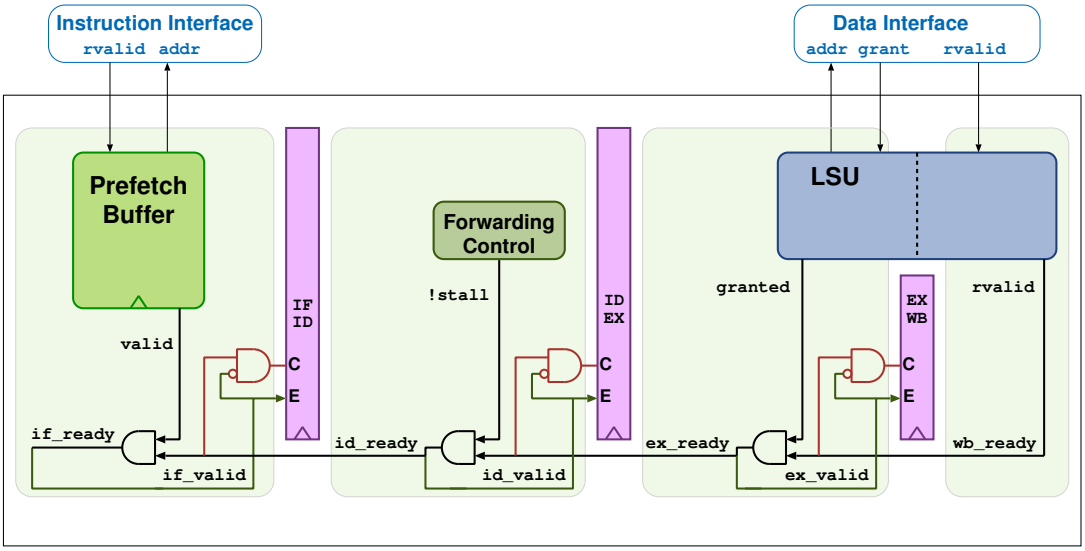


Figure 7.1: RI5CY Pipeline.

Note

In contrast to Or10n there is no global stall controller any more, instead every stage manages its own dependencies. The main controller of RI5CY is now only responsible for control flow operations, i.e. flushing the pipe, branches, jumps and exceptions.

8 Register File

RI5CY has 31 × 32-bit wide registers which form registers `x1` to `x31`. Register `x0` is statically bound `0` and can only be read and not written, 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.

8.1 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 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 explicitely need.

Table 9.1: Control and Status Register Map

CSR Address				Hex	Name	Access	Description
[11:10]	[9:8]	[7:6]	[5:0]				
00	11	00	000000	0x300	MSTATUS	R/W	Machine Status Register
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 Register
01	11	10	100001	0x7A1	PCMR	R/W	Performance Counter Mode Register
01	11	10	110XXX	0x7B0 - 0x7B6	HWLP	R/W	Hardware Loop Registers
01	11	10	111000	0x7C0	MESTATUS	R/W	Machine exception Status Register
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

9.1 Register Description

9.1.1 MSTATUS

CSR Address: `0x300`
Reset Value: `0x0000_0006`

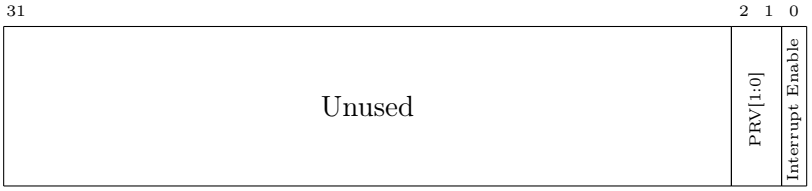


Figure 9.1: MSTATUS

Note that `PRV[1:0]` is statically `2'b11` and cannot be altered (read-only). When en 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 hanlder, set the `Interrupt Enable` to `1'b1` inside your handler code.

9.1.2 MESTATUS

CSR Address: `0x7C0`
Reset Value: `0x0000_0006`

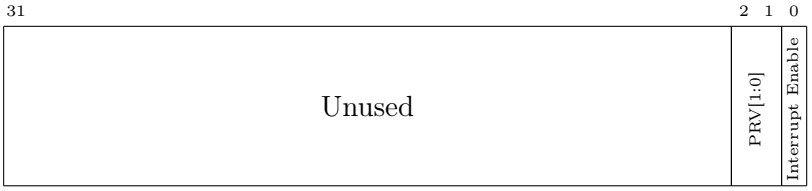


Figure 9.2: MESTATUS

Note that `PRV[1:0]` is statically `2'b11` and cannot be altered (read-only).

