Hardware Processing Engines

Release 2.0

Francesco Conti

Mar 15, 2024

CONTENTS:

Hardware Processing Engines (HWPEs) are special-purpose, memory-coupled accelerators that can be inserted in the SoC or cluster of a PULP system to amplify its performance and energy efficiency in particular tasks.

Differently from most accelerators in literature, HWPEs do not rely on an external DMA to feed them with input and to extract output, and they are not (necessarily) tied to a single core. Rather, they operate directly on the same memory that is shared by other elements in the PULP system (e.g. the L1 TCDM in a PULP cluster, or the shared L2 in PULPissimo). Their control is memory-mapped and accessed through a peripheral bus or interconnect. HW-based execution on an HWPE can be readily intermixed with software code, because all that needs to be exchanged between the two is a set of pointers and, if necessary, a few parameters.

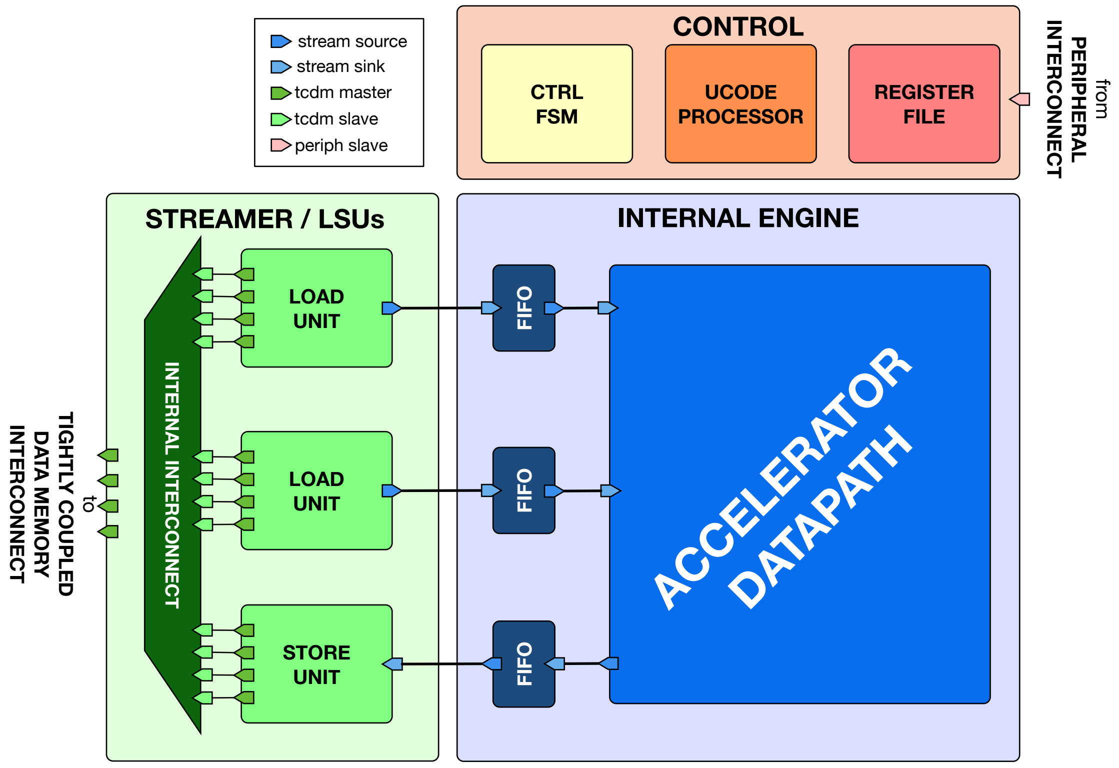


Fig. 1: Template of a Hardware Processing Engine (HWPE).

This document defines the interface protocols and modules that are used to enable connecting HWPEs in a PULP system. Typically, such a module is divided in a **streamer** interface towards the memory system, a **control/peripheral** interface used for programming it, and an **engine** containing the actual datapath of the accelerator.

CHAPTER

ONE

HWPE INTERFACE PROTOCOLS

1.1 HWPE-Stream protocol

The HWPE-Stream protocol is a simple protocol designed to move data between the various sub-components of an HWPE. As HWPEs are memory-based accelerators, streams are typically generated and consumed internally within the accelerator between fully synchronous devices. HWPE-Stream can cross between two clock domains using dual-clock FIFOs; handshakes still have to happen in a fully synchronous way. HWPE-Stream streams are directional, flowing from a *source* to a *sink* direction, using a two signal *handshake* and carrying a data *payload*. Fig. 1.1 and Table 1.1 report the signals used by the HWPE-Stream protocol.

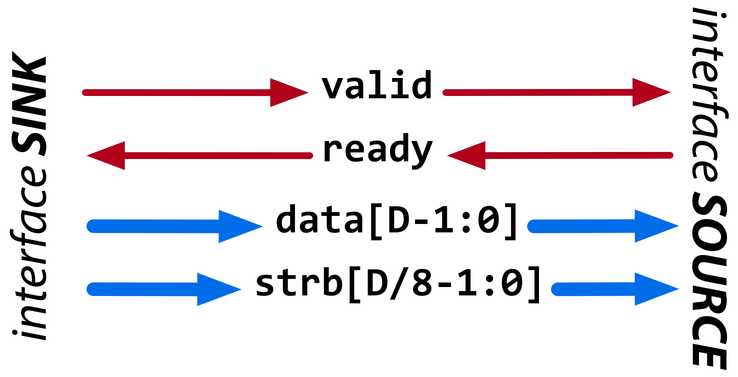


Fig. 1.1: Data flow of the HWPE-Stream protocol. Red signals carry the *handshake*, blue ones the *payload*.

Table 1.1: HWPE-Stream signals.

Signal	Size	Description	Direction
data	Multiple of 8 bits	The data payload transported by the stream.	from *source* to *sink*
strb	size(*data*)/8	Optional. Indicates valid bytes in the data payload (1=valid).	from *source* to *sink*
valid	1 bit	Handshake valid signal (1=asserted).	from *source* to *sink*
ready	1 bit	Handshake ready signal (1=asserted).	from *sink* to *source*

The handshake signals *valid* and *ready* are used to validate transactions between sources and sinks. Transactions are subject to the following rules:

1. **A handshake occurs in the cycle when both** *valid* **and** *ready* **are asserted**. The handshake is the "atomic" event after which the current payload is considered consumed by the consumer at the sink side of the HWPE-Stream interface.
2. *data* **and** *strb* **can change their value either a) when** *valid* **is deasserted, or b) in the cycle following a handshake, even if** *valid* **remains asserted**. In other words, valid data payloads must stay on the interface until a

valid handshake has occurred.

3. **The assertion of** *valid* **(transition 0 to 1) cannot depend combinationally on the state of** *ready*. On the other hand, the assertion of *ready* (transition 0 to 1) can depend combinationally on the state of *valid*. This rule, which is modeled around the similar behavior used by TCDM memories (see below) is meant to avoid any deadlock in ping-pong logic.
4. **The deassertion of** *valid* **(transition 1 to 0) can happen only in the cycle after a valid handshake**. In other words, valid data produced by a source must be correctly consumed before *valid* is deasserted.

`wavedrom_hwpe_stream` shows several correct handshakes on a HWPE-Stream, while `wavedrom_hwpe_stream_r2_no` and `wavedrom_hwpe_stream_r4_no` show two examples of incorrect transactions. Both behaviors are checked by means of asserts in the reference SystemVerilog code for HWPE-Stream interfaces. Rule 3 cannot be checked by means of asserts; it is up to the designer to avoid *valid* to *ready* combinational dependencies that could result in combinational loops, since the value of *ready* is assumed to be combinationally dependent from *valid*.

The only side channel that can be included in an HWPE-Stream is *strb*, which is optionally used to signal which bytes of the *data* payload contain meaningful data. HWPE-Stream streams in which *strb* is absent are assumed to have only valid bytes in their *data* payload. We refer HWPE-Stream streams with *strb* as *strobed streams*.

1.2 HWPE-Mem and HCI-Core protocols

1.2.1 HWPE-Mem

HWPEs are connected to external L1/L2 shared-memory by means of a simple memory protocol, using a request/grant handshake. The protocol used is called HWPE Memory (*HWPE-Mem*) protocol, and it is essentially similar to the protocol used by cores and DMAs operating on memories in standard PULP clusters. This document focuses on the specific signal names used within HWPEs and in the reference implementation of HWPE-Stream IPs. It supports neither multiple outstanding transactions nor bursts, as HWPEs using this protocol are assumed to be closely coupled to memories. It uses a two signal *handshake* and carries two phases, a *request* and a *response*.

The HWPE-Mem protocol is used to connect a *master* to a *slave*. Fig. 1.2 and Table 1.2 report the signals used by the HWPE-Mem protocol.

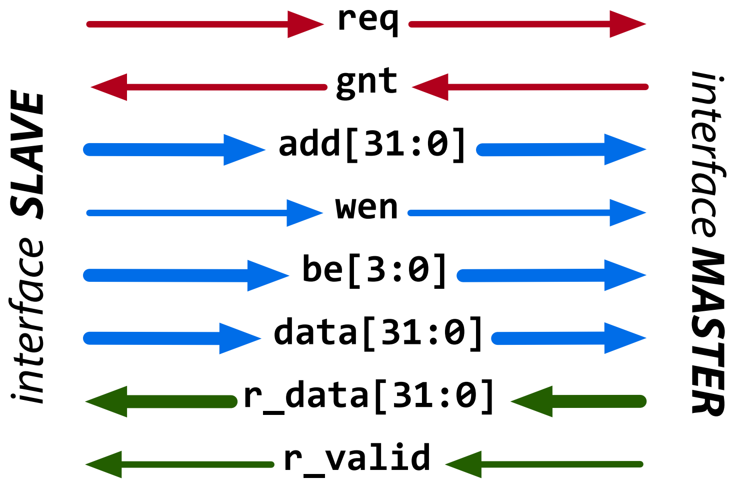


Fig. 1.2: Data flow of the HWPE-Mem protocol. Red signals carry the *handshake*; blue signals the *request* phase; green signals the *response* phase.

Table 1.2: HWPE-Mem signals.

Signal	Size	Description	Direction
req	1 bit	Handshake request signal (1=asserted).	*master* to *slave*
gnt	1 bit	Handshake grant signal (1=asserted).	*slave* to *master*
add	32 bit	Word-aligned memory address.	*master* to *slave*
wen	1 bit	Write enable signal (1=read, 0=write).	*master* to *slave*
be	4 bit	Byte enable signal (1=valid byte).	*master* to *slave*
data	32 bit	Data word to be stored.	*master* to *slave*
r_data	32 bit	Loaded data word.	*slave* to *master*
r_valid	1 bit	Valid loaded data word (1=asserted).	*slave* to *master*

The handshake signals *req* and *gnt* are used to validate transactions between masters and slaves. Transactions are subject to the following rules:

1. **A valid handshake occurs in the cycle when both** *req* **and** *gnt* **are asserted**. This is true for both write and read transactions.
2. *r_valid* **must be asserted the cycle after a valid read handshake;** *r_data* **must be valid on this cycle**. This is due to the tightly-coupled nature of memories; if the memory cannot respond in one cycle, it must delay granting the transaction.
3. **The assertion of** *req* **(transition 0 to 1) cannot depend combinationally on the state of** *gnt*. On the other hand, the assertion of *gnt* (transition 0 to 1) can depend combinationally on the state of *req* (and typically it does). This rule avoids deadlocks in ping-pong logic.

The semantics of the *r_valid* signal are not well defined with respect to the usual TCDM protocol. In PULP clusters, *r_valid* will be asserted also after write transactions, not only in reads. However, the HWPE-Mem protocol and the IPs in this repository should not make assumptions on the *r_valid* in write transactions.

1.2.2 HWPE-MemDecoupled

The HWPE-Mem protocol can be used to directly connect an accelerator to the shared memory of a PULP-based system. However, transactions using this protocol are inherently latency sensitive. HWPE-Mem rule 2 embodies this: an operation is complete only when its response has arrived. This means that HWPE-Mem streams, including load and store transactions, cannot be enqueued in a FIFO queue. To overcome this limitation, a variant of the HWPE-Mem protocol is HWPE-MemDecoupled. This protocol uses the same interface as HWPE-Mem but lifts rule 2 and adds a new rule 4. Transactions are thus following the following rules:

1. **A valid handshake occurs in the cycle when both** *req* **and** *gnt* **are asserted**. This is true for both write and read transactions.

3. **The assertion of** *req* **(transition 0 to 1) cannot depend combinationally on the state of** *gnt*. On the other hand, the assertion of *gnt* (transition 0 to 1) can depend combinationally on the state of *req* (and typically it does). This rule avoids deadlocks in ping-pong logic.
4. **The stream of transactions includes only reads (** *wen* **=1) or only writes (** *wen* **=0)**. Mixing reads and writes in the stream is not allowed.

HWPE-MemDecoupled transactions are insensitive to latency and their *request* and *response* phases can be treated similarly to separate HWPE-Stream streams. Once two or more HWPE-MemDecoupled transactions are mixed, the mixed interface has to be treated as a HWPE-Mem protocol (i.e. it is sensitive to latency).

1.2.3 HCI-Core

HCI-Core (Heterogeneous Cluster Interconnect – Core) is a protocol designed as a lighteweight extension of HWPE-Mem better suited for the needs of accelerators, and specifically of cluster-coupled HWPEs. This document focuses on the specific signal names used within HWPEs and in the reference implementation of HCI IPs. HCI-Core does not support bursts, but it supports in-order multiple outstanding transactions in a similar fashion to HWPE-MemDecoupled. Differently from HWPE-Mem, HCI-Core uses a two signal *handshake* but also includes an *lrdy* signal to support load backpressure on the response phase. HCI-Core carries two phases, a *request* and a *response*. HCI-Core signals have parametric width; Table 1.3 reports the parameters used by the HCI IPs; while Table 1.4 reports the signals used by the HCI-Core protocol.

Table 1.3: HCI-Core parameters.

Parameter	Description	Default	Range
DW	Data width in bits	32	mult. of *BW*, *WW*
AW	Address width in bits	32	1-32
BW	Width of an individually strobed “byte” in bits	8	1-32
WW	Width of a memory bank (“word”) in bits	32	mult. of *BW*
OW	Intra-bank offset width	32	1-32
UW	User-defined width	0	0-any

Table 1.4: HCI-Core signals.

Signal	Size	Phase	Description	Direction
req	1 bit	Request HS	Request valid (1=asserted).	*master* to *slave*
gnt	1 bit	Request HS	Request granted (1=asserted).	*slave* to *master*
r_valid	1 bit	Response HS	Response valid (1=asserted). Mandatory for load, optional for stores.	*slave* to *master*
lrdy	1 bit	Response HS	Response load ready (1=asserted).	*master* to *slave*
add	*AW* bit	Request	Word-aligned memory address.	*master* to *slave*
wen	1 bit	Request	Write enable signal (1=read, 0=write).	*master* to *slave*
be	*DW/BW* bit	Request	Byte enable signal (1=valid byte).	*master* to *slave*
boffs	*DW/WW* x *OW* bit	Request	Intra-bank offset.	*master* to *slave*
data	*DW* bit	Request	Data word to be stored.	*master* to *slave*
user	*UW* bit	Request	User-defined.	*master* to *slave*
r_data	32 bit	Response	Loaded data word.	*slave* to *master*
r_opc	1 bit	Response	Error code response.	*slave* to *master*
r_user	*UW* bit	Request	User-defined.	*slave* to *master*

The two phases of HCI-Core transactions can be treated as two separate channels, so HCI-Core transactions can be latency insensitive and support multiple in-order outstanding transactions (i.e., pipeline transactions). Request and response phases are organized to be treated like HWPE-Stream streams. Table 1.5 and Table 1.6 detail the rules that have to be followed for a valid transaction.

