SafePMU User's Manual

Guillem Cabo Pitarch

December 21, 2020

CONTENTS

1 OVERVIEW

The SafePMU (Safe Performance Monitoring Unit) is an AHB slave capable of monitoring SoC events, enforce contention control, and identifying profiling errors on run-time. Figure "1.1" shows the structure of the unit. It is composed of an ahb wrapper (*ahb_wrapper.vhd*) that maps the SystemVerilog implementation into a VHDL module that can be instanced in SELENE and De-RISC SoCs. The SystemVerilog AHB interface (*pmu_ahb.sv*) offers support for a subset of AHB requests. This module also instances the interface agnostic PMU (*PMU_raw.sv*). The latter is used as the generator of the statistic unit. It generates the memory map and the instances for each of the features.

The main features are:

- **Self-test:** Allows to configure the counters' inputs to a fixed value bypassing the crossbar and ignoring the inputs. This mode allows for tests of the software and the unit under known conditions.
- **Crossbar:** Allows to route any input event to any counter.
- **Counters:** Group of simple counters with settable initial values and general control register.
- **Overflow:** Detection of overflow for counters, interrupt capable with dedicated interruption vector and per counter interrupt enable.
- **Quota:** Deprecated *(It may be excluded in a future release)*
- **MCCU** (Maximum Contention Control Unit) : Contention control measures for each core. Interruption capable after a threshold of contention is exceeded. It accepts real contention signals or estimation through weights.
- **RDC** (Request Duration Counters): Provides measures of the pulse length of a given input signal (watermark). It can be used to determine maximum latency and cycles of uninterrupted contentions. Each of the counters can trigger an interrupt at a user-defined threshold.

The default configuration of this unit supports 4 cores and 32 input signals. In future releases such parameters (VHDL generics) will be exposed to the top level.

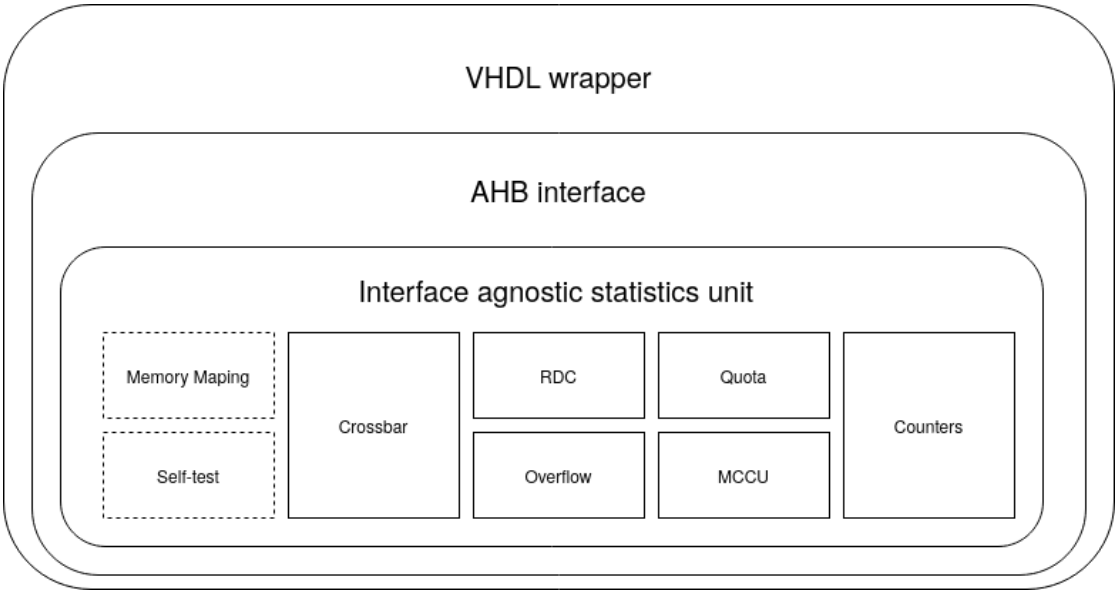


Figure 1.1: Block diagram statistics unit structure

2 OPERATION

2.1 GENERAL

The SafePMU attaches to a 128-bit wide AHB bus but only supports Single burst 32-bit accesses. AHB Lock accesses and protection control are not implemented.

The unit is compatible with GRLIB *plug&play*(P&P). Vendor and device's default configuration values are *"BSC"* and *" AHB Performance Monitoring Unit"* respectively. Regardless, release *v3.2.4* of GRMON, the unit may appear as *"Unknown device"*.

2.2 DEFAULT INPUT EVENTS

In its default implementation, the unit provides up to 32 input events. Each one of the events can be routed to any mechanism of the module through the crossbar.