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.

9.1.3 MEPC

CSR Address: `0x341`
Reset Value: `0x0000_0000`

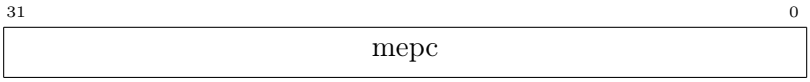


Figure 9.3: MEPC

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.

9.1.4 MCAUSE

CSR Address: `0x341`
Reset Value: `0x0000_0000`

Figure 9.4: MCAUSE

9.1.5 MCPUID

CSR Address: `0xF00`
Reset Value: `0x0000_0100`

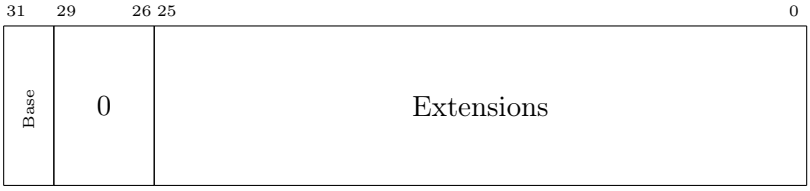


Figure 9.5: MCPUID

9.1.6 MIMPID

CSR Address: `0xF01`
Reset Value: `0x0000_8000`

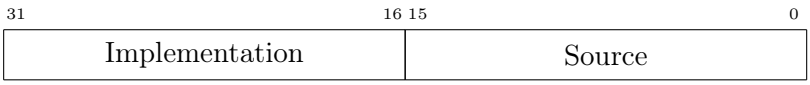


Figure 9.6: MIMPID

9.1.7 MHARTID

CSR Address: `0xF10`
Reset Value: `Defined`

Figure 9.7: MHARTID

Both `core id` and `cluster id` are set on the top-level module of the core and are read-only.

10 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.

10.1 Performance Counter Mode Register (PCMR)

CSR Address: `0x7A1`
Reset Value: `0x0000_0003`

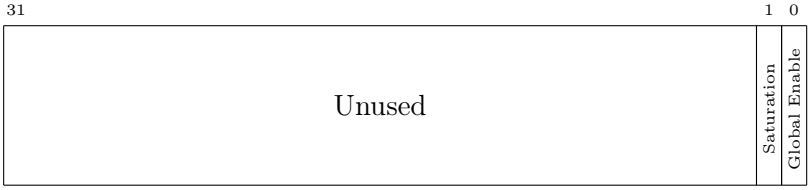


Figure 10.1: PCMR

The `Global Enable` bit controls all performance counters, i.e. if it is set to `0`, all performance counters are deactivated. After reset, the `Global Enable` bit is set.

The `Saturation` bit controls saturation behaviour of the performance counters. If it is set, saturating arithmetic is used. After reset, the `Saturation` bit is set.

10.2 Performance Counter Event Register (PCER)

CSR Address: `0x7A0`
Reset Value: `0x0000_0000`

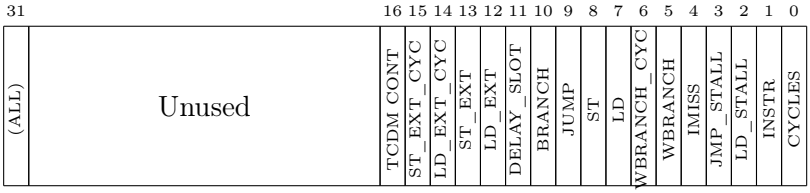


Figure 10.2: PCER

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 are 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 the FPGA or Simulation version each event has its own counter and can be accesses separately.

10.3 Performance Counter Counter Registers (PCCR0-31)

CSR Address: `0x780 - 0x79F`
Reset Value: `0x0000_0000`

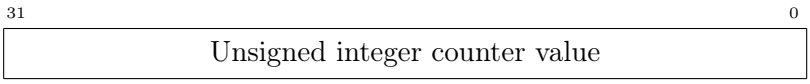


Figure 10.3: PCCR0-31

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

Reg Name	Name	Description
PCCR0	**CYCLES**	Count the number of cycles the core was running
PCCR1	**INSTR**	Count 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. the number of instructions wasted due to non-ideal caches
PCCR5	**LD**	Number of memory loads executed. Misaligned accesses are counted twice
PCCR6	**ST**	Number of memory stores executed. Misaligned accesses are counted twice
PCCR7	**JUMP**	Number of 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

