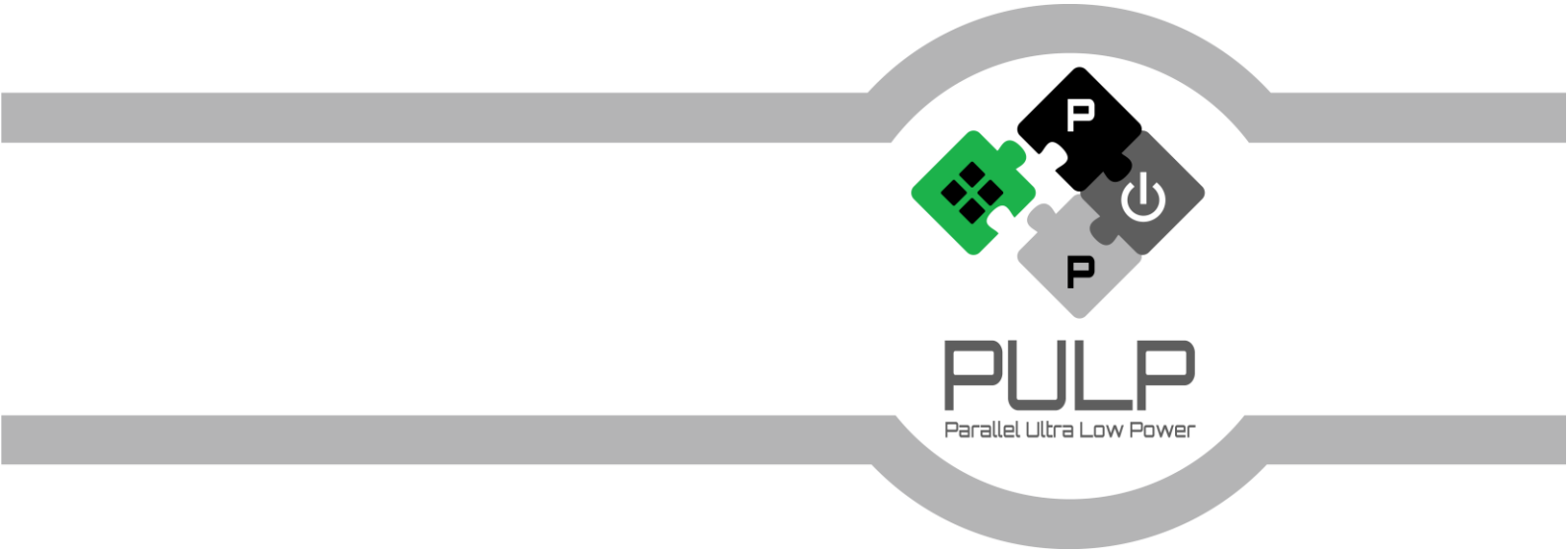


Event Unit Flex: User Manual

October 2017
Revision 4.1
Florian Glaser (glaser@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
4.0	24.05.16	Florian Glaser	First version using template, aligned documentation with current state of rtl development
4.1	06.10.17	Florian Glaser	Added HW mutex, updated event map

Table of Contents

1 Architectural description

The re-design of the event unit for the PULP ecosystem was done with a focus on the following points:

- Flexible and parametrable number of external events associated with SoC peripherals and other clusters; identification of peripherals through IDs over dedicated bus instead of fixed event lines.
- Fixed number of 32 master event lines per core with fixed event sources. One line to flag the presence of any event associated with SoC peripherals, one for inter-cluster communication.
- Flexible number of fast software events for inter-core communication; single-cycle event trigger and sleep
- Flexible number of fast hardware barriers for cluster synchronization; single-cycle barrier trigger and sleep.
- Advanced features for multi-cluster synchronization like mutexes, barriers, message dispatch mechanisms and the like. All listed features are currently in scratch status.
- RTL sourcecode simplification (rewrite from scratch).

The architecture is depicted in Figure 1Figure 1: Architecture and placement of the IP and described in the following subsections.

1.1 General considerations

The IP consists of several sub modules; each sub module contains both a set of self-contained functionality and (usually) private signals running to and from the outside of the IP. Communication within the modules is almost exclusively done by means of IP-internal events.

From the cores, SoC and off-chip areas, all configuration and also parts of the functionality (e.g., sw event triggering, barrier triggering) is accessible through the “Peripheral Crossbar Bus”, which is widely used in the PULP ecosystem. To meet different requirements, multiple (slave) ports of said bus are used:

- N_{Core} slave ports, directly attached to the peripheral demux of each core. Single-cycle access makes timing critical; thus only timing-critical configuration and functionality of the corresponding core is accessible (mostly event triggering, event buffer and mask operations, see memory map). The available range of 0x400 is aliased for each core.
- Two concatenated slave ports of the peripheral interconnect, each 0x400 in address range. Registering of the bus signals in the peripheral interconnect allows for relaxed timing and thus making a large set of registers accessible. While all configuration registers are accessible through this link, some of the functionality is not (peripheral demux link should be used). The external master port of the peripheral interconnect makes said register set accessible to bus masters located in the SoC, other clusters or off-chip (e.g., debug bridge).