Table 1.5: HCI-Core Request phase rules.

Rule	Description
RQ-1 *HAND-SHAKE*	A valid handshake occurs in the cycle when both *req* and *gnt* are asserted, for both write and read transactions. All request phase signals are sampled on handshake cycles.
RQ-2 *NODEAD-LOCK*	The assertion of *req* (transition 0 to 1) cannot depend combinationally on the state of *gnt*. On the other hand, the assertion of *gnt* (transition 0 to 1) can depend combinationally on the state of *req*. This rule avoids deadlocks in ping-pong logic.
RQ-3 *STABIL-ITY*	Request phase signals can change their value either in the cycle following a handshake, regardless if *req* is deasserted or stays asserted.
RQ-OPT-3 *NORE-TIRE*	(Optional) Requests cannot be retired after *req* is asserted. HCI accelerators satisfy this indication, but not all masters on HCI interconnects might be fully compliant.

Table 1.6: HCI-Core Response phase rules.

Rule	Description
RSP-1 *HAND-SHAKE*	For read transactions, a valid handshake occurs in the cycle when both *r_valid* and *lrdy* are asserted. All response phase signals are sampled on handshake cycles.
RSP-2 *NODEAD-LOCK*	The assertion of *r_valid* (transition 0 to 1) cannot depend combinationally on the state of *lrdy*. On the other hand, the assertion of *lrdy* (transition 0 to 1) can depend combinationally on the state of *r_valid*. This rule avoids deadlocks in ping-pong logic.
RSP-3 *STABIL-ITY*	Response phase signals can change their value either in the cycle following a handshake, regardless if *r_valid* is deasserted or stays asserted.
RSP-4 *ORDER-ING*	Response phase signals must follow the same ordering of the requests.

1.2.4 Exchanging data between HWPE-Mem and HWPE-Stream

As HWPEs ultimately consume and produce data to the external shared memory using one or more ports exposing TCDM interfaces, converting data between HWPE-Mem and HWPE-Stream (i.e., exchanging data between the memory-based and the stream-based worlds) is one of the main tasks to be accomplished in the design of an accelerator. The HWPE-Stream and HWPE-Mem protocols are similar by design, which makes the handling of handshakes signficantly easier. The following applies to HWPE-Mem, HWPE-MemDecoupled, and HCI-Core in a similar manner.

Three objectives have to be met:

- HWPE-Stream has no notion of address: to produce a stream out of HWPE-Mem loads, or consume a stream in a series of HWPE-Mem stores, it is necessary to generate addresses according to some rule.
- HWPE-Stream streams can be longer than 32 bits; it is necessary to generate them from / split them into multiple TCDM loads/stores.
- HWPE-Mem addresses may be misaligned with respect to word boundaries, in which case two TCDM loads/stores are necessary to transact a single 32-bit word and strobes have to be also aligned.

In the current version of the HWPE specifications, we address these issues by providing a set of modules which can incrementally be used to solve each of the problems above. This are referred to in a later section.

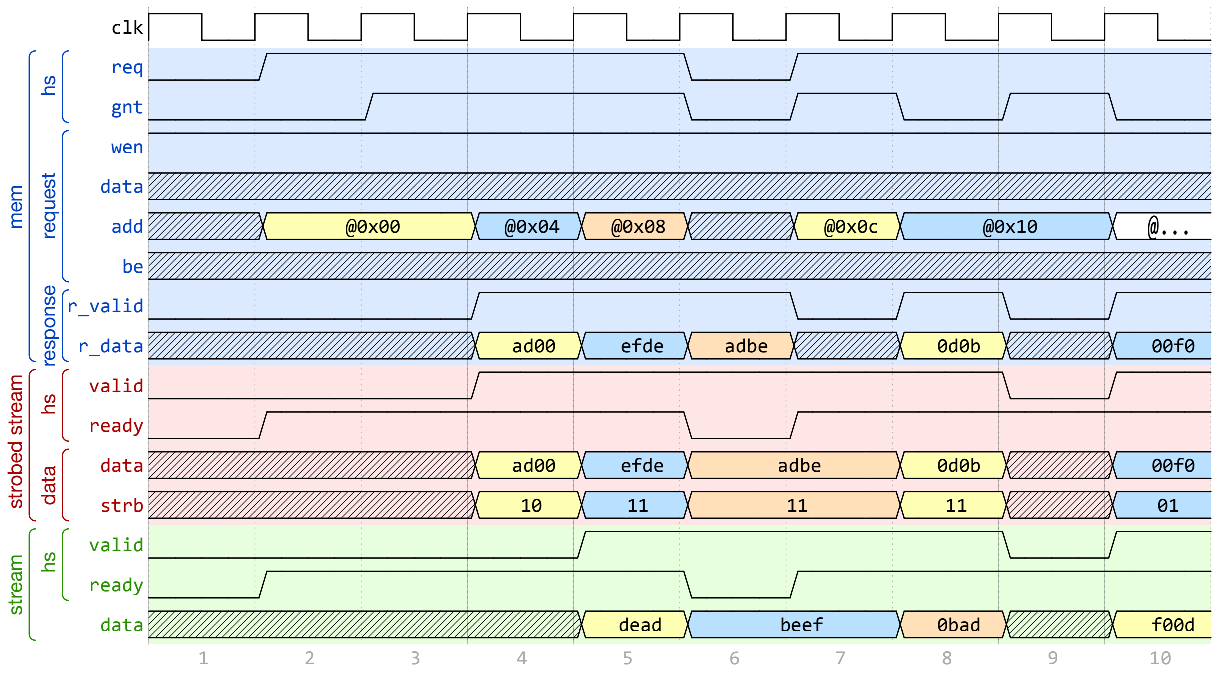


Fig. 1.3: Example of data exchange between a series of HWPE-Mem loads and a HWPE-Stream. Four data packets have to be produced at the sink end of the stream; since data is not well aligned in memory, this results in five loads on the HWPE-Mem interface, which are then transformed in a strobed HWPE-Stream. The stream is then realigned so that the correct four elements are available.

Fig. 1.3, Fig. 1.4 show two examples of transactions going (respectively) from a series of loads on the HWPE-Mem interface to internal HWPE-Streams and from an internal HWPE-Stream to a series of stores on HWPE-Mem. The example focuses on the realignment behavior.

1.3 HWPE-Periph protocol

To enable control, HWPEs typically expose a slave port to the peripheral system interconnect. The slave port follows an extension of the HWPE-Mem protocol which we call HWPE-Periph in this document. The HWPE-Periph protocol is essentially the same one exposed by most peripherals in a PULP system and used by the core to communicate with them.

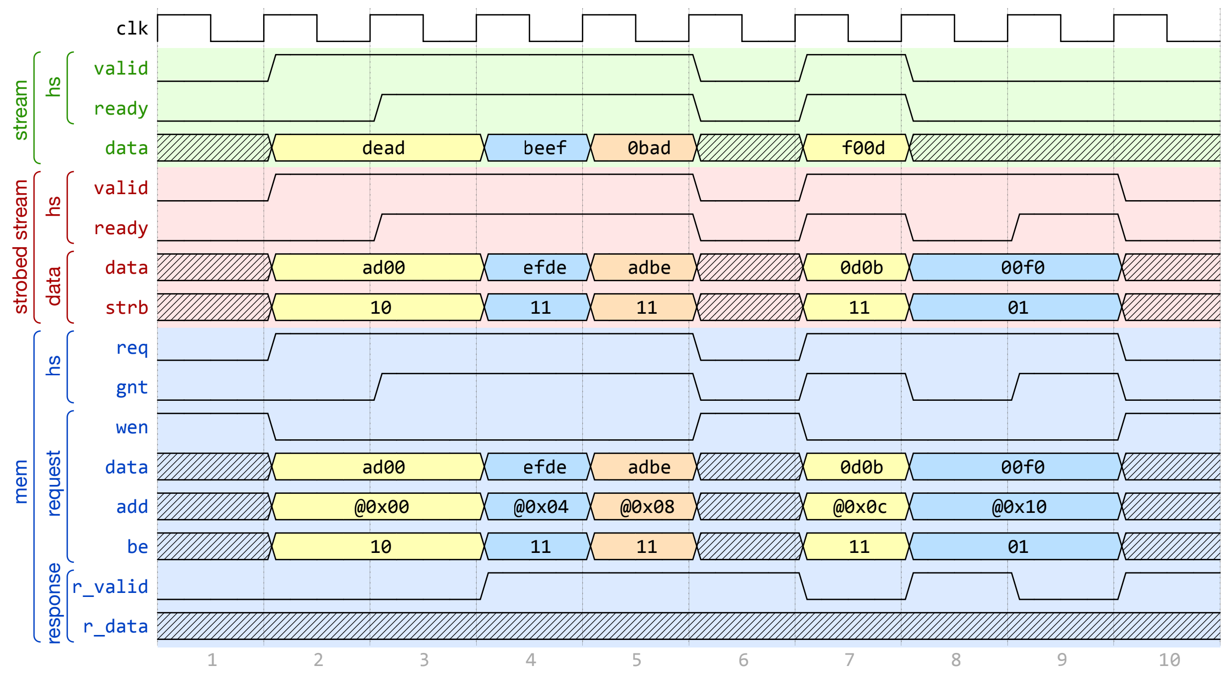


Fig. 1.4: Example of data exchange between a HWPE-Stream and a series of HWPE-Mem stores. Four data packets have to be consumed at the source end of the stream; since data is not well aligned in memory, this results in a strobed HWPE-Stream with five packets, the first and last of which contain also null data. The strobed stream is then converted in a set of five HWPE-Mem store transactions.

Table 1.7: HWPE-Periph signals.

Signal	Size	Description	Direction
req	1 bit	Handshake request signal (1=asserted).	*master* to *slave*
gnt	1 bit	Handshake grant signal (1=asserted).	*slave* to *master*
add	32 bit	Word-aligned memory address.	*master* to *slave*
wen	1 bit	Write enable signal (1=read, 0=write).	*master* to *slave*
be	4 bit	Byte enable signal (1=valid byte).	*master* to *slave*
data	32 bit	Data word to be stored.	*master* to *slave*
id	ID_WIDTH bits	ID used to identify the master (request).	*master* to *slave*
r_data	32 bit	Loaded data word.	*slave* to *master*
r_valid	1 bit	Valid loaded data word (1=asserted).	*slave* to *master*
r_id	ID_WIDTH bits	ID used to identify the master (reply).	*slave* to *master*

The HWPE-Periph protocol is distinguished by the HWPE-Mem protocol by the *id* and *r_id* side channels. These are used in load operations issued through a PERIPH interface: the *id* identifies the master during the request phase, is buffered by the slave peripherals and accompanies the response phase as *r_id*. In this way, multiple masters can distinguish which traffic is related to themselves. For the rest of the purposes related with HWPEs, HWPE-Periph and HWPE-Mem work in the same way. In particular, similarly to HWPE-Mem, PULP clusters will expect *r_valid* to be asserted after write transactions. This is enforced also in HWPE IPs.

CHAPTER

TWO

HARDWARE PROCESSING ENGINES: CONCEPT AND IPS

Hardware Processing Engines (HWPEs) are special-purpose, memory-coupled accelerators that can be inserted in the SoC or cluster of a PULP system to amplify its performance and energy efficiency in particular tasks.

Differently from most accelerators in literature, HWPEs do not rely on an external DMA to feed them with input and to extract output, and they are not (necessarily) tied to a single core. Rather, they operate directly on the same memory that is shared by other elements in the PULP system (e.g. the L1 TCDM in a PULP cluster, or the shared L2 in PULPissimo). Their control is memory-mapped and accessed through a peripheral bus or interconnect. HW-based execution on an HWPE can be readily intermixed with software code, because all that needs to be exchanged between the two is a set of pointers and, if necessary, a few parameters.

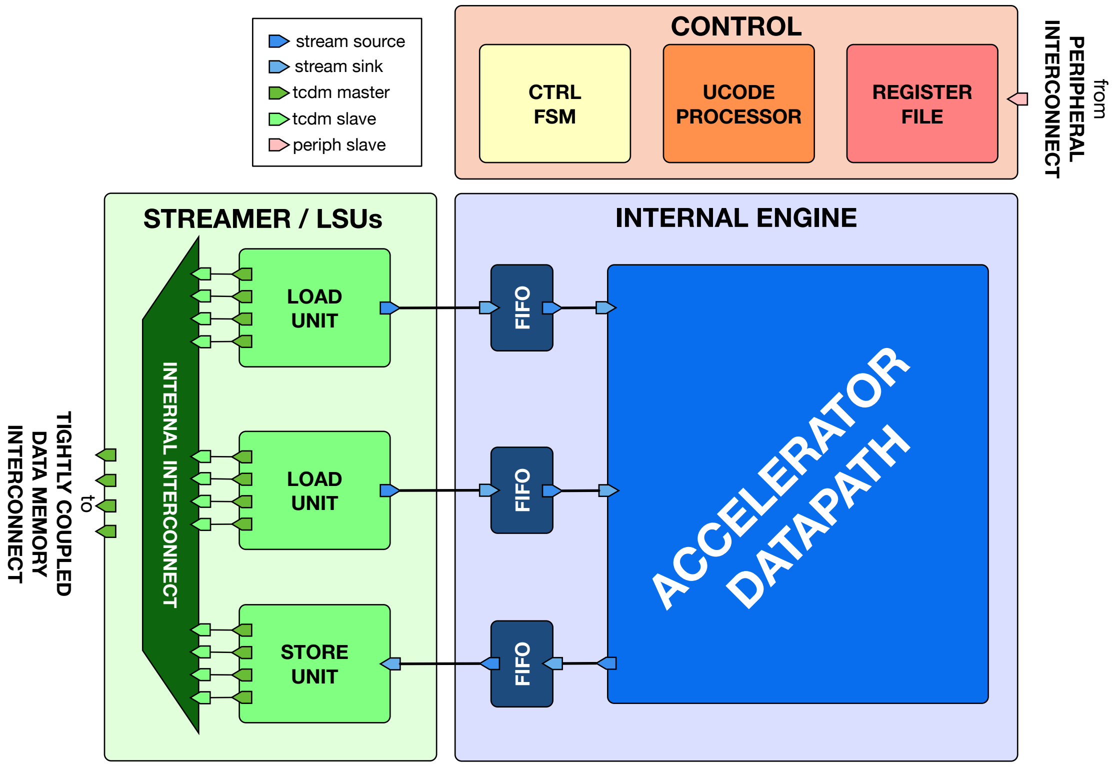


Fig. 2.1: Template of a Hardware Processing Engine (HWPE).