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

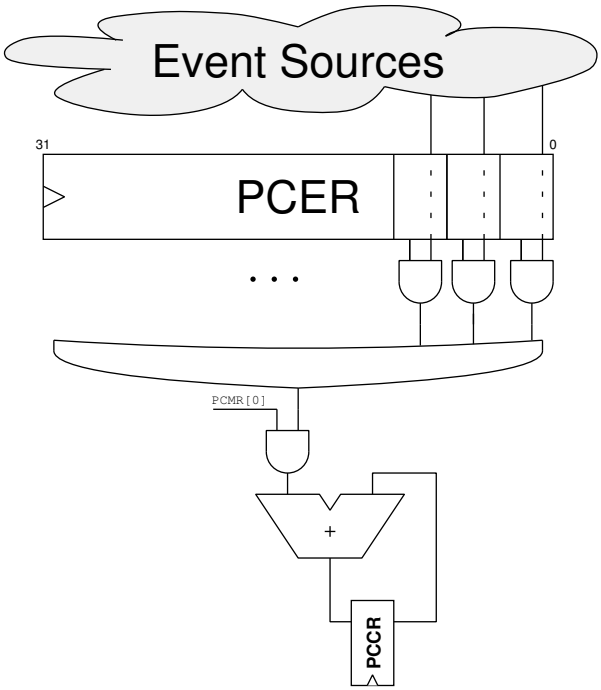


Figure 10.4: Events and PCCR, PCMR and PCER on the ASIC.

11 Exceptions and Interrupts

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

Table 11.1: Interrupt/exception offset vector table

Address	Description
0x00 - 0x0000_007C	Interrupts 0 - 31
0x80	Reset
0x84	Illegal Instruction
0x88	ECALL instruction executed
0x8C	LSU error (invalid memory access)

The instruction addresses in Table 11.1 are considered as an offset to the boot address given to the core. Specifically the core jumps to address {boot_addr[31:8], offset[7:0]} when encountering an exception/interrupt.

11.1 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 in 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 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.

11.2 Exceptions

The illegal instruction exception, the load and store invalid memory access exceptions and ecall instruction exceptions can not 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.

11.3 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 explicitely 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`.

12 Debug

RI5CY has full support for software breakpoints (`ebreak`), access to general-purpose and control and status registers via a debug port. It is also possible to halt the core from the debug port and put it into single-stepping mode. Similarly when an interrupt occurs, instead of jumping to the interrupt handler the core can trap to an attached debugger.

The debug port uses the following interface:

Table 12.1: Debug Signals

Signal	Direction	Description
`dbginf_strobe_i`	**input**	Command request
`dbginf_we_i`	**input**	Write Enable
`dbginf_addr_i[15:0]`	**input**	Address
`dbginf_data_i[31:0]`	**input**	Input data
`dbginf_data_o`	**output**	Output data
`dbginf_ack_o`	**output**	Command was executed
`dbginf_stall_i`	**input**	Stall the core
`dbginf_bp_o`	**output**	Breakpoint hit

This interface is natively supported by the advanced debug bridge that is used by PULP and PULPino, see also the documentation of this bridge.

12.1 Debug Address Map

This debug address map is not optimal and should be changed! See the OpenSoC debug project for a proposal for a better address map.

Table 12.2: Control and Status Register Map

Dbginf Addr [15:0]		Hex	Name	Description
Grp [15:11]	Addr [10:0]			
`0_0001`	`000_000X_XXXX`	`0x0800 - 0x081F`	GPR	General-Purpose Registers
`0_0110`	`000_0000_0000`	`0x3000 - 0x3014`	Debug	Debug Registers
`?_????`	`XXX_XXXX_XXXX`		CSR	Everything else is mapped to CSR

12.1.1 Debug Register: DMR1

CSR Address: `0x3010`
Reset Value: `0x0000_0000`

Figure 12.1: DMR1

Single-stepping activates single-stepping mode, meaning the core traps to the debugger after one instruction has been executed.

12.1.2 Debug Register: DSR

CSR Address: `0x3014`
Reset Value: `0x0000_0000`

Figure 12.2: DMR1

`IIE` stands for illegal instruction exception enabled. A value of `1` means trap to the debugger when an illegal instruction is encountered.

`INTE` stands for interrupt enabled. A value of `1` means trap to the debugger when an interrupt is encountered.