Table 2.1 shows the inputs and mapping to the crossbar input for the current SELENE release. The number of signals and arrangements may change in the upcoming versions.

Table 2.1: SELENE default input events

Index	Name	Type	Source	Description
0	'1'	Debug	Local	Constant HIGH signal, used for debug purposes or clock cycles
1	'0'	Debug	Local	Constant LOW signal, used for debug purposes
2	pmu_events(0).icnt(0)	Pulse	Core 0	Instruction count pipeline 0
3	pmu_events(0).icnt(1)	Pulse	Core 0	Instruction count pipeline 1
4	pmu_events(0).icmiss	Pulse	Core 0	Instruction cache miss
5	pmu_events(0).dcmiss	Pulse	Core 0	Data cache L1 miss
6	pmu_events(0).bpmiss	Pulse	Core 0	Branch Perdictor miss
7	ccs_contention(0).r_and_w	CCS	Core 0	Contention caused to core 0 due to core 1 AHB read or write accesses
8	ccs_contention(0).read	CCS	Core 0	Contention caused to core 0 due to core 1 AHB read accesses
9	ccs_contention(0).write	CCS	Core 0	Contention caused to core 0 due to core 1 AHB write accesses
10	ccs_latency(0).total	CCS	Core 0	-
11	ccs_latency(0).dcmiss	CCS	Core 0	-
12	ccs_latency(0).icmiss	CCS	Core 0	-
13	ccs_latency(0).write	CCS	Core 0	-
14	pmu_events(1).dcmiss	Pulse	Core 1	Data cache L1 miss
15	ccs_contention(1).r_and_w	CCS	Core 1	Contention caused to core 0 due to core 2 AHB read or write accesses
16	ccs_contention(1).read	CCS	Core 1	Contention caused to core 0 due to core 2 AHB read accesses
17	ccs_latency(1).total	CCS	Core 1	-
18	ccs_latency(1).dcmiss	CCS	Core 1	-
19	ccs_latency(1).write	CCS	Core 1	-
20	pmu_events(2).dcmiss	Pulse	Core 2	Data cache L1 miss
21	ccs_contention(2).r_and_w	CCS	Core 2	Contention caused to core 0 due to core 3 AHB read or write accesses
22	ccs_contention(2).read	CCS	Core 2	Contention caused to core 0 due to core 3 AHB read accesses
23	ccs_latency(2).total	CCS	Core 2	-
24	ccs_latency(2).dcmiss	CCS	Core 2	-
25	ccs_latency(2).write	CCS	Core 2	-
26	pmu_events(3).dcmiss	Pulse	Core 3	Data cache L1 miss
27	ccs_contention(1).write	CCS	Core 1	Contention caused to core 0 due to core 2 AHB write accesses
28	ccs_contention(2).write	CCS	Core 2	Contention caused to core 0 due to core 3 AHB write accesses
29	ccs_latency(3).total	CCS	Core 3	-
30	ccs_latency(3).dcmiss	CCS	Core 3	-
31	ccs_latency(3).write	CCS	Core 3	-

Signals labeled as *debug* are fix inputs that can be used to test hardware or software with known inputs. **Event 0** (fix '1') can be used to measure the **number of elapsed cycles**.

Signals of type *CCS* can be used to compute the total contention cycle stack of the system. These signals become high at the first rising edge of the clock after a given condition or event has been detected. They remain active until the condition or event that they are measuring becomes low. This behavior allows measuring the length of clock cycles. When generating CCS signals, the user must consider if they want to allow back to back events and generate the input signals accordingly at RTL level.

2.3 MAIN CONFIGURATION AND SELF-TEST

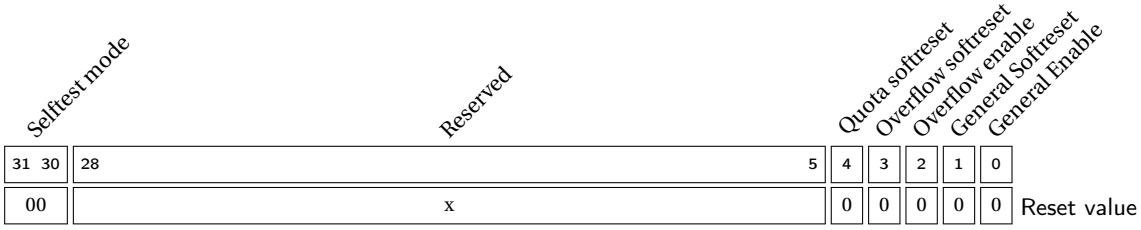
Reset and enable of overflow, quota, and regular counters can be performed with register 2.1. All signals are active high.