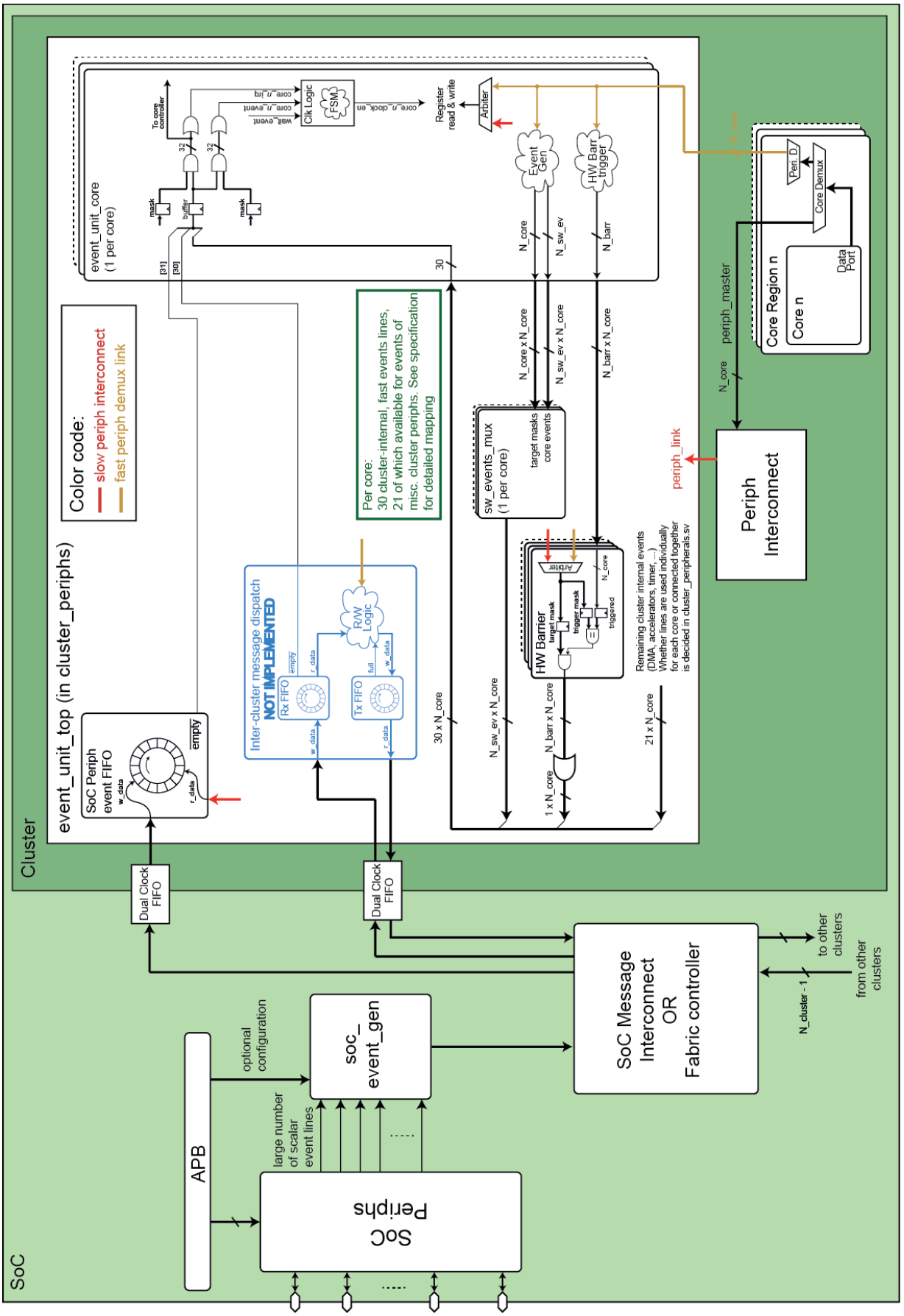


Figure 1: Architecture and placement of the IP

1.2 event_unit_core

Gets instantiated once per core and features:

- Private mask and buffer registers for each core. The event buffer stores all incoming events, regardless of which event lines are masked.
- Control logic for the clock gates of each core, located inside each core region. Takes care of disabling the core clock when waiting for events, initiated through read accesses to dedicated registers.
- Software event generator: Creates one of $N_{\mathrm{sw\_ev}}$ software events; fast event generation is ensured by accessing the sw event generator through the direct demux link.
- The address of writes to dedicated registers selects the ID of the sw event to be created, the data are used as a target core mask to possibly issue the sw event only to a subset of cores. Note: It is also possible for a core to send a sw event to itself.
- Trigger logic for each of the N_{Barr} hardware barriers. Only placed in this unit to simplify the rtl source code.