This document defines the interface protocols and modules that are used to enable connecting HWPEs in a PULP system. Typically, such a module is divided in a **streamer** interface towards the memory system, a **control/peripheral** interface used for programming it, and an **engine** containing the actual datapath of the accelerator.

CHAPTER

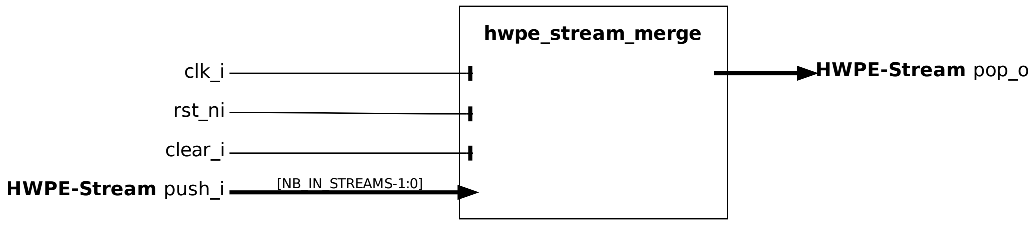
THREE

HWPE INTERFACE MODULES: DATA MOVEMENT & MARSHALING

3.1 Basic modules (HWPE-Stream)

Basic HWPE-Stream management modules are used to select multiple streams, merge multiple streams into one, split a stream in multiple ones, synchronize their handshakes and similar basic "morphing" functionality; or to delay and enqueue streams. Modules performing these functions can be found within the *rtl/basic* and *rtl/fifo* subfolders of the *hwpe-stream* repository.

3.1.1 hwpe_stream_merge



The **hwpe_stream_merge** module is used to merge *NB_IN_STREAMS* input streams into a single, bigger stream. The *data* and *strb* channels from the input streams are bound in order and the *valid* is generated as the AND of all *valid*'s from input streams. The *ready* is broadcasted from the output stream to all input streams.

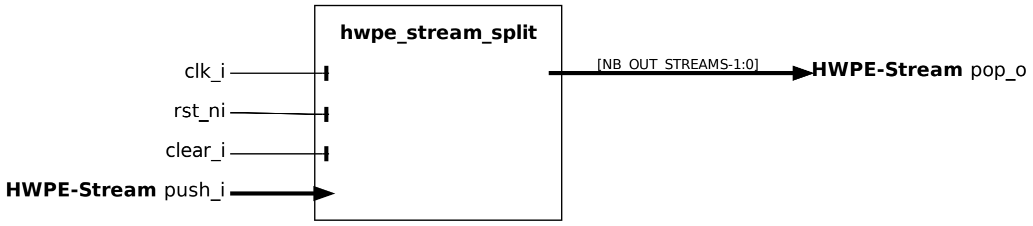
A typical use of this module is to take *NB_IN_STREAMS* 32-bit streams coming from a TCDM load interface to be merged into a single bigger stream.

The following shows an example of the **hwpe_stream_merge** operation:

Table 3.1: **hwpe_stream_merge** design-time parameters.

Name	Default	Description
NB_IN_STREAMS	2	Number of input HWPE-Stream streams.
DATA_WIDTH_IN	32	Width of the input HWPE-Stream streams.

3.1.2 hwpe_stream_split



The **hwpe_stream_split** module is used to split a single stream into *NB_OUT_STREAMS*, 32-bit output streams. The *data* and *strb* channel from the input stream is split in ordered output streams, and the *valid* is broadcast to all outgoing streams. The *ready* is generated as the AND of all *ready*'s from output streams.

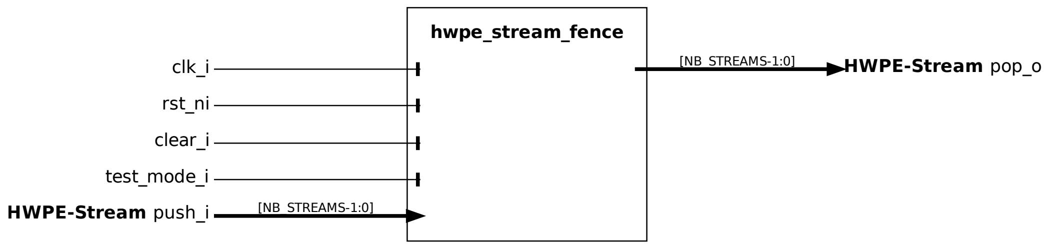
A typical use of this module is to take a multiple-of-32-bit stream coming from within the HWPE and split it into multiple 32-bit streams that feed a TCDM store interface.

The following shows an example of the **hwpe_stream_split** operation:

Table 3.2: **hwpe_stream_split** design-time parameters.

Name	Default	Description
NB_OUT_STREAMS	2	Number of output HWPE-Stream streams.
DATA_WIDTH_IN	128	Width of the input HWPE-Stream stream.

3.1.3 hwpe_stream_fence

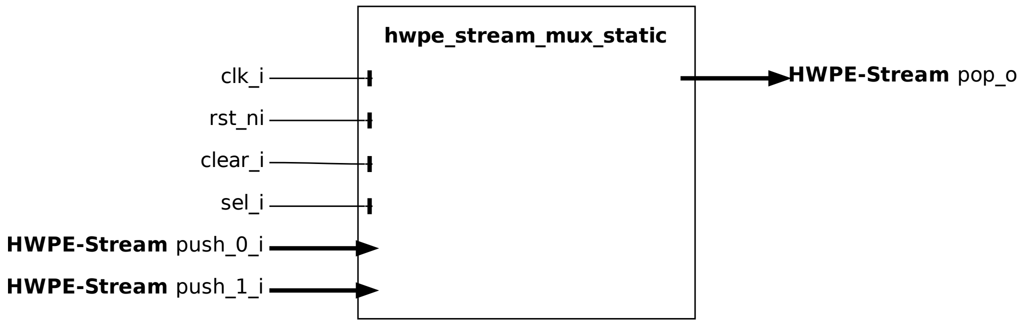


The **hwpe_stream_fence** module is used to synchronize the handshake between *NB_STREAMS* streams. This is necessary, for example, when multiple 32-bit streams are produced from separate TCDM accesses and have to be joined into a single, wider stream.

Table 3.3: **hwpe_stream_fence** design-time parameters.

Name	Default	Description
NB_STREAMS	2	Number of input/output HWPE-Stream streams.
DATA_WIDTH	32	Width of the HWPE-Stream streams.

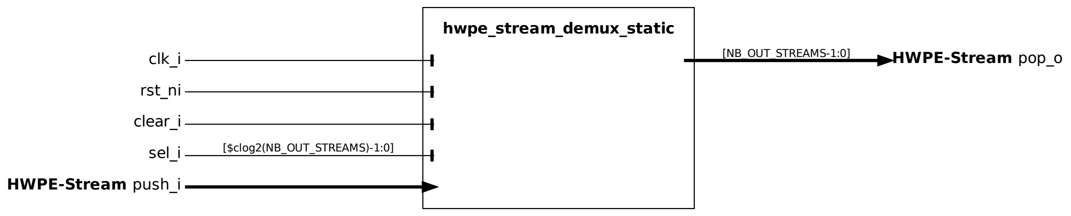
3.1.4 hwpe_stream_mux_static



The **hwpe_stream_mux_static** module is used to statically propagate one of 2 input streams of size *DATA_SIZE* into a single output stream. The multiplexer is static as the selection bit *sel_i cannot be changed* when there are transactions in flight; if the selection bit is changed when transactions are in flight, the result is undefined.

The following shows an example of the **hwpe_stream_mux_static** operation:

3.1.5 hwpe_stream_demux_static



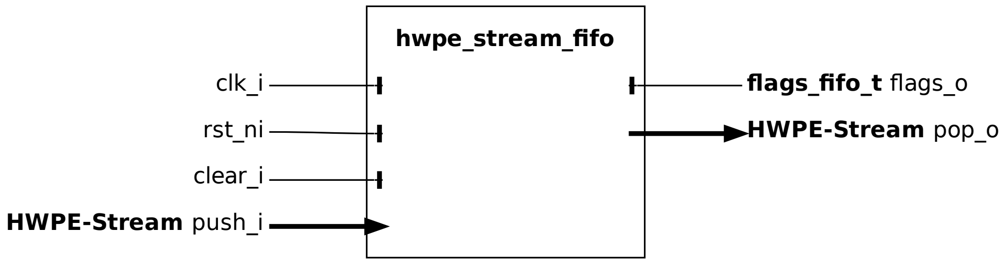
The **hwpe_stream_demux_static** module is used to propagate a single input stream of size *DATA_SIZE* into one of *NB_OUT_STREAMS* output streams. The non-selected output streams are all invalid. The demultiplexer is static as the selection bit *sel_i cannot be changed* when there are transactions in flight; if the selection bit is changed when transactions are in flight, the result is undefined.

The following shows an example of the **hwpe_stream_demux_static** operation:

Table 3.4: **hwpe_stream_demux_static** design-time parameters.

Name	Default	Description
NB_OUT_STREAMS	2	Number of output HWPE-Stream streams.

3.1.6 hwpe_stream_fifo



The **hwpe_stream_fifo** module implements a hardware FIFO queue for HWPE-Stream streams, used to withstand data scarcity (*valid`=0) or backpressure (`ready`=0), decoupling two architectural domains. This FIFO is single-clock and therefore cannot be used to cross two distinct clock domains. The FIFO will lower its `ready* signal on the input stream *push_i* interface when it is completely full, and will lower its *valid* signal on the output stream *pop_o* interface when it is completely empty.

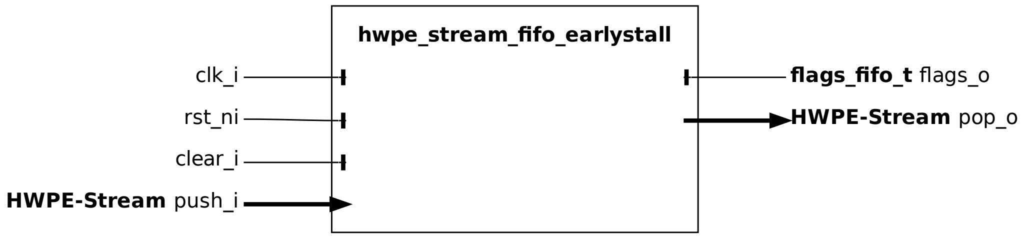
Table 3.5: **hwpe_stream_fifo** design-time parameters.

Name	Default	Description
DATA_WIDTH	32	Width of the HWPE-Streams (typically multiple of 32, but this module does not care).
FIFO_DEPTH	8	Depth of the FIFO queue (multiple of 2).
LATCH_FIFO	0	If 1, use latches instead of flip-flops (requires special constraints in synthesis).
LATCH_FIFO_TEST_WRAP	0	If 1 and *LATCH_FIFO* is 1, wrap latches with BIST wrappers.

Table 3.6: **hwpe_stream_fifo** output flags.

Name	Type	Description
empty	*logic*	1 if the FIFO is currently empty.
full	*logic*	1 if the FIFO is currently full.
push_pointer	*logic[7:0]*	Unused.
pop_pointer	*logic[7:0]*	Unused.

3.1.7 hwpe_stream_fifo_earlystall



The **hwpe_stream_fifo_earlystall** module implements a hardware FIFO queue for HWPE-Stream streams, used to withstand data scarcity (*valid* =1) or backpressure (*ready* =1), decoupling two architectural domains. This FIFO is single-clock and therefore cannot be used to cross two distinct clock domains. The only difference with respect to **hwpe_stream_fifo** is that this version of the FIFO lowers its *ready* signal one cycle earlier, i.e. when it is filled by *FIFO_DEPTH* -1 elements. It will lower its *valid* signal on the output stream *pop_o* interface when it is completely empty.

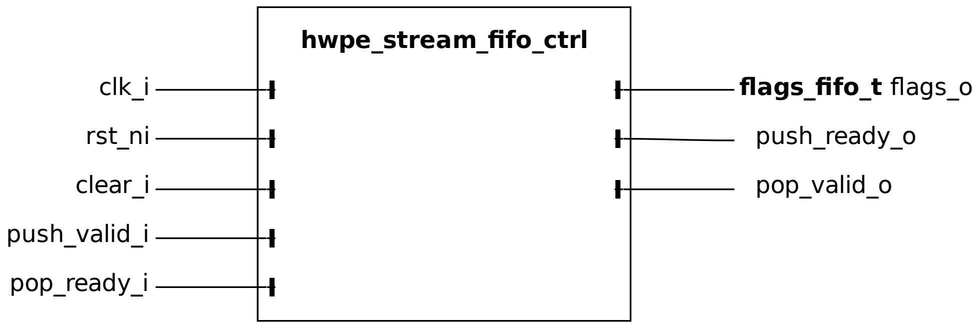
Table 3.7: **hwpe_stream_fifo_earlystall** design-time parameters.

Name	Default	Description
DATA_WIDTH	32	Width of the HWPE-Streams (multiple of 32).
FIFO_DEPTH	8	Depth of the FIFO queue (multiple of 2).
LATCH_FIFO	0	If 1, use latches instead of flip-flops (requires special constraints in synthesis).

Table 3.8: **hwpe_stream_fifo_earlystall** output flags.

Name	Type	Description
empty	*logic*	1 if the FIFO is currently empty.
full	*logic*	1 if the FIFO is currently full.
push_pointer	*logic[7:0]*	Unused.
pop_pointer	*logic[7:0]*	Unused.

3.1.8 hwpe_stream_fifo_ctrl



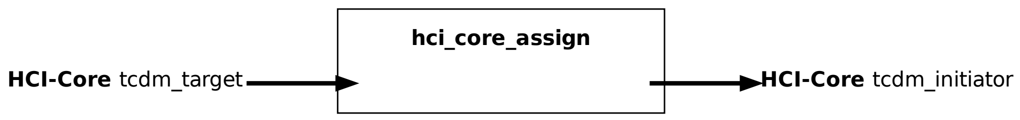
The **hwpe_stream_fifo_ctrl** module implements a hardware FIFO queue similar to that implemented by **hwpe_stream_fifo**, but without any actual interface handshake forced on HWPE-Streams. Instead, it will push its "virtual" handshake on the *push_valid_i*/*push_ready_o* and *pop_valid_o*/*pop_ready_i* signals. It can be used to operate multiple big FIFO queues (e.g. with latches) in a synchronized fashion without breaking the HWPE-Stream protocol.

Table 3.9: **hwpe_stream_fifo_ctrl** design-time parameters.

Name	Default	Description
FIFO_DEPTH	8	Depth of the FIFO queue (multiple of 2).

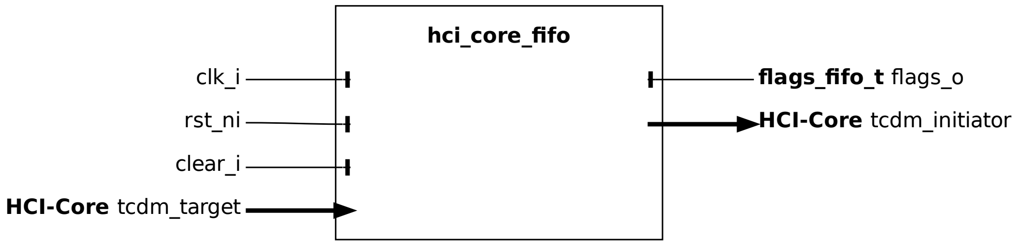
3.2 HCI Core modules

3.2.1 hci_core_assign



The **hci_core_assign** module implements a simple assignment for HCI-Core streams.