Self-test mode allows to bypass the input events from the crossbar and instead use a specific input pattern where signals are constant. This mode can be used for debugging. After the addition of the crossbar and debug inputs, there is a certain overlap. The same results can be achieved with the correct crossbar configuration. Nevertheless, it has been included in this release for compatibility.

These are the Self-test modes for each configuration value of the fied *selftest mode* in register 2.1:

- 0b00: Events depend on the crossbar. Self-test is disabled
- 0b01: All signals are set to 1.
- 0b10: All signals are set to 0.
- 0b11: Signal 0 is set to 1. The remaining signals are set to 0.

Register 2.1: BASE CONFIGURATION REGISTER (0x000)



2.4 CROSSBAR

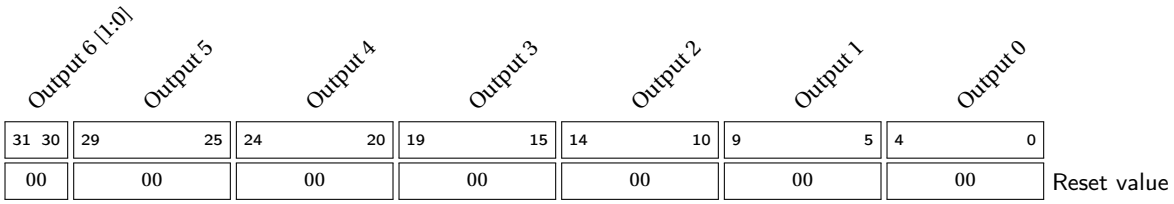
This feature allows routing any of the input signals of table 2.1 into any of the 24 counters of the PMU. Each one of the counters has a 5-bit configuration value. This values are stored in registers 2.2, 2.3, 2.4,2.5. All the configuration values are consecutive. Thus some values may have configuration bits in two consecutive memory addresses. Examples of this are Output 6, 12, 19 in our current configuration. As a consequence, the previous outputs may require two writes to configure the desired input signal.

Configuration fields match one to one with the internal counters. So the field *output 0* matches with *counter 0*, *output 1* with *counter 1* and so on.

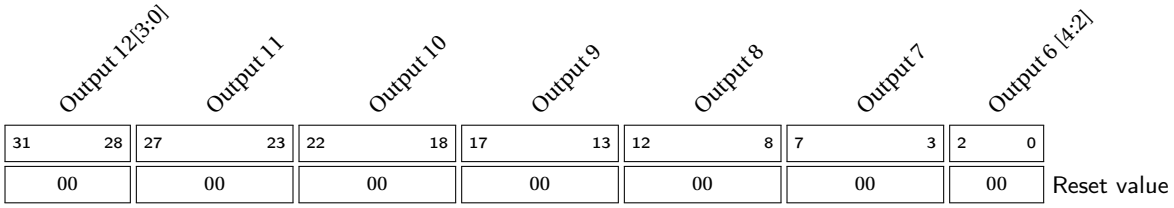
As a usage example, suppose the user wants to route the signal *pmu_events(0).icnt(0)* to the internal *counter 0*. The field *Output 0* of register 2.2 shall match the Index of the signal in table 2.1. In this case, the index is 2. After this configuration, the event count will be recorded

in *counter 0*. The addresses for counter values are indicated in figure 2.1.

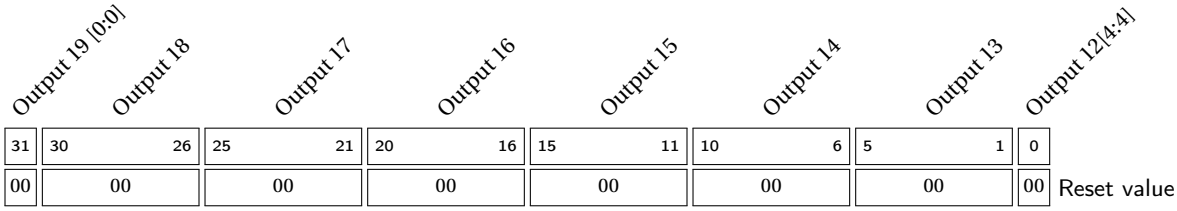
Register 2.2: CROSSBAR CONFIGURATION REGISTER 0 (0x0AC)