1.3 External events

External events are created by either SoC peripherals or other clusters. An event generates a transaction on separate message busses with an ID (currently, 8 bit) that identifies the source of the event. A message bus is essentially a mini-AXI channel that runs from the SoC into the cluster(s), meaning one multi-bit, uni-directional data signal and a `valid` and `ready` handshake pair. Data is considered transferred iff both `valid` and `ready` are high at the rising clock edge of the controlling clock.

Events from SoC peripherals are broadcasted to all cores of a cluster to simplify rtl source code. For inter-cluster communication, a solution has yet to be found.

1.3.1 soc_event_gen

Basically a one-hot to AXI converter, creates a message bus transaction as soon as a peripheral issues an event. Also included is a FIFO with the size roughly equal to the number of peripherals; sw must exclude cases where more events are generated at the same time (highly unlikely anyways).

1.3.2 SoC message interconnect

Takes care of dispatching events from SoC peripherals and clusters to the correct target cluster. Not implemented at the moment. Should be replaced by a fabric controller, depending on the level of required flexibility.

1.3.3 soc_periph_fifo

This unit is the data sink for the message bus associated with SoC events, running into each cluster. As soon as an event reaches this unit, it is stored into the next empty place of the 8-entry FIFO, as well as bit 31 of the event buffer of all cores set to 1. A core responsible for SoC events can then successively read the entries of the FIFO, a valid bit per entry is used to flag an empty FIFO. A full FIFO is flagged to the data producer by a 0-tied `ready` signal, which the data producer must respect and not send any further events.

1.3.4 Inter-cluster message dispatch

Needed once per cluster to send and receive messages to and from other clusters. Separate message busses and FIFOs for each direction, the data channels will probably be 32 bit wide. In addition, an address signal is needed to identify the target cluster. The SoC message interconnect must take care of connecting the correct `ready` and `valid` signals with each other upon a transaction. If a transmitter FIFO is full, the sending core must not be stalled but instead check a transmit state register to detect the full FIFO and try again. The rationale behind this measure is to avoid deadlocks where two clusters wait for messages from each other and are stalled on the hardware level where sw has no chance of resolving the conflict.

1.4 Internal events

All events generated inside the cluster are considered internal events, of which there are 30. They are guaranteed to be contention-free to enable fast and reliable core-to-core communication. While software and barrier events are generated inside the `event_unit_flex`, the remaining 21 internal event lines can be connected to various modules inside the cluster, depending on the requirements of the chip at hand. Each of the 21 lines is available independently for each core. It can be decided on the cluster level, whether a specific event line is common to all cores or kept independently.

1.4.1 sw_events_mux

Purely combinational block without any configuration, needed once per core. All N_{Core} instances together mux the total N_{Core}^2 mask bits and the $N_{\mathrm{sw\_ev}} \times N_{\mathrm{Core}}$ sw event lines to $N_{\mathrm{sw\_ev}} \times N_{\mathrm{Core}}$ lines that get then split up by core association. This is needed since each core has the possibility to send each sw event to all cores (even to itself).

There are N_{Core} possible sources for a specific sw event line of a specific core, which get ORed in this unit. The target core masks, which are specific to each event issuing core, are applied before the OR operation. To avoid a situation where the event source is unclear for a core receiving a sw event, it must be made sure by user software that at no time the same sw event is sent to a core from more than one core at a time.

1.4.2 HW barrier units

Up to 16 hardware barrier units can be instantiated, a small interconnect that routes all demux links from the cores and the link from the peripheral crossbar correctly is automatically adapted.

Each unit features a status register that stores one bit per core, reflecting whether a specific core has already reached/triggered the barrier or not. As soon as all cores activated in a programmable trigger mask have reached the barrier, an event is sent to all cores that are activated in a programmable target mask while simultaneously clearing the status register to all zeroes.

Since each core can only be waiting for a single barrier at a time, all event lines from each barrier unit, associated with a specific core, are ORed (a core receives the barrier event if any of the barrier units triggers an event to said core).

1.4.3 HW mutex

The hardware mutex implements a hardware-supported lock/unlock mechanism. Locking is done with a read access to a specific register, then the blocking read requests are leveraged: If the mutex is unlocked, it gets locked and the core can continue execution. If the mutex is already locked, the core gets blocked on the read and put to sleep.

Unlocking is done through a write to the same register; one of the cores waiting to lock the mutex (if any) is then woken up by means of asserting the mutex event for this core. In case of multiple waiting cores, the one with the lowest core id gets selected for locking. The hardware does not check on illegal unlocks, i.e., writes from cores that do not own the mutex. This can for example happen if a malfunctioning runtime allows other events than the mutex event to wake up a core during a mutex wait state, which the core then interprets as owning the mutex. However, there are (non-synthesizable) simulation checkers for these erroneous accesses and similar ones inside the RTL, which print on the simulation shell in such cases.