3.2.2 hci_core_fifo



The **hci_core_fifo** module implements a hardware FIFO queue for HCI-Core interfaces, used to withstand data scarcity (*req=0*) or backpressure (*gnt=0*), decoupling two architectural domains. This FIFO is single-clock and therefore cannot be used to cross two distinct clock domains. The FIFO treats a HCI-Core load stream as a combination of two 32-bit HWPE-Streams, one going from the *tcdm_initiator* to the *tcdm_target* interface carrying the *addr* (*outgoing stream*); the other from the *tcdm_target* to the *tcdm_initiator* interface, carrying the *r_data* (*incoming stream*).

On the target side, the *req* and *gnt* of the HCI-Core interfaces are mapped on *valid* and *ready* respectively in the outgoing stream. Backpressure on the incoming stream (target side) cannot be enforced by means of the HCI-Core target interface and thus is carried by a specific input *ready_i* that must be generated outside of the TCDM FIFO, typically by a **hwpe_stream_source** module (output *tcdm_fifo_ready_o*). On the initiator side, *req* is mapped to the AND of the incoming stream *ready* signal and the outgoing stream *valid* signal. *gnt* is hooked to the outgoing stream *ready* signal. The *r_valid* is mapped on *valid* in the incoming stream. `_hci_core_fifo_mapping` shows this mapping.

Mapping of HCI-Core and HWPE-Stream signals inside the load FIFO.

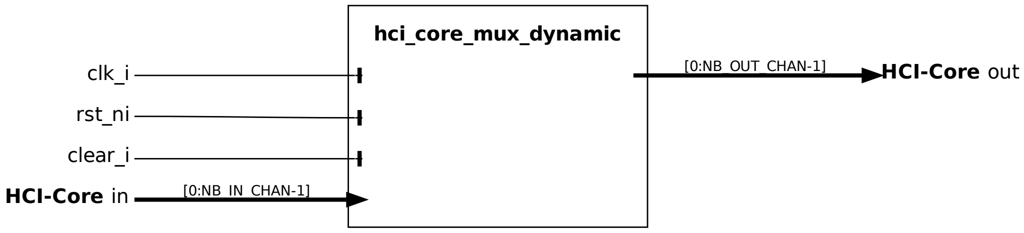
Table 3.10: **hci_core_fifo** design-time parameters.

Name	Default	Description
FIFO_DEPTH	8	Depth of the FIFO queue (multiple of 2).
LATCH_FIFO	0	If 1, use latches instead of flip-flops (requires special constraints in synthesis).

Table 3.11: **hci_core_fifo** output flags.

Name	Type	Description
empty	*logic*	1 if the FIFO is currently empty.
full	*logic*	1 if the FIFO is currently full.
push_pointer	*logic[7:0]*	Unused.
pop_pointer	*logic[7:0]*	Unused.

3.2.3 hci_core_mux_dynamic



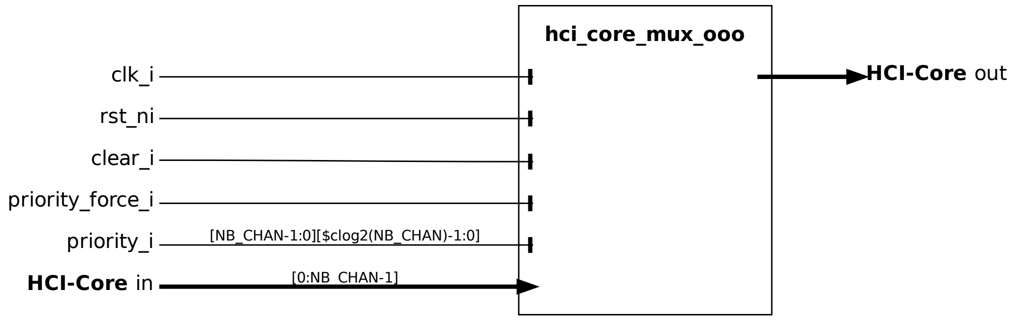
The **HCI multiplexer** can be used to funnel more input "virtual" HCI channels *in* into a smaller set of initiator ports *out*. It uses a round robin counter to avoid starvation, and differs from the modules used within the logarithmic interconnect in that arbitration is performed depending on the round robin counter and not on the target port; in other words, its task is to fill all out ports with requests from the in port, and not to route in requests to a specific out port.

Notice that the multiplexer is not "optimal" in the sense that there is no reorder buffer, so transactions cannot be swapped in-flight to optimally fill the downstream available bandwidth. However, in real accelerators many systematic issues with bandwidth sharing can be solved by upstream HCI FIFOs and by clever reordering of channels, since the dataflow schedule is known. For a multiplexer with reorder buffer, see **hci_core_mux_ooo**.

Table 3.12: **hci_core_mux** design-time parameters.

Name	Default	Description
NB_IN_CHAN	2	Number of input HWPE-Mem channels.
NB_OUT_CHAN	1	Number of output HWPE-Mem channels.

3.2.4 hci_core_mux_ooo

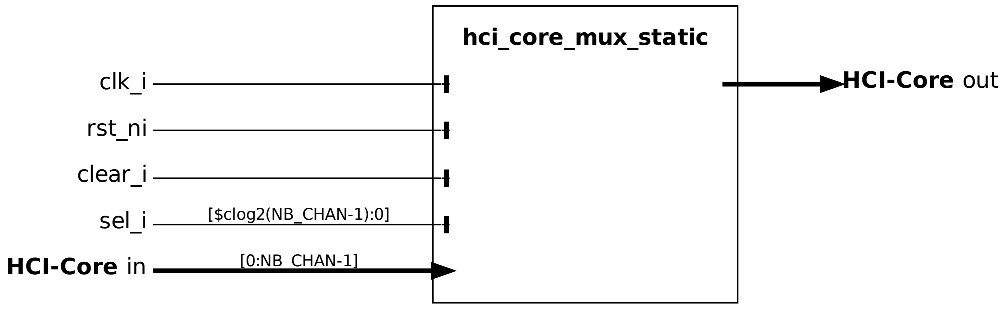


The **HCI dynamic OoO N-to-1 multiplexer** enables to funnel multiple HCI ports into a single one. It supports out-of-order responses by means of ID. As the ID is implemented as user signal, any FIFO coming after (i.e., nearer to memory side) with respect to this block must respect id signals - specifically it must return them identical in the response. At the end of the chain, there will typically be a *hci_core_r_id_filter* block reflecting back all the IDs. This must be placed at the 0-latency boundary with the memory system. Priority is normally round-robin but can also be forced from the outside by setting *priority_force_i* to 1 and driving the *priority_i* array to the desired priority values.

Table 3.13: **hci_core_mux_ooo** design-time parameters.

Name	Default	Description
NB_CHAN	2	Number of input HCI channels.

3.2.5 hci_core_mux_static

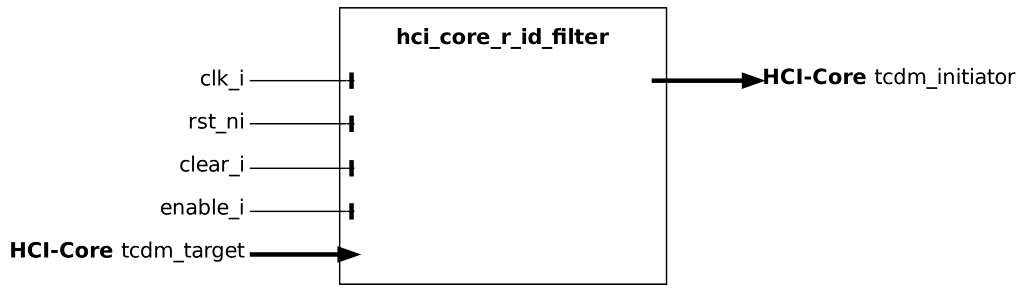


The HCI static multiplexer can be used in place of the dynamic ones when two sets of ports are guaranteed to be used in a strictly alternative fashion.

Table 3.14: **hci_core_mux_static** design-time parameters.

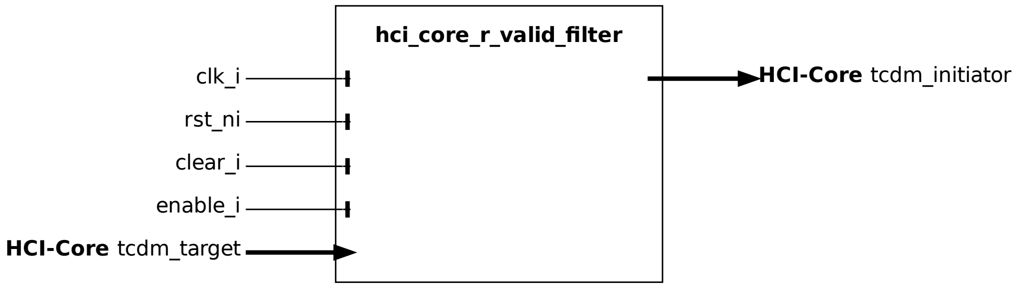
Name	Default	Description
NB_CHAN	2	Number of input HCI channels.

3.2.6 hci_core_r_id_filter



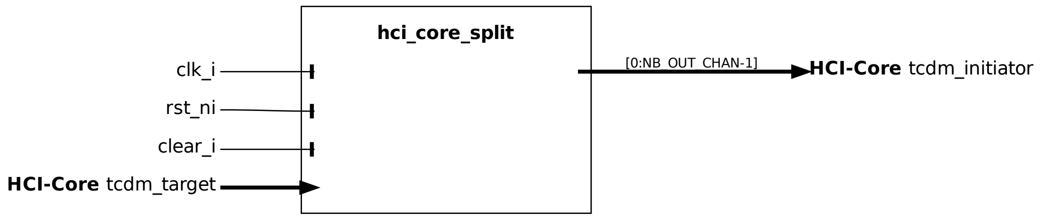
This block filters the id field of the TCDM request, and forwards it to the r_id field of the TCDM response.

3.2.7 hci_core_r_valid_filter



This block filters the *r_valid* field of the TCDM response: when *enable_i* is 1, only responses with *r_valid=1* in case of a read transaction. The block is currently **only** working at the zero-latency boundary between core and memory (it expects that the latency between *gnt* and *r_valid* is exactly one cycle).

3.2.8 hci_core_split



The **hci_core_split** module uses FIFOs to enqueue a split version of the HCI transactions. The FIFO queues evolve in a synchronized fashion on the accelerator side and evolve freely on the TCDM side. In this way, split transactions that can not be immediately brought back to the accelerator do not need to be repeated, massively reducing TCDM traffic. The hci_core_split requires to be followed (not preceded!) by any hci_core_r_id_filter that is used, for example, to implement HCI IDs for the purpose of supporting out-of-order access from a hci_core_mux.

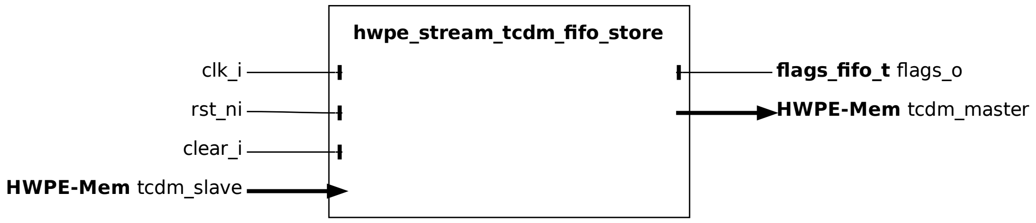
Table 3.15: **hci_core_split** design-time parameters.

Name	Default	Description
NB_OUT_CHAN	2	Number of output channels.
FIFO_DEPTH	0	Depth of internal HCI Core FIFOs.

3.3 Basic modules (HWPE-Mem / HWPE-MemDecoupled - deprecated)

Basic HWPE-Mem management modules are used to delay/enqueue HWPE-MemDecoupled interfaces, multiplex multiple HWPE-Mem, or reorder them before hooking the accelerator to a Tightly-Coupled Data Memory (TCDM). Modules performing these functions can be found within the *rtl/tcdm* subfolder of the *hwpe-stream* repository.

3.3.1 hwpe_stream_tcdm_fifo_store



The **hwpe_stream_tcdm_fifo_store** module implements a hardware FIFO queue for HWPE-MemDecoupled store streams, used to withstand data scarcity (*req`=0) or backpressure (`gnt`=0), decoupling two architectural domains. This FIFO is single-clock and therefore cannot be used to cross two distinct clock domains. The FIFO treats a HWPE-MemDecoupled store stream as a wide HWPE-Stream where, on both sides, the `data* field contains *addr*, *data*, *be* of the input *tcdm_slave*; the *req* and *gnt* of the HWPE-MemDecoupled interfaces are mapped on *valid* and *ready* respectively. The FIFO will lower its *gnt* signal on the slave interface *tcdm_slave* when it is completely full, and will lower its *req* signal on the master interface *tcdm_master* when it is completely empty. `_hwpe_stream_tcdm_fifo_store_mapping` shows this mapping.

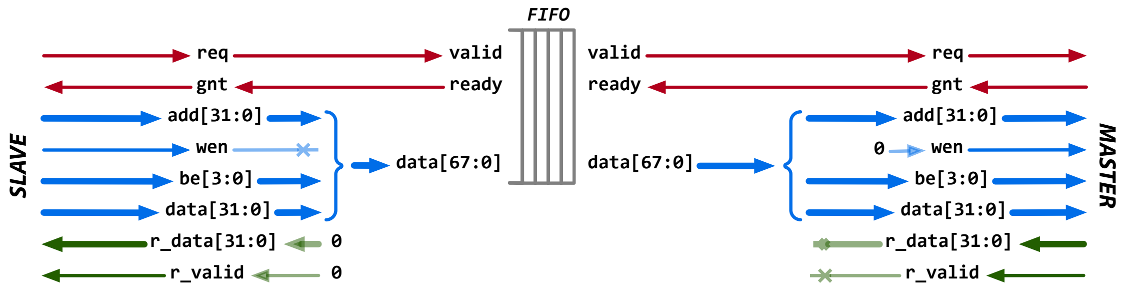


Fig. 3.1: Mapping of HWPE-MemDecoupled and HWPE-Stream signals inside the store FIFO.

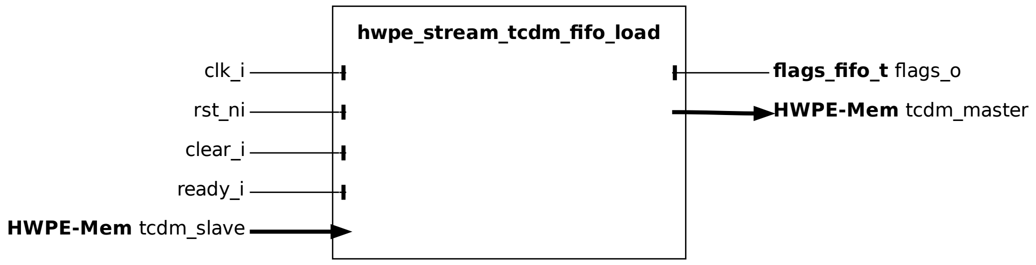
Table 3.16: **hwpe_stream_tcdm_fifo_store** design-time parameters.