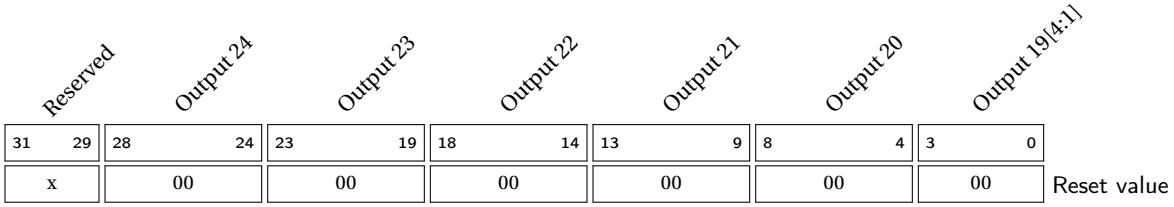
Register 2.3: CROSSBAR CONFIGURATION REGISTER 1 (0x0B0)



Register 2.4: CROSSBAR CONFIGURATION REGISTER 2 (0x0B4)



Register 2.5: CROSSBAR CONFIGURATION REGISTER 3 (0x0B8)



Signal routing is important since some of the PMU features are only available at different crossbar outputs. Table 2.2 shows the available capabilities for each one of the outputs.

Table 2.2: Crossbar outputs and PMU capabilities

Output	**Counters**	**Overflow**	**MCCU**	**RDC**
0	Yes	Yes	Core 0	Yes
1	Yes	Yes	Core 0	Yes
2	Yes	Yes	Core 1	Yes
3	Yes	Yes	Core 1	Yes
4	Yes	Yes	Core 2	Yes
5	Yes	Yes	Core 2	Yes
6	Yes	Yes	Core 3	Yes
7	Yes	Yes	Core 3	Yes
8	Yes	Yes	No	No
9	Yes	Yes	No	No
10	Yes	Yes	No	No
11	Yes	Yes	No	No
12	Yes	Yes	No	No
13	Yes	Yes	No	No
14	Yes	Yes	No	No
15	Yes	Yes	No	No
16	Yes	Yes	No	No
17	Yes	Yes	No	No
18	Yes	Yes	No	No
19	Yes	Yes	No	No
20	Yes	Yes	No	No
21	Yes	Yes	No	No
22	Yes	Yes	No	No
23	Yes	Yes	No	No

2.5 COUNTERS

The unit in the default configuration contains 24 counters, 32-bit each. Figures 2.1 and 2.2 indicate the memory address where each counter's value can be access. Counter values can be **read** or **written**, thus allowing to set the initial value of the counters.

Enable and reset is managed by the base configuration register 2.1.

Counters can overflow. In such a case, the count will wrap back to 0 and keep counting. Section 2.6 describes how to enable the overflow detection interrupts.

Figure 2.1: 0 to 12 Counter adresses

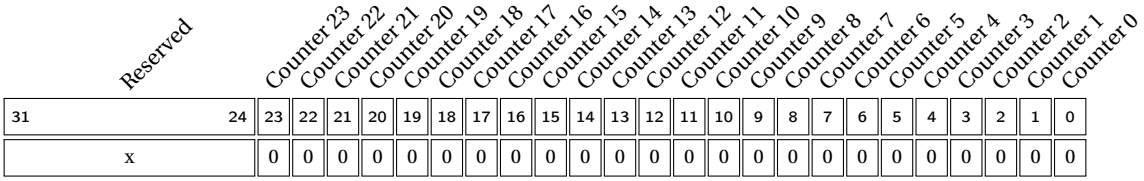
Figure 2.2: 12 to 24 Counter addresses

2.6 OVERFLOW

The user can enable overflow detection for each one of the counters in section 2.5. Enables are active high and individual for each counter, as indicated in register 2.6. If a counter with overflow detection active wraps over the maximum value, the corresponding bit of register 2.7 will become 1, and AHB interrupt number 6 will become active.

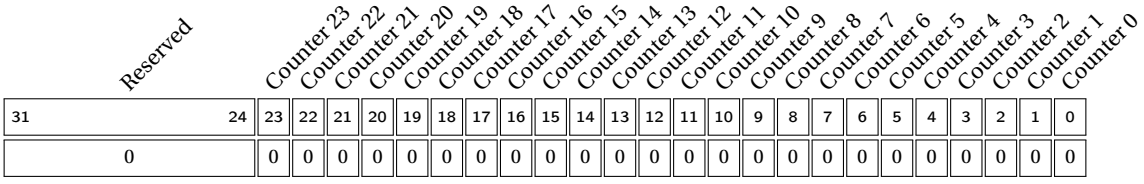
The default AHB interrupt mapping can be modified within the file *ahb_wrapper.vhd*.