In addition, an unlocking core can send a message to the core that will lock the mutex next by posting the message as write data of the unlocking write access; the message is stored inside the HW mutex and transmitted to the next woken up core as read data.

The hardware is designed to have multiple instances of the mutex, however so far only one instance is instantiated and supported by the runtimes.

1.4.4 Remaining cluster events

The remaining internal event lines are available on the higher hierarchies (cluster peripherals, cluster top level) for each core independently. Some lines are reserved for common components like DMAs and timers and shall not be used for other, chip-specific modules. See Table 1 for a detailed mapping.

Event line #	Event source	Common/individual
31	Inter-cluster message receive event	Common to all cores
30:28	reserved	
27	SoC peripheral events from message bus	Common to all cores
26:19	Miscellaneous cluster events	Chip-specific
18	HW dispatch	Individual per core
17	HW mutex	Individual per core
16	HW barriers	Individual per core
15:12	Accelerators	Chip-specific
11:10	Timers	Chip-specific
9:8	DMA events	Individual per core
7:0	Software events	Individual per core

Table 1: Association of individual event lines

2 Going to sleep and waking up

Instead of flushing the pipeline of a core and, after flushing is completed, shut off its clock, the going-to-sleep process is solved in a more efficient way:

1. When a core is idle and wants to go to sleep, it performs a read to a register out of a set of special registers, contained in the demuxed and aliased address space of its corresponding `event_unit_core` unit. However, the unit will not give the grant to this read and consequently stall the issuing core, beginning from the WB stage. For advanced functionality and energy efficiency, the read is performed by using a special read instruction, `p.elw`. The abstraction and helper functions provided in the SDK take care of using the correct read instruction.

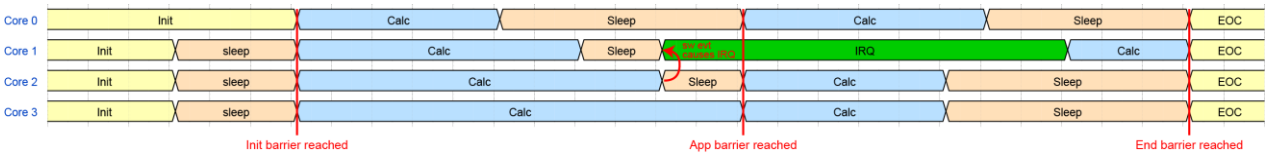
2. With the grant to the read still not given, the `event_unit_core` will switch off the clock of the core. This can safely be done, since the register read is a `p.elw` instruction, the request for which only gets sent out of the core in the WB stage, where all prior instructions have already been fully executed. The process of going to sleep is hereby completed.

3. Once either an active event line or an active interrupt source gets asserted, the `event_unit_core` unit activates the clock of the core again and gives the grant for the register read. The data of the register read are the contents of the event buffer, thus saving cycles that would instead be needed to read the reason for the core being woken up. There are options for a simultaneous, automatic buffer clear on waking up as well as for simultaneous sw event or barrier triggering before going to sleep, thereby saving additional cycles.

2.1 Interrupts during wait states

A prominent use case includes being sensitive to one or more interrupt sources while waiting for events to happen. Dedicated hardware logic is employed both in the core and the `event_unit_flex` to support such scenarios, ensuring correct execution of interrupt handlers during wait states and returning to the wait state after execution is completed.

If one of the events in question has occurred during the execution of the interrupt handler, the handler is followed directly by regular program execution without going back to sleep. See the figure below for an illustration of an example use case in a four-core system, employing barriers and an interrupt associated with a sw event.



Note:

It is possible to trigger some corner-cases when being sensitive to interrupts during wait states. What follows is a list of corner-cases that have already been found/investigated and whether you should avoid them by software or not. If you found a new corner-case, please email a description to glaserf@ethz.ch .

1. ***NOT supported: Being sensitive to an interrupt source and waiting for the event associated with the same source (i.e., master event line) at the same time.***
   The problem arises from fact that the interrupt handler must clear the bit in the buffer by which it has been triggered to avoid its own re-execution. As a consequence, the wake-up supposed to happen after will never occur. Workarounds will either cause considerable hardware overhead (i.e., an additional buffer per core) or introduce the danger of losing events that arrive during the execution of the interrupt handler. If your application requires this scenario, use a (possibly different) software event for waking up one or more cores after handling the interrupt.

3 Register description

3.1 Address map

Parts of the `event_unit_flex` can be accessed both through the peripheral interconnect and the peripheral demux link of each core; on a simultaneous write access to such a register through both ports the access from the peripheral demux port gets prioritized. Almost all registers can be accessed through the peripheral interconnect, with the exception of those that put the issuing core to sleep. This principle is illustrated in Figure 1.