Name	Default	Description
FIFO_DEPTH	8	Depth of the FIFO queue (multiple of 2).
LATCH_FIFO	0	If 1, use latches instead of flip-flops (requires special constraints in synthesis).

Table 3.17: **hwpe_stream_tcdm_fifo_store** output flags.

Name	Type	Description
empty	*logic*	1 if the FIFO is currently empty.
full	*logic*	1 if the FIFO is currently full.
push_pointer	*logic[7:0]*	Unused.
pop_pointer	*logic[7:0]*	Unused.

3.3.2 hwpe_stream_tcdm_fifo_load



The **hwpe_stream_tcdm_fifo_load** module implements a hardware FIFO queue for HWPE-MemDecoupled load streams, used to withstand data scarcity (*req`=0) or backpressure (`gnt`=0), decoupling two architectural domains. This FIFO is single-clock and therefore cannot be used to cross two distinct clock domains. The FIFO treats a HWPE-MemDecoupled load stream as a combination of two 32-bit HWPE-Streams, one going from the `tcdm_master* to the *tcdm_slave* interface carrying the *addr* (*outgoing stream*); the other from the *tcdm_slave* to the *tcdm_master* interface, carrying the *r_data* (*incoming stream*).

On the slave side, the *req* and *gnt* of the HWPE-MemDecoupled interfaces are mapped on *valid* and *ready* respectively in the outgoing stream. Backpressure on the incoming stream (slave side) cannot be enforced by means of the HWPE-MemDecoupled slave interface and thus is carried by a specific input *ready_i* that must be generated outside of the TCDM FIFO, typically by a **hwpe_stream_source** module (output *tcdm_fifo_ready_o*). On the master side, *req* is mapped to the AND of the incoming stream *ready* signal and the outgoing stream *valid* signal. *gnt* is hooked to the outgoing stream *ready* signal. The *r_valid* is mapped on *valid* in the incoming stream. `_hwpe_stream_tcdm_fifo_load_mapping` shows this mapping.

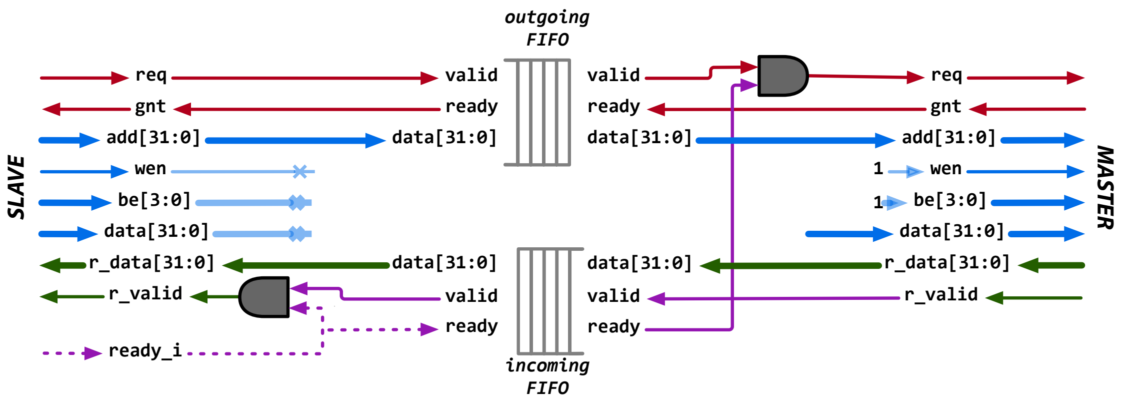


Fig. 3.2: Mapping of HWPE-MemDecoupled and HWPE-Stream signals inside the load FIFO.

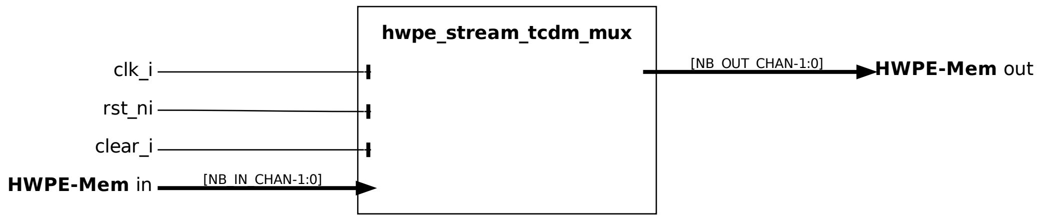
Table 3.18: **hwpe_stream_tcdm_fifo_load** design-time parameters.

Name	Default	Description
FIFO_DEPTH	8	Depth of the FIFO queue (multiple of 2).
LATCH_FIFO	0	If 1, use latches instead of flip-flops (requires special constraints in synthesis).

Table 3.19: **hwpe_stream_tcdm_fifo_load** output flags.

Name	**Type**	**Description**
empty	*logic*	1 if the FIFO is currently empty.
full	*logic*	1 if the FIFO is currently full.
push_pointer	*logic[7:0]*	Unused.
pop_pointer	*logic[7:0]*	Unused.

3.3.3 hwpe_stream_tcdm_mux



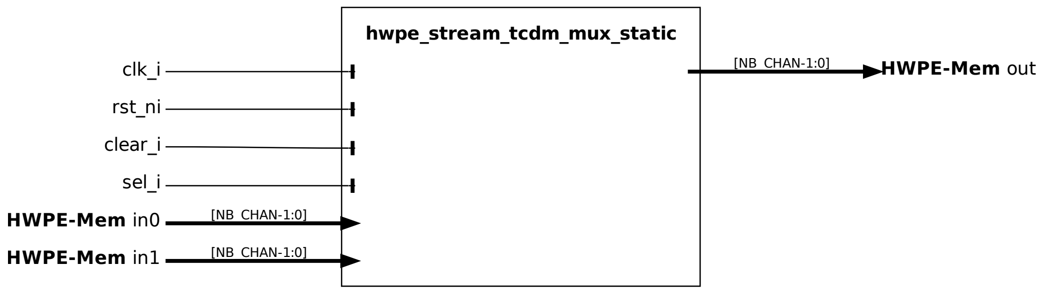
The **TCDM multiplexer** can be used to funnel more input "virtual" TCDM channels *in* into a smaller set of master ports *out*. It uses a round robin counter to avoid starvation, and differs from the modules used within the logarithmic interconnect in that arbitration is performed depending on the round robin counter and not on the slave port; in other words, its task is to fill all out ports with requests from the in port, and not to route in requests to a specific out port.

Notice that the multiplexer is not "optimal" in the sense that there is no reorder buffer, so transactions cannot be swapped in-flight to optimally fill the downstream available bandwidth. However, in real accelerators many systematic issues with bandwidth sharing can be solved by upstream TCDM FIFOs and by clever reordering of channels, since the dataflow schedule is known.

Table 3.20: **hwpe_stream_tcdm_mux** design-time parameters.

Name	Default	Description
NB_IN_CHAN	2	Number of input HWPE-Mem channels.
NB_OUT_CHAN	1	Number of output HWPE-Mem channels.

3.3.4 hwpe_stream_tcdm_mux_static

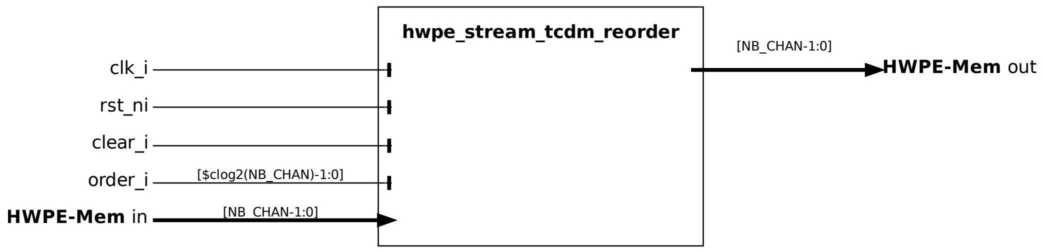


The **hwpe_stream_tcdm_mux_static** module is used to statically share a set of *out* master ports using the HWPE-Mem protocol between two sets of slave ports *in0* and *in1*. It works similarly to the **hwpe_stream_mux_static** and similarly requires a strictly static selector *sel_i*.

Table 3.21: **hwpe_stream_tcdm_mux_static** design-time parameters.

Name	Default	Description
NB_CHAN	2	Number of output HWPE-Mem channels.

3.3.5 hwpe_stream_tcdm_reorder



The **hwpe_stream_tcdm_reorder** block can be used to rotate the order of a set of HWPE-Mem channels depending on an *order_i* input, which can be changed dynamically (e.g. a counter). This is used to "equalize" channels with different probabilities of issuing a request so that the downstream HWPE-Mem channels are used with the same average probability, minimizing the chances for memory starvation.

Table 3.22: **hwpe_stream_tcdm_reorder** design-time parameters.

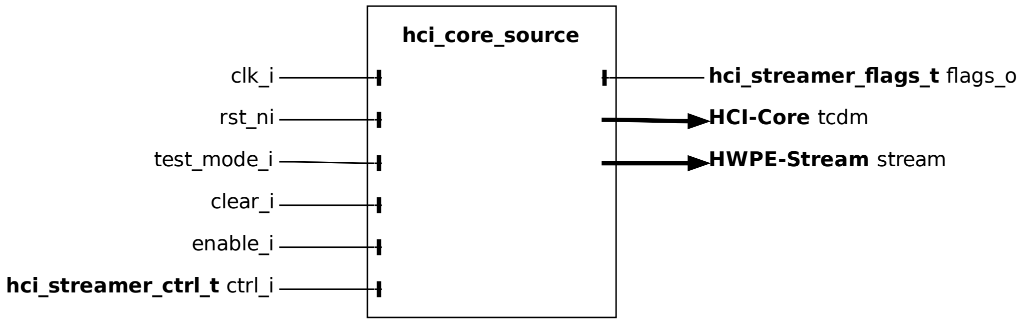
Name	Default	Description
NB_CHAN	2	Number of HWPE-Mem channels.

3.4 HCI Streamer modules

Streamer modules constitute the heart of the IPs use to interface HWPEs with a PULP system. They include all the modules that are used to generate HWPE-Streams from address patterns on the TCDM, including the address generation itself, data realignment to enable access to data located at non-byte-aligned addresses, strobe generation to selectively disable parts of a stream, and the main streamer source and sink modules used to put these functions together. HCI Modules performing these functions can be found within the *rtl/core* subfolder of the *hci* repository.

Two main streamer modules (**hci_core_source** and **hci_core_sink**) are composite of several other IPs, including address generation and strobe generation blocks included in this section, as well as of basic HWPE-Stream management blocks.

3.4.1 hci_core_source



The **hci_core_source** module is the high-level source streamer performing a series of loads on a HCI-Core interface and producing a HWPE-Stream data stream to feed a HWPE engine/datapath. The source streamer is a composite module that makes use of many other fundamental IPs.

Fundamentally, a source streamer acts as a specialized DMA engine acting out a predefined pattern from an **hwpe_stream_addressgen_v3** to perform a burst of loads via a HCI-Core interface, producing a HWPE-Stream data stream from the HCI-Core *r_data* field. By default, the HCI-Core streamer supports delayed accesses using a HCI-Core interface.

Misaligned accesses are supported by widening the HCI-Core data width of 32 bits compared to the HWPE-Stream that gets produced by the streamer. Unused bytes are simply ignored. This feature can be deactivated by unsetting the *MISALIGNED_ACCESS* parameter; in this case, the sink will only work correctly if all data is aligned to a word boundary.

In principle, the source streamer is insensitive to latency. However, when configured to support misaligned memory accesses, the address FIFO depth sets the maximum supported latency. This parameter can be controlled by the *ADDR_MIS_DEPTH* parameter (default 8).

Table 3.23: **hci_core_source** design-time parameters.

Name	Default	Description
LATCH_FIFO	0	If 1, use latches instead of flip-flops (requires special constraints in synthesis).
TRANS_CNT	16	Number of bits supported in the transaction counter of the address generator, which will overflow at 2^ *TRANS_CNT*.
ADDR_MIS_DEPTH	8	Depth of the misaligned address FIFO. This **must** be equal to the max-latency between the HCI-Core *gnt* and *r_valid*.
MISALIGNED_ACCESS	1	If set to 0, the source will not support non-word-aligned HCI-Core accesses.
PASSTHROUGH_FIFO	0	If set to 1, the address FIFO will be capable of fall-through operation (i.e., skipping the FIFO latency entirely).

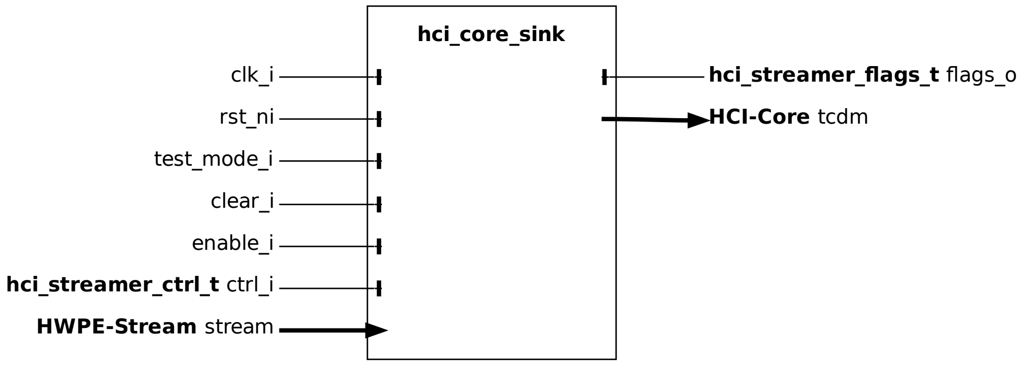
Table 3.24: **hci_core_source** input control signals.

Name	Type	Description
req_start	*logic*	When 1, the source streamer operation is started if it is ready.
addressgen_ctrl	*ctrl_addressgen_v3_t*	Configuration of the address generator (see **hwpe_stream_addresgen_v3**).

Table 3.25: **hci_core_source** output flags.

Name	Type	Description
ready_start	*logic*	1 when the source streamer is ready to start operation, from the first IDLE state cycle on.
done	*logic*	1 for one cycle when the streamer ends operation, in the cycle before it goes to IDLE state .
addressgen_flags	*flags_addressgen_v3_t*	Address generator flags (see **hwpe_stream_addresgen_v3**).

3.4.2 hci_core_sink



The **hci_core_sink** module is the high-level sink streamer performing a series of stores on a HCI-Core interface from an incoming HWPE-Stream data stream from a HWPE engine/datapath. The sink streamer is a composite module that makes use of many other fundamental IPs.

Fundamentally, a sink streamer acts as a specialized DMA engine acting out a predefined pattern from an **hwpe_stream_addressgen_v3** to perform a burst of stores via a HCI-Core interface, consuming a HWPE-Stream data stream into the HCI-Core *data* field. The sink streamer is insensitive to memory latency. This is due to the nature of store streams, which are unidirectional (i.e. *addr* and *data* move in the same direction).