Register 2.6: OVERFLOW INTERRUPT ENABLE MASK (0x064)



	Reserved	Counter 23	Counter 22	Counter 21	Counter 20	Counter 19	Counter 18	Counter 17	Counter 16	Counter 15	Counter 14	Counter 13	Counter 12	Counter 11	Counter 10	Counter 9	Counter 8	Counter 7	Counter 6	Counter 5	Counter 4	Counter 3	Counter 2	Counter 1	Counter 0
31 24 / 23		23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset value	x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Register 2.7: OVERFLOW INTERRUPT VECTOR (0x068)



	Reserved	Counter 23	Counter 22	Counter 21	Counter 20	Counter 19	Counter 18	Counter 17	Counter 16	Counter 15	Counter 14	Counter 13	Counter 12	Counter 11	Counter 10	Counter 9	Counter 8	Counter 7	Counter 6	Counter 5	Counter 4	Counter 3	Counter 2	Counter 1	Counter 0
31 24 / 23		23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

2.7 QUOTA

This feature has been replaced by the MCCU and will disappear in future releases. Usage is not recommended.

2.8 MCCU

The Maximum-Contention Control Unit (MCCU) allows to monitor a subset of the input events and track the approximate contention that they will cause. Currently, events assigned to counters 0 to 7 can be used as inputs of the MCCU. Thanks to the crossbar, any of the 32 SoC signals can be used by the MCCU.

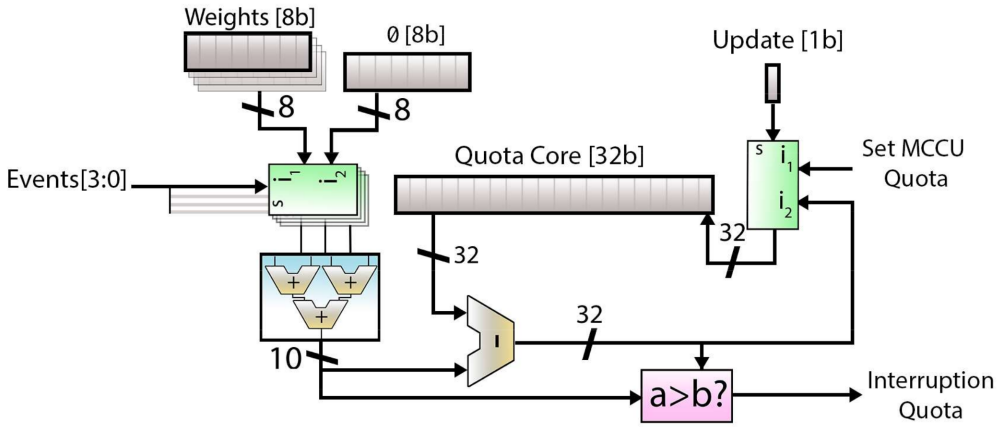


Figure 2.3: Block diagram MCCU mechanism for one core.

Figure 2.3 shows the internal elements required to monitor the quota consumption of one core, given four input events. When events are active, they pass the value assigned in the weight register 2.9 for the given signal to a series of adders. The addition is subtracted from the corresponding quota register 2.5. When the remaining quota is smaller than the cycle contention, an interrupt is triggered.

Register 2.8: MCCU MAIN CONFIGURATION (0x074)

Reserved										
	Soft reset RDC	Enable RDC	Update Quota Core 3	Update Quota Core 2	Update Quota Core 1	Update Quota Core 0	Soft reset MCCU	Enable reset MCCU		
32	8	7	6	5	4	3	2	1	0	
x		0	0	0	0	0	0	0	0	