Accesses to the registers of the `event_unit_core` through the address space of the peripheral demux link are aliased to the registers belonging to the issuing core. All N_{Core} register sets are replicated in the address space of the peripheral interconnect, triggering software events or hardware barriers is handled as if an additional core would trigger them. The register descriptions make frequent use of the address placeholders expanded in Table 2 and the unit ranges listed in Table 3.

Placeholder	Expansion
eu_base_addr	0x1020_0800
eu_core_base_addr	eu_base_addr
hw_barr_base_addr	eu_base_addr + 0x400
eu_sw_events_base_	eu_base_addr + 0x600
eu_soc_events_base	eu_base_addr + 0x700
eu_cluster_message_base_addr	eu_base_addr + 0x780
Base for next cluster peripheral	eu_base_addr + 0x800
eu_demux_base_addr	0x1020_4000
eu_hw_dispatch_demux_base_addr	eu_demux_base_addr + 0x80
eu_hw_mutex_demux_base_addr	eu_demux_base_addr + 0xC0
eu_sw_events_demux_base_addr	eu_demux_base_addr + 0x100
hw_barr_demux_base_addr	eu_demux_base_addr + 0x200
Base for next demuxed peripheral	eu_demux_base_addr + 0x400

Table 2: Address placeholders

Unit range	Value
core_id	$0 \ldots N_{\mathrm{Core}} - 1$
sw_event_id	$0 \ldots N_{\mathrm{sw\_ev}} - 1$
int_event_id	$0 \ldots 29$
hw_mutex_id	$0 \ldots N_{\mathrm{mut}} - 1$
hw_barr_id	$0 \ldots N_{\mathrm{barr}} - 1$

Table 3: Unit ranges

The address map is designed such that each N_{Core}, N_{barr} and N_{mut} can be at most 16, $N_{\mathrm{sw\_dev}}$ at most 8.