Misaligned accesses are supported by widening the HCI-Core data width of 32 bits compared to the HWPE-Stream that gets consumed by the streamer. The stream is shifted according to the address alignment and invalid bytes are disabled by unsetting their *strb*. This feature can be deactivated by unsetting the *MISALIGNED_ACCESS* parameter; in this case, the sink will only work correctly if all data is aligned to a word boundary.

Table 3.26: **hci_core_sink** design-time parameters.

Name	Default	Description
TCDM_FIFO_DEPTH	2	If >0, the module produces a HWPE-MemDecoupled interface and includes a TCDM FIFO of this depth.
TRANS_CNT	16	Number of bits supported in the transaction counter of the address generator, which will overflow at 2^ *TRANS_CNT*.
MISALIGNED_ACCESS	1	If set to 0, the sink will not support non-word-aligned HWPE-Mem accesses.

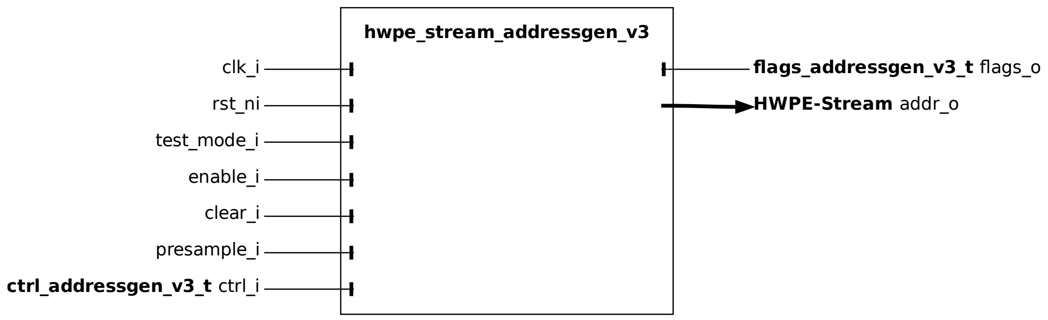
Table 3.27: **hci_core_sink** input control signals.

Name	Type	Description
req_start	*logic*	When 1, the sink streamer operation is started if it is ready.
addressgen_ctrl	*ctrl_addressgen_v3_t*	Configuration of the address generator (see **hwpe_stream_addresgen_v3**).

Table 3.28: **hci_core_sink** output flags.

Name	**Type**	**Description**
ready_start	*logic*	1 when the sink streamer is ready to start operation, from the first IDLE state cycle on.
done	*logic*	1 for one cycle when the streamer ends operation, in the cycle before it goes to IDLE state .
addressgen_flags	*flags_addressgen_v3_t*	Address generator flags (see **hwpe_stream_addresgen_v3**).

3.4.3 hwpe_stream_addressgen_v3


hwpe_stream_addressgen_v3
clk_i
rst_ni
test_mode_i
enable_i
clear_i
presample_i
ctrl_addressgen_v3_t ctrl_i
flags_addressgen_v3_t flags_o
HWPE-Stream addr_o

The **hwpe_stream_addressgen_v3** module is used to generate addresses to load or store HWPE-Stream stream. In this version of the address generator, the address is itself carried within a HWPE-Stream, making it easily stallable. The address generator can be used to generate address from a three-dimensional space, which can be visited with configurable strides in all three dimensions.

The multiple loop functionality is partially overlapped by the functionality provided by the microcode processor *hwce_ctrl_ucode* that can be embedded in HWPEs. The latter is much more flexible and smaller, but less fast.

One iteration is performed per each cycle when *enable_i* is 1 and the output *addr_o* stream is ready. *presample_i* should be 1 in the first cycle in which the address generator can start generating addresses, and no further. The following piece of pseudo-C code resumes the basic functionality provided by the address generator.

```
hwpe_stream_addressgen_v3(
  int base_addr,                                     // base address (byte-aligned)
  int d0_len,    int d1_len,    int tot_len          // d0,d1,total length (in␣
↪number of transactions)
  int d0_stride, int d1_stride, int d2_stride,       // d0,d1,d2 strides (in bytes)
  int *d0_addr,  int *d1_addr,  int *d2_addr,        // d0,d1,d2 addresses (by␣
↪reference)
  int *d0_cnt,   int *d1_cnt,   int *ov_cnt          // d0,d1,overall counters (by␣
↪reference)
) {
  // compute current address
  int current_addr = 0;
  int done = 0;
  if (dim_enable & 0x1 == 0) { // 1-dimensional streaming
    current_addr = base_addr + *d0_addr;
  }
  else if(dim_enable & 0x2 == 0) { // 2-dimensional streaming
    current_addr = base_addr + *d1_addr + *d0_addr;
  }
  else { // 3-dimensional streaming
    current_addr = base_addr + *d2_addr + *d1_addr + *d0_addr;
  }
  // update counters and dimensional addresses
  if(*ov_cnt == tot_len) {
    done = 1;
  }
```

(continues on next page)

(continued from previous page)

```
  if((*d0_cnt < d0_len) || (dim_enable & 0x1 == 0)) {
    *d0_addr = *d0_addr + d0_stride;
    *d0_cnt  = *d0_cnt + 1;
  }
  else if ((*d1_cnt < d1_len) || (dim_enable & 0x2 == 0)) {
    *d0_addr = 0;
    *d1_addr = *d1_addr + d1_stride;
    *d0_cnt  = 1;
    *d1_cnt  = *d1_cnt + 1;
  }
  else {
    *d0_addr = 0;
    *d1_addr = 0;
    *d2_addr = *d2_addr + d2_stride;
    *d0_cnt  = 1;
    *d1_cnt  = 1;
  }
  *ov_cnt = *ov_cnt + 1;
  return current_addr, done;
}
```

Table 3.29: **hwpe_stream_addressgen_v3** design-time parameters.

Name	Default	Description
TRANS_CNT	32	Number of bits supported in the transaction counter, which will overflow at 2^ *TRANS_CNT*.
CNT	32	Number of bits supported in non-transaction counters, which will overflow at 2^ *CNT*.

Table 3.30: **hwpe_stream_addressgen_v3** input control signals.

Name	Type	Description
base_addr	*logic[31:0]*	Byte-aligned base address of the stream in the HWPE-accessible memory.
tot_len	*logic[31:0]*	Total number of transactions in stream; only the *TRANS_CNT* LSB are actually used.
d0_len	*logic[31:0]*	d0 length in number of transactions
d0_stride	*logic[31:0]*	d0 stride in bytes
d0_len	*logic[31:0]*	d0 length in number of transactions
d1_stride	*logic[31:0]*	d1 stride in bytes
d1_len	*logic[31:0]*	d1 length in number of transactions
d2_stride	*logic[31:0]*	d2 stride in bytes
dim_enable_1h	*logic[1:0]*	One-hot switch to enable 3-d counting (11), 2-d (01), or 1-d (00).

Table 3.31: **hwpe_stream_addressgen_v3** output flags.

Name	Type	Description
done	*logic*	1 when the address generation has finished.

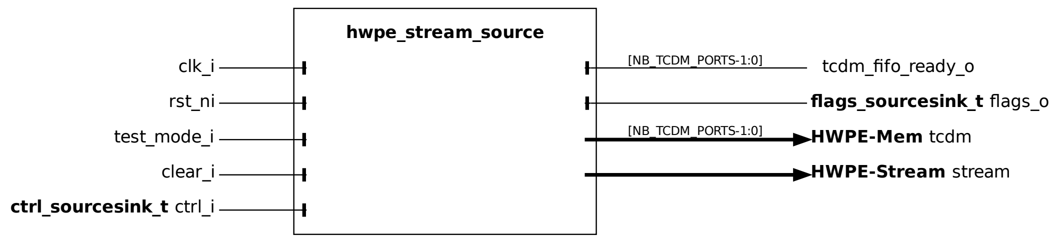
3.5 Plain HWPE-Mem Streamer modules (deprecated)

The “plain” HWPE-Mem Streamer modules, although still functional, have generally been superseded by the HCI Streamer modules. We suggest using those for new designs.

Streamer modules constitute the heart of the IPs use to interface HWPEs with a PULP system. They include all the modules that are used to generate HWPE-Streams from address patterns on the TCDM, including the address generation itself, data realignment to enable access to data located at non-byte-aligned addresses, strobe generation to selectively disable parts of a stream, and the main streamer source and sink modules used to put these functions together. Modules performing these functions can be found within the *rtl/streamer* subfolder of the *hwpe-stream* repository.

Two main streamer modules (**hwpe_stream_source** and **hwpe_stream_sink**) are composite of several other IPs, including address generation and strobe generation blocks included in this section, as well as of basic HWPE-Stream management blocks.

3.5.1 hwpe_stream_source



The **hwpe_stream_source** module is the high-level source streamer performing a series of loads on a HWPE-Mem or HWPE-MemDecoupled interface and producing a HWPE-Stream data stream to feed a HWPE engine/datapath. The source streamer is a composite module that makes use of many other fundamental IPs. Its architecture is shown in :numfig: *_hwpe_stream_source_archi*.

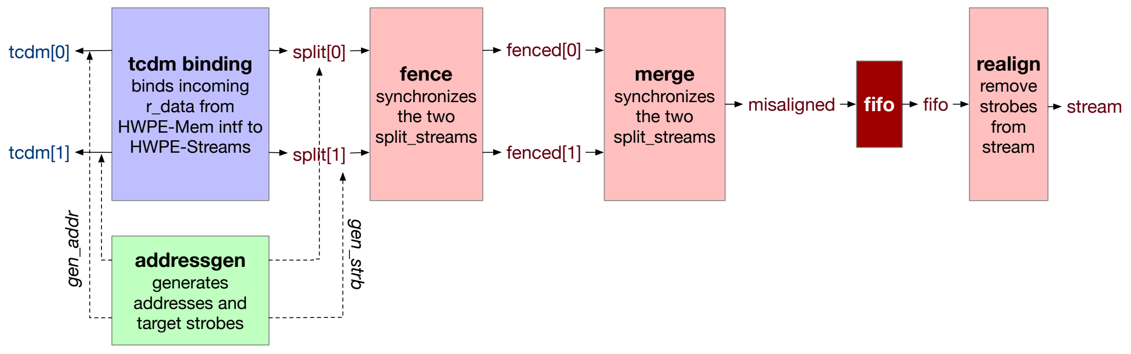

tcdm[0]
tcdm[1]
tcdm binding
binds incoming r_data from HWPE-Mem intf to HWPE-Streams
split[0]
split[1]
fence
synchronizes the two split_streams
fenced[0]
fenced[1]
merge
synchronizes the two split_streams
misaligned
fifo
fifo
realign
remove strobes from stream
stream
gen_addr
addressgen
generates addresses and target strobes
gen_strb

Fig. 3.3: Architecture of the source streamer.

Fundamentally, a source streamer acts as a specialized DMA engine acting out a predefined pattern from an **hwpe_stream_addressgen** to perform a burst of loads via a HWPE-Mem interface, producing a HWPE-Stream data stream from the HWPE-Mem *r_data* field.

Depending on the *DECOUPLED* parameter, the streamer supports delayed accesses using a HWPE-MemDecoupled interface. The source streamer does not include any TCDM FIFO inside on its own; rather, it provides a specific *tcdm_fifo_ready_o* output signal that can be hooked to an external **hwpe_stream_tcdm_fifo_load**. *tcdm_fifo_ready_o* provides a backpressure mechanism from the source streamer to the TCDM FIFO (this is unnecessary in the case of TCDM FIFOs for store).

Table 3.32: **hwpe_stream_source** design-time parameters.

Name	Default	Description
DECOUPLED	0	If 1, the module expects a HWPE-MemDecoupled interface instead of HWPE-Mem.
DATA_WIDTH	32	Width of input/output streams (multiple of 32).
LATCH_FIFO	0	If 1, use latches instead of flip-flops (requires special constraints in synthesis).
TRANS_CNT	16	Number of bits supported in the transaction counter of the address generator, which will overflow at 2^ *TRANS_CNT*.
REALIGNABLE	1	If set to 0, the source will not support non-word-aligned HWPE-Mem accesses.

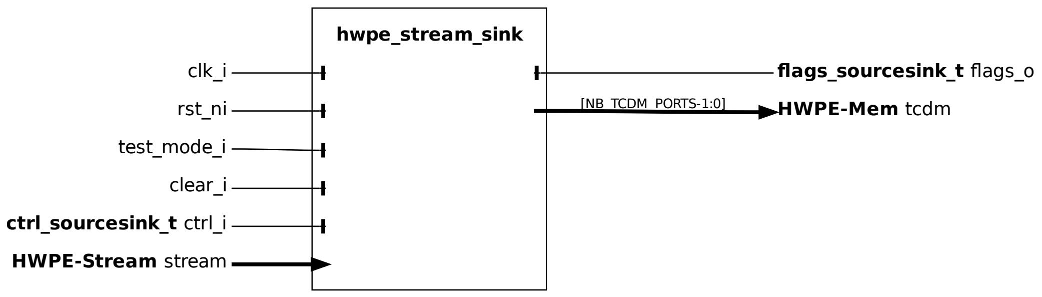
Table 3.33: **hwpe_stream_source** input control signals.

Name	**Type**	**Description**
req_start	*logic*	When 1, the source streamer operation is started if it is ready.
addressgen_ctrl	*ctrl_addressgen_t*	Configuration of the address generator (see **hwpe_stream_addresgen**).

Table 3.34: **hwpe_stream_source** output flags.

Name	**Type**	**Description**
ready_start	*logic*	1 when the source streamer is ready to start operation.
done	*logic*	1 for one cycle when the streamer ends operation.
addressgen_flags	*flags_addressgen_t*	Address generator flags (see **hwpe_stream_addresgen**).
ready_fifo	*logic*	Unused.

3.5.2 hwpe_stream_sink



The **hwpe_stream_sink** module is the high-level sink streamer performing a series of stores on a HWPE-Mem or HWPE-MemDecoupled interface from an incoming HWPE-Stream data stream from a HWPE engine/datapath. The sink streamer is a composite module that makes use of many other fundamental IPs. Its architecture is shown in :numfig: *_hwpe_stream_sink_archi*.

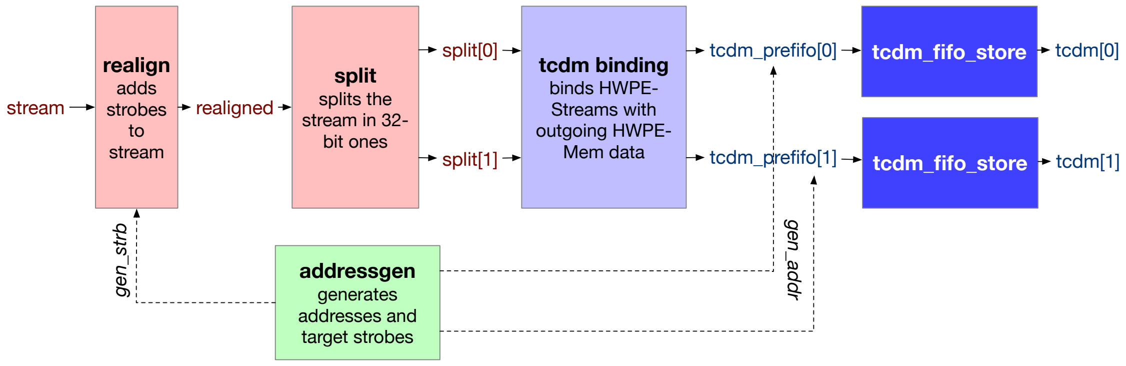


Fig. 3.4: Architecture of the source streamer.