Reset value

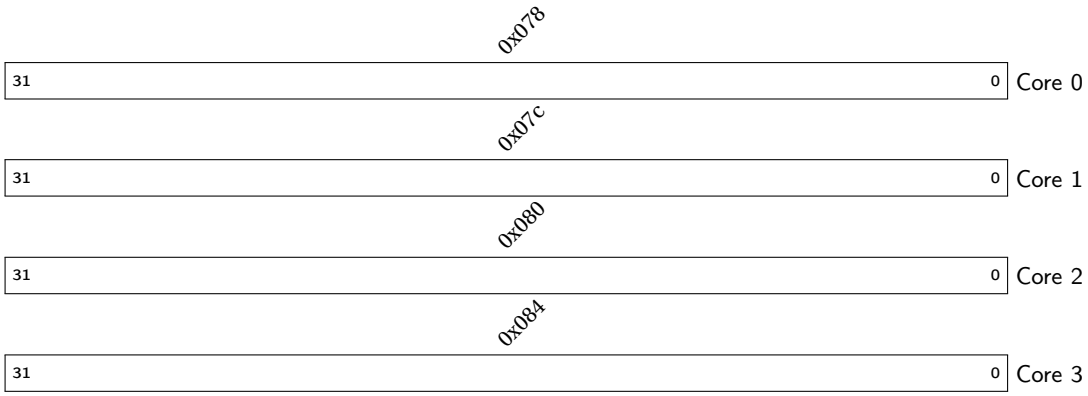


Figure 2.4: MCCU Quota limits for each core

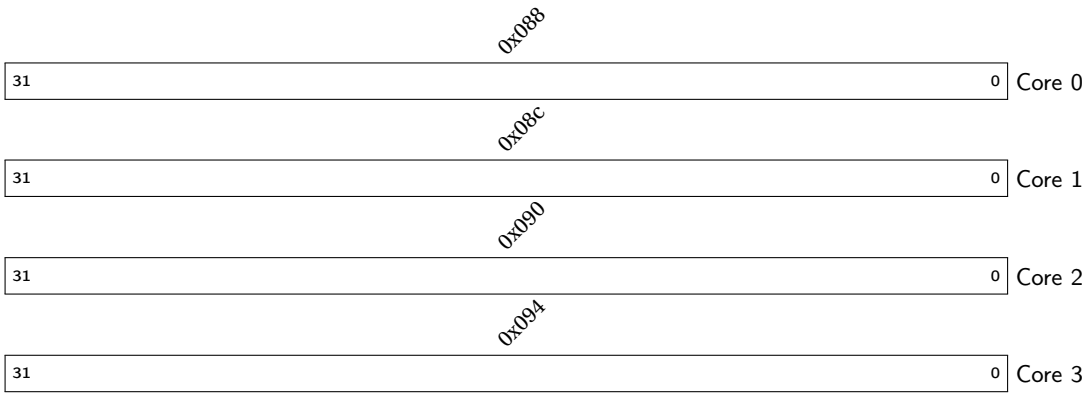


Figure 2.5: MCCU Current remaning Quota for each core

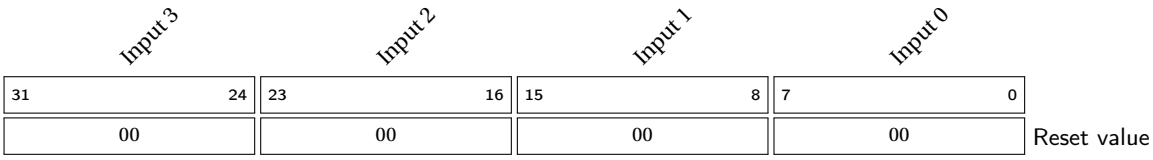
In the current release, the MCCU can be reset and activated with the respective fields of register 2.8. The fields labeled as *Update Quota core* are used to update the available quota of each core (figure 2.5). While *Update Quota core* is high, the content is assigned to the available quota. Once released (low), the available quota can start to decrease if the MCCU is active. The current quota can be read while the unit is active.

In the current release, each core can monitor two input events. The MCCU module is parametric and more events can be provided in future releases. Table 2.2 shows the features available for each crossbar output. Under the column MCCU, you can see towards which core quota the event will be computed. The unit provides one interruption for each of the monitored cores. Quota exhaustion for cores 3, 2, 1, and 0 is mapped to AHB interrupts 10, 9, 8, and 7, respectively.

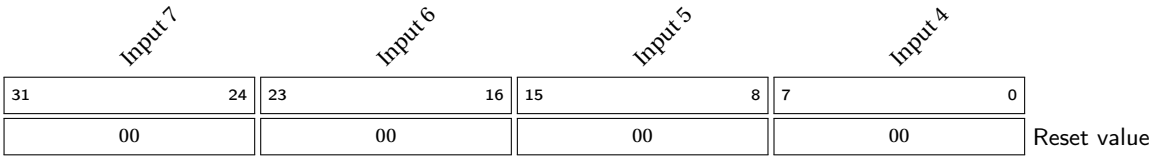
Weights for each monitored event are registered in registers 2.9 and 2.10. Currently, each

weight is an 8-bit field. Each input of the MCCU maps directly to the outputs of the crossbar. Thus the weight for the MCCU input 0 corresponds to the signal in crossbar output 0.

Register 2.9: MCCU EVENT WEIGHTS REGISTER 0 (SHARED WITH RDC) (0x098)



Register 2.10: MCCU EVENT WEIGHTS REGISTER 1 (SHARED WITH RDC) (0x09c)



2.9 RDC

The Request Duration Counter or RDC is comprised of a set of 8-bit counters and comparators that allow monitoring the length of a CCS signal, record the number of clock cycles of the longest pulse and compare such value with the defined weight.