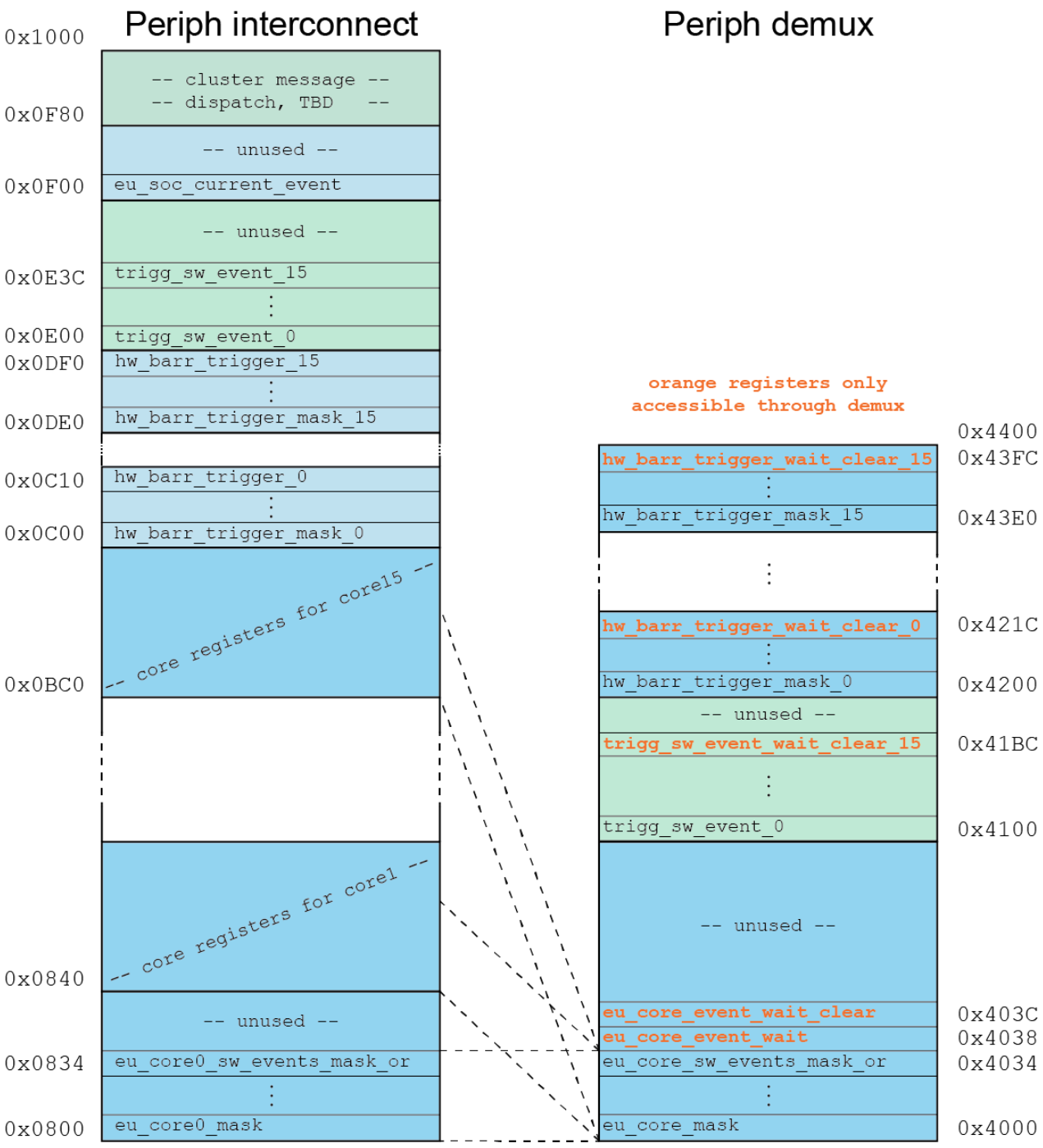


Figure 2: Address space of the event unit. Not to scale. All addresses are offsets to `0x1020_0000`.

For describing whether a register or specific bits of it can be read or written, the syntax listed in Table 4 is used.

R	Register/bit can only be read
R/W	Register/bit can be written and read
Rn	Register/bit always reads as n
X/X-n	Register/bit has reset value n

Table 4: Register access syntax

3.2 event_unit_core

Name	`eu_core_mask`
Periph address	`eu_core_base_addr + 0x40*core_id + 0x00`
Demux address	`eu_demux_base_addr + 0x00`

Bit #	R/W	Description
31	R/W-0	Mask bit for SoC peripheral events. Setting this bit to 1 will wake up the corresponding core as soon as a transaction on the SoC peripheral message bus takes place.
30	R/W-0	Mask bit for inter-cluster message events. Setting this bit to 1 will wake up the corresponding core as soon as a transaction on the inter-cluster message bus takes place.
29:0	R/W-0	Mask bits for cluster internal events. Setting a bit to 1 will wake up the corresponding core as soon as the associated event source issues an event.

Name	`eu_core_mask_and`
Periph address	`eu_core_base_addr + 0x40*core_id + 0x04`
Demux address	`eu_demux_base_addr + 0x04`

Bit #	R/W	Description
31:0	R0/W	Writing to this register allows clearing one more bits of the event mask. On a write access, the following operation is performed: `eu_core_mask = eu_core_mask & ~eu_core_mask_and.`

Name	eu_core_mask_or
Periph address	eu_core_base_addr + 0x40*core_id + 0x08
Demux address	eu_demux_base_addr + 0x08

Bit #	R/W	Description
31:0	R0/W	Writing to this register allows setting one more bits of the event mask. On a write access, the following operation is performed: eu_core_mask = eu_core_mask | eu_core_mask_or.

Name	eu_core_mask_irq
Periph address	eu_core_base_addr + 0x40*core_id + 0x0C
Demux address	eu_demux_base_addr + 0x0C

Bit #	R/W	Description
31	R/W-0	Mask bit for SoC peripheral interrupts. Setting this bit to 1 will cause the calling of an interrupt handler in the corresponding core as soon as a transaction on the SoC peripheral message bus takes place.
30	R/W-0	Mask bit for inter-cluster message events. Setting this bit to 1 will cause the calling of an interrupt handler in the corresponding core as soon as a transaction on the inter-cluster message bus takes place.
29:0	R/W-0	Mask bits for cluster internal events. Setting a bit to 1 will cause the calling of an interrupt handler in the corresponding core as soon as the associated event source issues an event.

Name	eu_core_mask_irq_and
Periph address	eu_core_base_addr + 0x40*core_id + 0x10
Demux address	eu_demux_base_addr + 0x10

Bit #	R/W	Description
31:0	R0/W	Writing to this register allows clearing one more bits of the interrupt mask. On a write access, the following operation is performed: eu_core_mask_irq = eu_core_mask_irq & ~eu_core_mask_irq_and.

Name	eu_core_mask_irq_or
Periph address	eu_core_base_addr + 0x40*core_id + 0x14
Demux address	eu_demux_base_addr + 0x14

Bit #	R/W	Description
31:0	R0/W	Writing to this register allows setting one more bits of the interrupt mask. On a write access, the following operation is performed: eu_core_mask_irq = eu_core_mask_irq | eu_core_mask_irq_or.

Name	eu_core_status
Periph address	eu_core_base_addr + 0x40*core_id + 0x18
Demux address	eu_demux_base_addr + 0x18

Bit #	R/W	Description
31:1	R0	Unused.
0	R	Status of the core clock. 0: The clock of the core is stopped. 1: The clock of the core is running.