Fundamentally, a ink streamer acts as a specialized DMA engine acting out a predefined pattern from an **hwpe_stream_addressgen** to perform a burst of stores via a HWPE-Mem interface, consuming a HWPE-Stream data stream into the HWPE-Mem *data* field.

The sink streamer indifferently supports standard HWPE-Mem or delayed HWPE-MemDecoupled accesses. This is due to the nature of store streams, that are unidirectional (i.e. *addr* and *data* move in the same direction) and hence insensitive to latency.

Table 3.35: **hwpe_stream_sink** design-time parameters.

Name	Default	Description
TCDM_FIFO_DEPTH	2	If >0, the module produces a HWPE-MemDecoupled interface and includes a TCDM FIFO of this depth.
DATA_WIDTH	32	Width of input/output streams.
LATCH_FIFO	0	If 1, use latches instead of flip-flops (requires special constraints in synthesis).
TRANS_CNT	16	Number of bits supported in the transaction counter of the address generator, which will overflow at 2^ *TRANS_CNT*.
REALIGNABLE	1	If set to 0, the sink will not support non-word-aligned HWPE-Mem accesses.

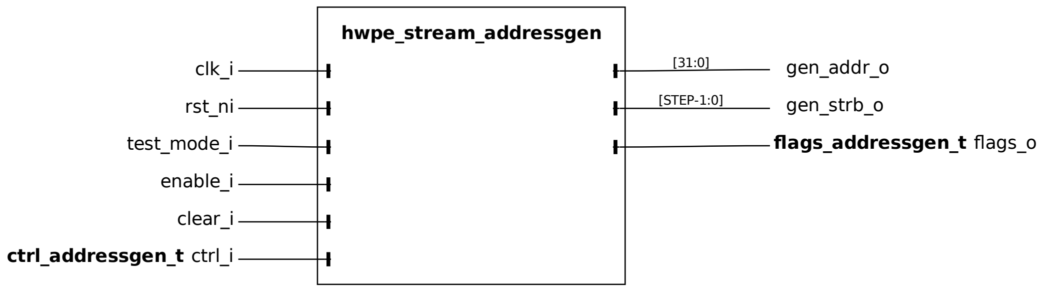
Table 3.36: **hwpe_stream_sink** input control signals.

Name	**Type**	**Description**
req_start	*logic*	When 1, the sink streamer operation is started if it is ready.
addressgen_ctrl	*ctrl_addressgen_t*	Configuration of the address generator (see **hwpe_stream_addresgen**).

Table 3.37: **hwpe_stream_sink** output flags.

Name	**Type**	**Description**
ready_start	*logic*	1 when the sink streamer is ready to start operation.
done	*logic*	1 for one cycle when the streamer ends operation.
addressgen_flags	*flags_addressgen_t*	Address generator flags (see **hwpe_stream_addresgen**).
ready_fifo	*logic*	Unused.

3.5.3 hwpe_stream_addressgen



_**hwpe_stream_addressgen** is DEPRECATED. New designs should use **hwpe_stream_addressgen_v3** instead._

The **hwpe_stream_addressgen** module is used to generate addresses to load or store HWPE-Stream streams, as well as the related byte enable strobes (*gen_addr_o* and *gen_strb_o* respectively). The address generator can be used to generate address from a three-dimensional space of "words", "lines" and "features". Lines and features can be separated by a certain stride, and a roll parameter can be used to reuse the same offsets multiple times.

The multiple loop functionality is partially overlapped by the functionality provided by the microcode processor *hwce_ctrl_ucode* that can be embedded in HWPEs. The latter is much more flexible and smaller, but less fast. When using a single loop in the address generator, the HWPE designer should statically set *line_stride* =0, *feat_length* =1, *feat_stride* =0.