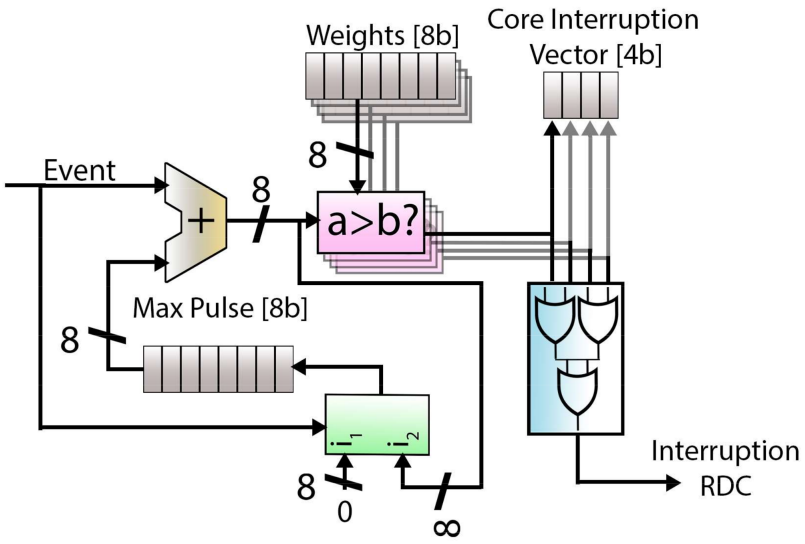


Figure 2.6: Block diagram RDC mechanism.

The current release provides monitoring for crossbar outputs 0 to 7. The weights for each signal are shared with the MCCU and are stored in registers 2.12 and 2.13. Weights are 8-bit fields.Counters have overflow protection, preventing the count from wrapping over the maximum value. The maximum value for each event (watermarks), are stored in registers 2.14 and 2.15.

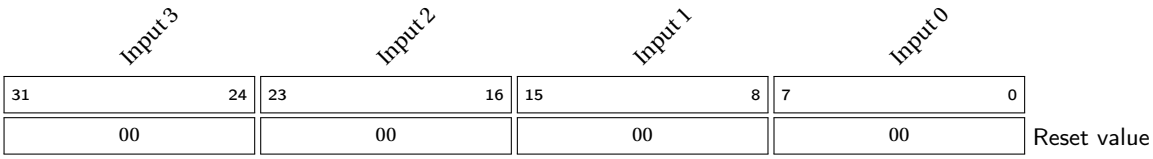
The RDC shares the main configuration register with the MCCU (register 2.8). Through this register, the unit can be reset and enabled through the corresponding fields. Such fields are active high signals.

The unit does provide access to the internal interrupt vector (register 2.11), but such information is redundant and may be removed in future releases. Given the current watermarks and assigned weights, the events responsible for the interrupt can be identified. The RDC interrupt has been routed to AHB interrupt 11.

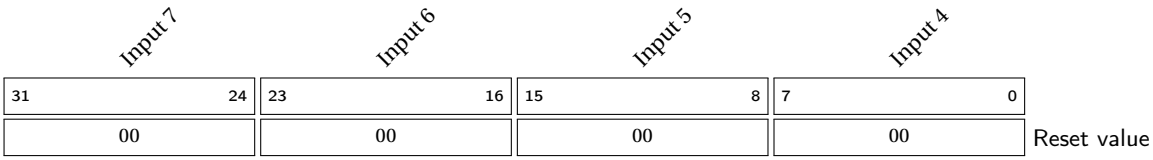
Register 2.11: RDC INTERRUPT VECTOR (0x0a0)

Reserved				Core 3	Core 2	Core 1	Core 0	
31	...	4		3	2	1	0	
x	...			00	00	00	00	Reset value

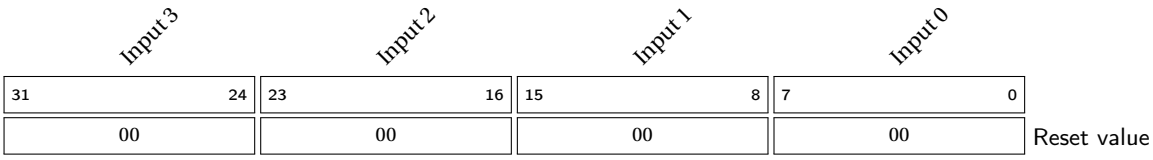
Register 2.12: RDC EVENT WEIGHTS REGISTER 0 (SHARED WITH MCCU) (0x098)