Name	eu_core_buffer
Periph address	eu_core_base_addr + 0x40*core_id + 0x1C
Demux address	eu_demux_base_addr + 0x1C

Bit #	R/W	Description
31	R-0	Status bit of the SoC peripheral event source. Reading this bit as 1 means that *at least* one SoC peripheral event has occurred since this bit was cleared.
30	R-0	Status bit of the inter-cluster message event source. Reading this bit as 1 means that *at least* one inter-cluster message event has occurred since this bit was cleared.
29:0	R-0	Status bit of the cluster internal event sources. Reading a bit as 1 means that the corresponding event source has issued *at least* one event since the bit was cleared.

Name	eu_core_buffer_masked
Periph address	eu_core_base_addr + 0x40*core_id + 0x20
Demux address	eu_demux_base_addr + 0x20

Bit #	R/W	Description
31:0	R-0	Holds the same content as eu_core_buffer but with eu_core_mask applied: eu_core_buffer_masked = eu_core_buffer & eu_core_mask.

Name	eu_core_buffer_irq_masked
Periph address	eu_core_base_addr + 0x40*core_id + 0x24
Demux address	eu_demux_base_addr + 0x24

Bit #	R/W	Description
31:0	R-0	Holds the same content as eu_core_buffer but with eu_core_mask_irq applied: eu_core_buffer_irq_masked = eu_core_buffer & eu_core_mask_irq.

Name	eu_core_buffer_clear
Periph address	eu_core_base_addr + 0x40*core_id + 0x28
Demux address	eu_demux_base_addr + 0x28

Bit #	R/W	Description
31:0	R0/W	Clears the corresponding buffered bit in eu_core_buffer to 0. If the corresponding event line is active in the same cycle where the clear operation gets effective, the bit **still does get cleared** to 0.

Name	eu_core_sw_events_mask
Periph address	eu_core_base_addr + 0x40*core_id + 0x2C
Demux address	eu_demux_base_addr + 0x2C

Bit #	R/W	Description
31:16	R0	Unused.
15:0	R/W-0	Target core mask that is used when a software event gets triggered by a read of one of the registers eu_core_trigg_sw_event_wait or eu_core_trigg_sw_event_wait_clear. The target core mask is shared for all software events of a core when they are triggered through the aforementioned registers. For no-wait software event triggering, the target core mask is set through the write data of a write access to eu_core_trigg_sw_event, available separately for each software event of a core.

Name	eu_core_sw_events_mask_and
Periph address	eu_core_base_addr + 0x40*core_id + 0x30
Demux address	eu_demux_base_addr + 0x30

Bit #	R/W	Description
31:0	R0/W	Writing to this register allows clearing one more bits of eu_core_sw_events_mask. On a write access, the following operation is performed: eu_core_sw_events_mask = eu_core_sw_events_mask & ~eu_core_sw_events_mask_and.

Name	eu_core_sw_events_mask_or
Periph address	eu_core_base_addr + 0x40*core_id + 0x34
Demux address	eu_demux_base_addr + 0x34

Bit #	R/W	Description
31:0	R0/W	Writing to this register allows setting one more bits of eu_core_sw_events_mask. On a write access, the following operation is performed: eu_core_sw_events_mask = eu_core_sw_events_mask | eu_core_sw_events_mask_or.

Name	eu_core_event_wait
Demux address	eu_demux_base_addr + 0x38

Bit #	R/W	Description
31:0	R-0	Reading this register will stop the clock of the core until at least one event with the corresponding mask bit set to 1 occurs. The read content of this register is identical to that of eu_core_buffer_masked.

Name	eu_core_event_wait_clear
Demux address	eu_demux_base_addr + 0x3C

Bit #	R/W	Description
31:0	R-0	Reading from this register has the same effect as reading from eu_core_event_wait. In addition, the bits of eu_core_buffer that are set to 1 in eu_core_mask will be cleared to 0 after the read.

Name	eu_core_hw_mutex
Demux address	eu_hw_mutex_demux_base_addr + 0x04*hw_mutex_id

Bit #	R/W	Description
31:0	R0/W	Reading tries to lock the corresponding mutex, writing unlocks it. The value written during an unlock gets transmitted to the core which locks the mutex next. There is no protection against illegal unlocks in hardware, i.e., the runtime must make sure that only the core who locked a mutex performs a write on this register.

Name	eu_core_trigg_sw_event
Periph address	eu_sw_events_base_addr + 0x04*sw_event_id
Demux address	eu_sw_events_demux_base_addr + 0x04*sw_event_id

Bit #	R/W	Description
31:16	R0	Unused.
15:0	R0/W	Writing to this register will trigger software event sw_event_id for all cores whose corresponding bit is written as 1. If less than 16 cores are present, the $16 - N_{\mathrm{Core}}$ MSBs are ignored.

Name	eu_core_trigg_sw_event_wait
Demux address	eu_sw_events_demux_base_addr + 0x40 + 0x04*sw_event_id

Bit #	R/W	Description
31:0	R-0	This register combines both the functionality of eu_core_event_wait and eu_core_trigg_sw_event. Since at the moment no data can be sent with a p.elw (or similar) instruction, the target core mask needs to be written first through a write to eu_core_sw_events_mask. When reading from this register, the issuing core is put to sleep and software event sw_event_id triggered for all cores that are activated in eu_core_sw_events_mask. While the target core mask needs to be separately set during boot/sw initialization, a software event can be issued and the issuing core put to sleep in one cycle during regular program execution. After the issuing core has been woken up again, the read contents are identical to those of eu_core_buffer_masked.

Name	eu_core_trigg_sw_event_wait_clear
Demux address	eu_sw_events_demux_base_addr + 0x80 + 0x04*sw_event_id

Bit #	R/W	Description
31:0	R-0	Same functionality as eu_core_trigg_sw_event_wait, with the exception that all bits of eu_core_buffer that are set to 1 in eu_core_mask are cleared to 0 after the read.

3.3 event_unit_soc_events

Name	eu_core_current_event
Periph address	eu_soc_events_base_addr

Bit #	R/W	Description
31	R-0	Valid bit. If read as 1, the event id contained in the bits [7:0] is valid.
30:8	R0	Unused.
7:0	R-0	Oldest SoC peripheral event ID that was written into the FIFO. After a read of this register, the next event ID (if any) will be read at the next access.

3.4 HW barriers

Name	hw_barr_trigger_mask
Periph address	hw_barr_base_addr + 0x20*barr_id
Demux address	hw_barr_demux_base_addr + 0x20*barr_id

Bit #	R/W	Description
31:16	R0	Unused.
15:0	R/W-0	Trigger mask for HW barrier unit barr_id. Every bit set to 1 needs to be matched in hw_barr_status in order for HW barrier event barr_id to be triggered. Writing this register with all 0s corresponds to disabling HW barrier barr_id.

Name	hw_barr_status
Periph address	hw_barr_base_addr + 0x20*barr_id + 0x04
Demux address	hw_barr_demux_base_addr + 0x20*barr_id + 0x04

Bit #	R/W	Description
31:16	R0	Unused.
15:0	R-0	Current status of HW barrier unit barr_id. A 1 means that the corresponding bit has already been triggered. Gets cleared to all 0s once the trigger mask set in hw_barr_trigger_mask has been matched.

Name	hw_barr_status_summary
Periph address	hw_barr_base_addr + 0x20*barr_id + 0x08
Demux address	hw_barr_demux_base_addr + 0x20*barr_id + 0x08

Bit #	R/W	Description
31:16	R0	Unused.

15:0	R-0	Each bit represents a summary of the barrier status for the corresponding core.
		All N_{barr} implemented HW barrier units are included in the summary:
		`hw_barr_status_summary =`
		`  hw_barr_status_0 | hw_barr_status_1 | ... | hw_barr_status_(N_barr-1)`

Name	hw_barr_target_mask
Periph address	hw_barr_base_addr + 0x20*barr_id + 0x0C
Demux address	hw_barr_demux_base_addr + 0x20*barr_id + 0x0C

Bit #	R/W	Description
31:16	R0	Unused.
15:0	R/W-0	Target core mask for HW barrier unit barr_id. Once every bit set to 1 in hw_barr_trigger_mask is matched in hw_barr_status, the cluster event line associated with HW barrier barr_id will be asserted for all cores whose corresponding bit is set to 1.

Name	hw_barr_trigger
Periph address	hw_barr_base_addr + 0x20*barr_id + 0x10
Demux address	hw_barr_demux_base_addr + 0x20*barr_id + 0x10

Bit #	R/W	Description
31:16	R0	Unused.
15:0	R0/W	Every bit written as 1 will trigger the corresponding bit of HW barrier barr_id and set the corresponding bit in hw_barr_status.

Name	hw_barr_trigger_self
Demux address	hw_barr_demux_base_addr + 0x20*barr_id + 0x14

Bit #	R/W	Description
31:0	R0	When read by core core_id, bit [core_id] for HW barrier barr_id is set.

Name	hw_barr_trigger_wait
Demux address	hw_barr_demux_base_addr + 0x20*barr_id + 0x18

Bit #	R/W	Description
31:16	R0	Unused.
15:0	R-0	When read by core core_id, bit [core_id] for HW barrier barr_id is set and the issuing core put to sleep. After waking up again, the read data are identical to the contents of eu_core_buffer_masked of the issuing core.

Name	hw_barr_trigger_wait_clear
Demux address	hw_barr_demux_base_addr + 0x20*barr_id + 0x1C

Bit #	R/W	Description
31:16	R0	Unused.
15:0	R-0	Same functionality as hw_barr_trigger_wait. In addition, all bits set to 1 in eu_core_mask are cleared to 0 in eu_core_buffer after its contents have been read, following a sleep period.