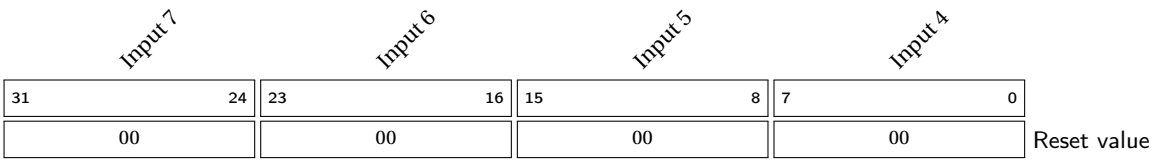
Register 2.13: RDC EVENT WEIGHTS REGISTER 1 (SHARED WITH MCCU) (0x09c)



Register 2.14: RDC WATERMARK REGISTER 0 (0x0a4)



Register 2.15: RDC WATERMARK REGISTER 1 (0x0a8)



2.10 SOFTWARE SUPPORT

The unit can be configured by the user at a low level following the description of previous sections and the documents associated for each unit. In addition we provide a small bare-metal driver under the path *grlib/software/noelv/BSC_tests/BSC_libraries/PMU*.

The driver is composed of three files.

- *pmu_vars.h* : Defines a set of constants with the RTL parameters that generated the unit. Such constants allow the reusing of functions among hardware configurations.
- *pmu_test.h*: Defines the memory position of the PMU within the memory map of the SoC. It also contains the prototypes of each function of the driver.
- *pmu_test.c*: Contains the definition of each of the functions.

Given the current development status, the driver is not autogenerated, and **modifications to the RTL may require manual changes on the previous files**.

3 CONFIGURATION OPTIONS

Table 3.1 shows the configuration parameters exposed by *ahb_wrapper.vhd*. Given the current development status, changes to the configuration options may require manual modifications to internal modules and software drivers. Future releases will expose parameters to enable individual PMU features and allow for more flexibility.

Table 3.1: Configuration options (VHDL ports)

Generic	Function	Allowed range	Default
HADDR	AHB base address	0 to 16#fff#	0
HMASK	AHB address mask	0 to 16#fff#	16#fff#
N_REGS	Total of accessible registers	2 to 64	43
PMU_COUNTERS	Number of generic counters. Same as crossbar outputs	1 to 32	24
N_SOC_EV	SoC signals. Inputs to the crossbar	1 to 64	32
REG_WIDTH	Size of registers and counters	32 **or** 64	32

4 SIGNAL DESCRIPTIONS

Table 4.1 shows the interface of the core (VHDL ports).

Table 4.1: Signal descriptions (VHDL ports)

Signal name	Field	Type	Function	Active
RST		Input	Reset	Low
CLK		Input	AHB master bus clock	-
PMU_EVENTS		Input	Input for regular SoC events	-
CCS_CONTENTION		Input	Input for contention cycle stack signals that measure contention	-
CCS_LATENCY		Input	Input for contention cycle stack signals that measure access latency	-
AHBSI	*	Input	AHB slave input signals	-
AHBSO	*	Output	AHB slave output signals, includes interrupts	-

5 LIBRARY DEPENDENCES

Table 5.1 shows the libraries used when instantiating the core (VHDL libraries).

Table 5.1: Library dependencies

Library	Package	Imported units	Description
IEEE	std_logic_1164	Types	Standard logic types
GRLIB	amba	Signals	AMBA signal definitions
GRLIB	config	Types	Amba P&P types
GRLIB	devices	Types	Device names and vendors
GRLIB	stdlib	All	Common VHDL functions
GAISLER	noelv	Signals	Counter vectors and types
BSC	pmu_module	Instances and signals	Instances and signal definitions for the PMU

6 Instantiation

An example design is provided in the context of SELENE and De-RISC. Integration examples of earlier releases of the unit along LEON3MP can be provided under demand.

Listing 1: SafePMU instance example for gpp_sys

```
-- Include BSC library
library bsc;
use bsc.pmu_module.all;

--Provide a non-overlaping hsidx
constant hsidx_pmu : integer := 6;

--Update the hsidx of the next slave
constant nextslv      : integer := hsidx_pmu + 1;

--Declare events  and signals of interest. They may change
--for each use case. Route such events to  pmu_events,
--ccs_contention and ccs_latency ports

--Instance of the unit

PMU_inst : ahb_wrapper
generic map(
ncpu   => CFG_NCPU,
hindex => hsidx_pmu,
haddr  => 16#801#,
hmask  => 16#FFF#
)
port map(
```

```
rst                 => rstn,
clk                 => clkm,
pmu_events          => pmu_events,
ccs_contention      => ccs_contention,
ccs_latency         => ccs_latency,
ahbsi               => ahbsi,
ahbso               => ahbso(hsidx_pmu));
```

Given the development status of the module and drivers it is recommended to not modify the internal parameters of the module.