The address generation loop considers three-dimensional vectors, where the three dimensions are called *packet*, *line* and *features* from the innermost to the outermost. One iteration is performed per each cycle when *enable_i* is 1. Feature loops can behave in two different fashions, modeled after the behavior of input/output features in CNNs. The following piece of code resumes the basic functionality provided by the address generator, discarding more complex situations where the address is misaligned (resulting in one more transaction, introduced automatically).

```
int word_addr=0, line_addr=0, feat_addr=0;
int trans_idx=0;
while(trans_idx < trans_size) {
  if(!enable)
    continue;
  for(int feat_idx=0; feat_idx<feat_roll; feat_idx++) { // feature loop
    for(int line_idx=0; line_idx<feat_length; line_idx++) { // line loop
      for(int word_idx=0; word_idx<line_length; word_idx++) { // word loop
        gen_addr = base_addr + feat_addr + line_addr + word_idx * STEP;
      }
      line_addr += line_stride;
    }
    if((loop_outer) && (feat_idx == feat_roll-1)) {
      feat_addr += feat_stride;
      feat_idx  = 0;
    }
    else if ((!loop_outer) && (feat_idx < feat_roll-1)){
      feat_addr += feat_stride;
    }
    else if ((!loop_outer) && (feat_idx == feat_roll-1)){
      feat_addr = 0;
      feat_idx  = 0;
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

Table 3.38: **hwpe_stream_addressgen** design-time parameters.

Name	Default	Description
REALIGN_TYPE	HWPE_STREAM_REALIGN_SOURCE	Type of realignment, can be set to HWPE_STREAM_REALIGN{SOURCE,SINK}.
STEP	4	Step of address generation (untested with != 4).
TRANS_CNT	16	Number of bits supported in the transaction counter, which will overflow at 2^ *TRANS_CNT*.
CNT	10	Number of bits supported in non-transaction counters, which will overflow at 2^ *CNT*.
DELAY_FLAGS	0	If 1, delay the production of flags by one cycle.

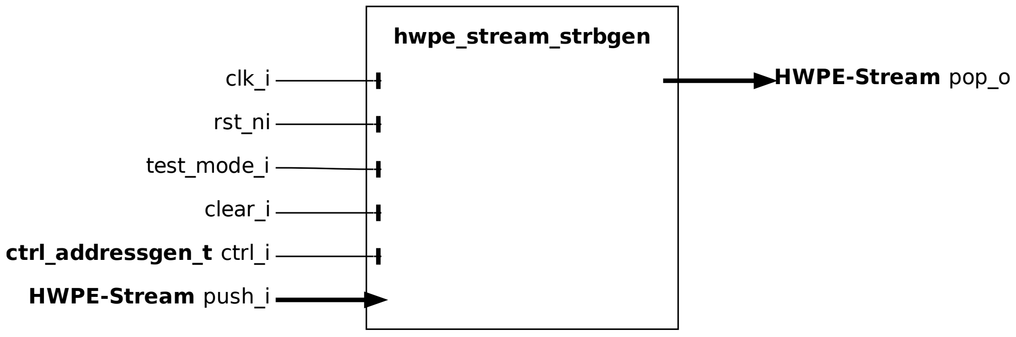
Table 3.39: **hwpe_stream_addressgen** input control signals.

Name	Type	Description
base_addr	*logic[31:0]*	Byte-aligned base address of the stream in the HWPE-accessible memory.
trans_size	*logic[31:0]*	Total size of transaction; only the *TRANS_CNT* LSB are actually used.
line_stride	*logic[15:0]*	Distance between two adjacent lines in bytes.
line_length	*logic[15:0]*	Length of a line in words, rounded by including also incomplete final words.
feat_stride	*logic[15:0]*	Distance between two adjacent features in bytes.
feat_length	*logic[15:0]*	Length of a feature in number of lines.
loop_outer	*logic*	Whether this corresponds to an outer or inner feature loop.
feat_roll	*logic[15:0]*	After this number of features, depending on *loop_outer*, feature index will be rolled back or incremented.
realign_type	*logic*	Unused.
line_length_remainder	*logic[7:0]*	Unused.

Table 3.40: **hwpe_stream_addressgen** output flags.

Name	Type	Description
realign_flags	*ctrl_realign_t*	Control signals to be used for realignment by **hwpe_stream_{source,sink}_realign** modules.
word_update	*logic*	1 when the word loop has been updated.
line_update	*logic*	1 when the line loop has been updated.
feat_update	*logic*	1 when the feature loop has been updated.
in_progress	*logic*	1 when the address generation has progressed.

3.5.4 hwpe_stream_strbgen



The **hwpe_stream_strbgen** module is used to generate strobes for load or store HWPE-Stream streams, in case of incomplete transfers. It uses information passed through the same configuration struct used for the address generator.

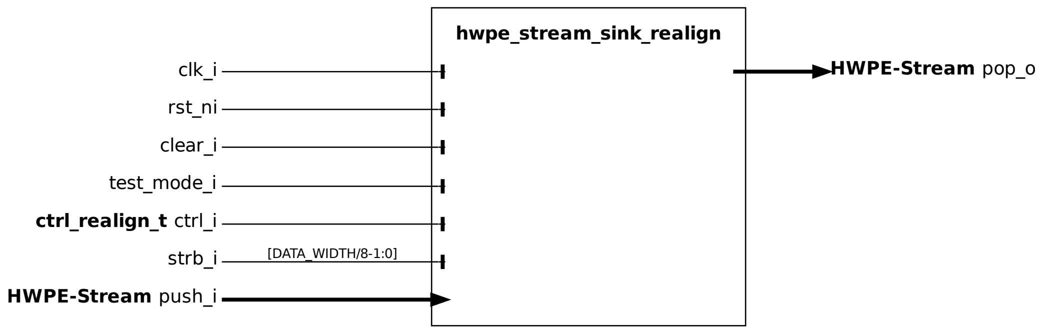
Table 3.41: **hwpe_stream_strbgen** design-time parameters.

Name	Default	Description
DATA_WIDTH	32	Width of input/output streams.

Table 3.42: **hwpe_stream_strbgen** input control signals.

Name	Type	Description
base_addr	*logic[31:0]*	Unused.
trans_size	*logic[31:0]*	Unused.
line_stride	*logic[15:0]*	Unused.
line_length	*logic[15:0]*	Length of a line in words, rounded by including also incomplete final words.
feat_stride	*logic[15:0]*	Unused.
feat_length	*logic[15:0]*	Unused.
loop_outer	*logic*	Unused.
feat_roll	*logic[15:0]*	Unused.
realign_type	*logic*	Unused.
line_length_remainder	*logic[7:0]*	Number of valid bytes in the final word in a line; if 0, the final word is considered fully valid.

3.5.5 hwpe_stream_sink_realign



The **hwpe_stream_sink_realign** module realigns HWPE-Streams to prepare them for storage in memory. Specifically, it rotates *strb* signals according to its control interface, produced along with addresses in the address generator.

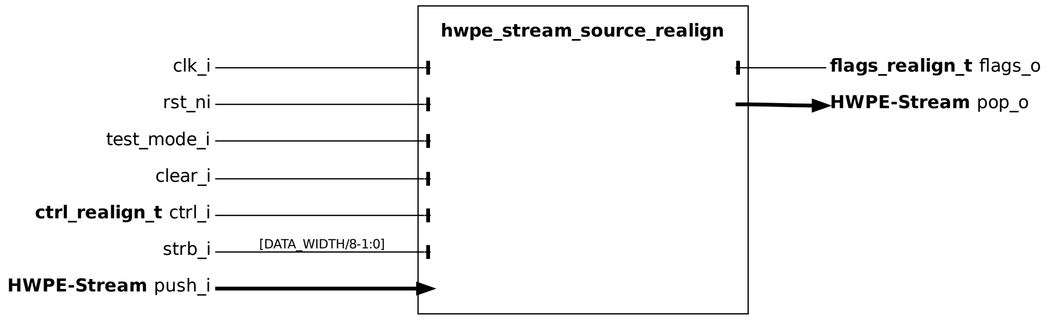
Table 3.43: **hwpe_stream_sink_realign** design-time parameters.

Name	Default	Description
DATA_WIDTH	32	Width of input/output streams.

Table 3.44: **hwpe_stream_sink_realign** input control signals.

Name	Type	Description
enable	*logic*	Unused.
strb_valid	*logic*	Unused.
realign	*logic*	If 1, the realigner is actively used to generate strobed HWPE-Streams. If 0, it is bypassed.
first	*logic*	Strobe at 1 for the first packet in a line.
last	*logic*	Strobe at 1 for the last packet in a line.
last_packet	*logic*	Strobe at 1 for the last packet of the transfer.
line_length	*logic[15:0]*	Unused.

3.5.6 hwpe_stream_source_realign



The **hwpe_stream_source_realign** module realigns HWPE-Streams loaded in a misaligned fashion from memory. Specifically, it rotates *strb* signals according to its control interface, produced along with addresses in the address generator.

Table 3.45: **hwpe_stream_source_realign** design-time parameters.

Name	Default	Description
DECOUPLED	0	If 1, the module expects a HWPE-MemDecoupled interface instead of HWPE-Mem.
DATA_WIDTH	32	Width of input/output streams.
STRB_FIFO_DEPTH	4	Depth of the FIFO queue used for strobes; when full, the realigner will lower its ready signal at the input interface.

Table 3.46: **hwpe_stream_source_realign** input control signals.

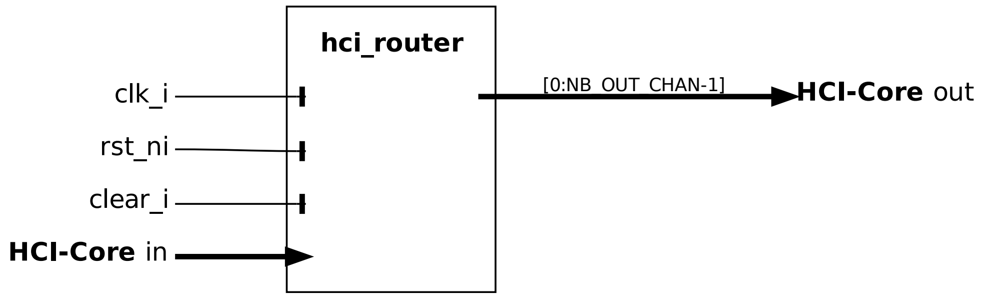
Name	Type	Description
enable	*logic*	If 0, the realigner is fully clock-gated.
strb_valid	*logic*	If 1, the strobe at the *strb_i* interface is considered valid.
realign	*logic*	If 1, the realigner is actively used to generate strobed HWPE-Streams. If 0, it is by-passed.
first	*logic*	Strobe at 1 for the first packet in a line.
last	*logic*	Strobe at 1 for the last packet in a line.
last_packet	*logic*	Strobe at 1 for the last packet of the transfer.
line_length	*logic[15:0]*	Length of a line in words, rounded by including also incomplete final words.

Table 3.47: **hwpe_stream_source_realign** output flags.

Name	Type	Description
decoupled_stall	*logic*	Do not use.

3.6 HCI Interconnect modules

3.6.1 hci_router

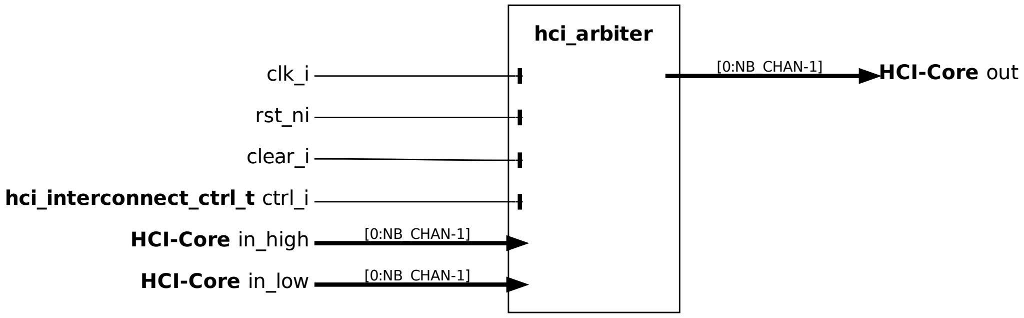


The *hci_router* is a specialized router used to build interconnects in a heterogeneous PULP cluster. It takes as input a single *in* HCI channel of width *DWH* (typically "wide", i.e., greater than 32 bits) that gets routed *without arbitration* to *DWH/32* adjacent *out* targets from a set of *NB_OUT_CHAN out* channels (typically, one per memory bank). Routing is performed by splitting the address of the *DWH*-bit wide word in an *index* (bits *[$clog2(DWH)+2-1:2]*) and an *offset* part (bits *[AWH:$clog2(DWH)+2]*). The index is used to select which *out* targets need to propagate the request, while the offset is used to compute the target-level address for each *out* channel – since word interleaving is assumed, the same address is generally propagated to all targeted *out* channels. However, if *index* > *NB_OUT_CHAN-DWH/32*, then the set of selected targets "wraps around": the first *NB_OUT_CHAN-DWH/32-index out* channels are activated, propagating as address the offset+4. See https://ieeexplore.ieee.org/document/9903915 Sec. II-A (open-access) for details (the router is called a *shallow* router).

Table 3.48: **hci_router** design-time parameters.

Name	Default	Description
FIFO_DEPTH	0	If > 0, insert a HCI FIFO of this depth after the input channel.
NB_OUT_CHAN	8	Number of output HCI channel

3.6.2 hci_arbiter



The *hci_arbiter* is a specialized arbiter used to build interconnects in a heterogeneous PULP cluster, and in particular to arbitrate between two sets of *NB_CHAN* input channels, one with "default high" (*in_high*) and the other with "default low" priority (*in_low*). The arbitration is meant to be performed generally at the direct boundary between the interconnect and the tightly-coupled memory banks. The arbiter uses a starvation-free unbalanced-priority scheme where one of the input channels has by default access to most of the bandwidth guaranteed by the output channels. To prevent starvation effects, depending on the control settings, the other input channel is always granted after a given number of stall cycles. For more details, see:

- https://ieeexplore.ieee.org/document/9903915, Sec. II-A (open-access);
- https://ieeexplore.ieee.org/document/10247945 , Sec. II-A, III-B, and III-C.

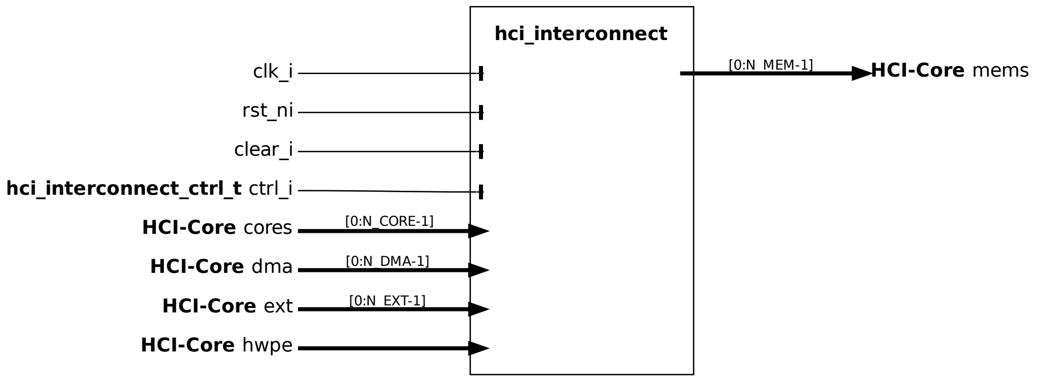
Table 3.49: **hci_arbiter** design-time parameters.

Name	Default	Description
NB_CHAN	2	Number of HCI channels.

Table 3.50: **hci_arbiter** input control signals.

Name	Type	Description
invert_prio	*logic*	When 1, invert priorities between *in_high* and *in_low*.
low_prio_max_stall	*logic[7:0]*	Maximum number of consecutive stalls on low-priority channel.

3.6.3 hci_interconnect



Convenience top-level for the PULP heterogeneous cluster interconnect. It wraps both a logarithmic interconnect (LIC) and an (optional) HCI router meant to realize a LIC and a HWPE branch of the interconnect, respectively. The two branches are (optionally) arbitrated via a HCI arbiter.

Table 3.51: **hci_interconnect** design-time parameters.

Name	Default	Description
N_HWPE	1	Number of HWPEs attached as initiator to the interconnect (LIC or HWPE branch).
N_CORE	8	Number of cores attached as initiator to the interconnect (LIC branch).
N_DMA	4	Number of DMA ports attached as initiator to the interconnect (LIC branch).
N_EXT	4	Number of external ports attached as initiator to the interconnect (LIC branch).
N_MEM	16	Number of memory banks attached as target to the interconnect.
TS_BIT	21	Bit passed to LIC to define test&set aliased memory region.
IW	*N_HWPE+N_CORE+N_DMA+N_EXT*	ID Width.
EXPFIFO	0	Depth of HCI router FIFO.
SEL_LIC	0	Kind of LIC to instantiate (0=regular L1, 1=L2).

3.7 Control interface modules (HWPE-Periph)

The control interface of HWPEs exposes a HWPE-Periph interface that is used to program a memory-mapped register file. Several IPs can be used to compose the control interface, delivering a standard accelerator control interface that is described below. Modules performing these functions can be found within the *rtl/* subfolder of the *hwpe-ctrl* repository.

CHAPTER

FOUR

GITHUB OPEN-SOURCE REPOSITORIES

IPs to build HWPE-based accelerators:

- https://github.com/pulp-platform/hwpe-stream/ : basic streamer IPs, HWPE-Stream SystemVerilog interfaces
- https://github.com/pulp-platform/hwpe-ctrl/ : basic control IPs
- https://github.com/pulp-platform/hci : Heterogeneous Cluster Interconnect streamer IPs and interconnect IPs

Simple examples of HWPEs:

- https://github.com/pulp-platform/hwpe-mac-engine : basic HWPE example with basic streamers - MAC engine with single Multiply-Accumulate
- https://github.com/pulp-platform/hwpe-datamover-example : basic HWPE example with HCI streamers - pure datamover

Complex HWPEs:

- https://github.com/pulp-platform/rbe : Reconfigurable Binary Engine - neural accelerator with flexible precision for weights and activations
- https://github.com/pulp-platform/ne16 : Neural Engine (16 input-channels) - neural accelerator with flexible precision for weights (TinyML applications)
- https://github.com/pulp-platform/neureka : NEureka Neural Engine - neural accelerator with flexible precision for weights (AR/VR applications)
- https://github.com/pulp-platform/redmule : RedMulE (REDuced-precision Matrix MULtipication Engine) is a 8-bit and 16-bit floating-point systolic array

CHAPTER

FIVE

BIBLIOGRAPHY & REFERENCES

Disclaimer: most of these references are the effect of the point of view of myself, Francesco Conti, maintainer and dictator of this site. Most papers are related to work on HWPEs performed in the context of the PULP project during my activity at University of Bologna (2012-ongoing) and ETH Zurich (2015-2020). Although there is a Other authors section, there may be several missing papers using the HWPE IPs and/or a similar template. In case you spot a missing reference, let me know and I'll be happy to amend the list.

5.1 Hardware Accelerators based on HWPE template

- **NEureka**: A. S. Prasad, L. Benini, and F. Conti, "Specialization meets Flexibility: a Heterogeneous Architecture for High-Efficiency, High-flexibility AR/VR Processing," in 2023 60th ACM/IEEE Design Automation Conference (DAC), DAC 2023. [IEEE]
- **SNE**: A. Di Mauro, A. S. Prasad, Z. Huang, M. Spallanzani, F. Conti, and L. Benini, "SNE: an Energy-Proportional Digital Accelerator for Sparse Event-Based Convolutions," in Design, Automation & Test in Europe Conference & Exhibition, DATE 2022. [arXiv]
- **RedMulE**: Y. Tortorella, L. Bertaccini, D. Rossi, L. Benini, and F. Conti, "RedMulE: A Compact FP16 Matrix-Multiplication Accelerator for Adaptive Deep Learning on RISC-V-Based Ultra-Low-Power SoCs," in Design, Automation & Test in Europe Conference & Exhibition, DATE 2022. [arXiv extension]
- **IMA**: A. Garofalo, G. Ottavi, F. Conti, G. Karunaratne, I. Boybat, L. Benini, and D. Rossi, "A Heterogeneous In-Memory Computing Cluster for Flexible End-to-End Inference of Real-World Deep Neural Networks," in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 12, no. 2, pp. 422-435, June 2022, doi: 10.1109/JETCAS.2022.3170152. [arXiv]
- **FFT**: L. Bertaccini, L. Benini, and F. Conti, "To Buffer, or Not to Buffer? A Case Study on FFTAccelerators for Ultra-Low-Power Multicore Clusters", in IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2021. [IEEE]
- **XNE**: F. Conti, P. D. Schiavone, and L. Benini, "XNOR Neural Engine: A Hardware Accelerator IP for 21.6-fJ/op Binary Neural Network Inference.," IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., vol. 37, no. 11, pp. 2940–2951, 2018, doi: 10.1109/TCAD.2018.2857019 (**ESWEEK CODES+ISSS 2018 Best Paper Award**). [arXiv]
- **HWCE**: F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters.," in Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015, 2015, pp. 683–688, [ACM]

5.2 HWPEs on FPGA

- G. Bellocchi, A. Capotondi, F. Conti, and A. Marongiu, “A RISC-V-based FPGA Overlay to Simplify Embedded Accelerator Deployment”, in DSD 2021 Conference.
- P. Meloni, D. Loi, G. Deriu, M. Carreras, F. Conti, A. Capotondi, and D. Rossi, “Exploring NEURAghe: A Customizable Template for APSoC-Based CNN Inference at the Edge.,” IEEE Embed. Syst. Lett., vol. 12, no. 2, pp. 62–65, 2020, doi: 10.1109/LES.2019.2947312.
- P. Meloni et al., “Optimization and deployment of CNNs at the edge: the ALOHA experience.,” in Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019., 2019, pp. 326–332, doi: 10.1145/3310273.3323435
- P. Meloni, G. Deriu, F. Conti, I. Loi, L. Raffo, and L. Benini, “Curbing the roofline: a scalable and flexible architecture for CNNs on FPGA.,” in Proceedings of the ACM International Conference on Computing Frontiers, CF’16, Como, Italy, May 16-19, 2016, 2016, pp. 376–383, doi: 10.1145/2903150.2911715.
- P. Meloni, G. Deriu, F. Conti, I. Loi, L. Raffo, and L. Benini, “A high-efficiency runtime reconfigurable IP for CNN acceleration on a mid-range all-programmable SoC.,” in International Conference on ReConFigurable Computing and FPGAs, ReConFig 2016, Cancun, Mexico, November 30 - Dec. 2, 2016, 2016, pp. 1–8, doi: 10.1109/ReConFig.2016.7857144.

5.3 HWPE-augmented SoC’s

- **Siracusa**: A. S. Prasad, M. Scherer, F. Conti, D. Rossi, A. Di Mauro, M. Eggimann, J. T. Gomez, Z. Li, S. S. Sarwar, Z. Wang, B. De Salvo, and L. Benini, “Siracusa: A 16 nm Heterogenous RISC-V SoC for Extended Reality with At-MRAM Neural Engine.” [arXiv]
- **Siracusa**: M. Scherer, M. Eggimann, A. Di Mauro, A. S. Prasad, F. Conti, D. Rossi, J. T. Gomez, Z. Li, S. S. Sarwar, Z. Wang, B. De Salvo, and L. Benini, “Siracusa: A Low-Power On-Sensor RISC-V SoC for Extended Reality Visual Processing in 16nm CMOS.” ESSCIRC 2023-IEEE 49th European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2023.
- **Marsellus**: F. Conti, G. Paulin, A. Garofalo, D. Rossi, A. Di Mauro, G. Rutishauser, G. Ottavi, M. Eggimann, H. Okuhara, and L. Benini, “Marsellus: A Heterogeneous RISC-V AI-IoT End-Node SoC with 2-to-8b DNN Acceleration and 30%-Boost Adaptive Body Biasing,” in IEEE Journal of Solid-State Circuits, doi: 10.1109/JSSC.2023.3318301. [arXiv]
- **Marsellus**: F. Conti, D. Rossi, G. Paulin, A. Garofalo, A. Di Mauro, G. Rutishauser, G. Ottavi, M. Eggimann, H. Okuhara, V. Huard, O. Montfort, L. Jure, N. Exibard, P. Gouedo, M. Louvat, E. Botte, and L. Benini, “A 12.4 TOPS/W@ 136GOPS AI-IoT system-on-chip with 16 RISC-V, 2-to-8b precision-scalable DNN acceleration and 30%-boost adaptive body biasing,” 2023 IEEE International Solid-State Circuits Conference (ISSCC), ISSCC 2023.
- **Echoes**: M. Sinigaglia, L. Bertaccini, L. Valente, A. Garofalo, S. Benatti, L. Benini, F. Conti, and D. Rossi, “ECHOES: a 200 GOPS/W Frequency Domain SoC with FFT Processor and I2S DSP for Flexible Data Acquisition from Microphone Arrays,” 2023 IEEE International Symposium on Circuits and Systems (ISCAS), Monterey, CA, USA, 2023, pp. 1-5, doi: 10.1109/ISCAS46773.2023.10181862. [arXiv]
- **Darkside**: A. Garofalo, Y. Tortorella, M. Perotti, L. Valente, A. Nadalini, L. Benini, D. Rossi, and F. Conti, “DARKSIDE: A Heterogeneous RISC-V Compute Cluster for Extreme-Edge On-Chip DNN Inference and Training,” in IEEE Open Journal of the Solid-State Circuits Society, vol. 2, pp. 231-243, 2022, doi: 10.1109/OJSSCS.2022.3210082. [open access]
- **Darkside**: A. Garofalo, M. Perotti, L. Valente, Y. Tortorella, A. Nadalini, L. Benini, D. Rossi, and F. Conti, “DARKSIDE: 2.6GFLOPS, 8.7mW Heterogeneous RISC-V Cluster for Extreme-Edge On-Chip DNN Inference

and Training", ESSCIRC 2022- IEEE 48th European Solid State Circuits Conference (ESSCIRC), Milan, Italy, 2022, pp. 273-276, doi: 10.1109/ESSCIRC55480.2022.9911384.

- **Vega**: D. Rossi, F. Conti, M. Eggimann, A. Di Mauro, G. Tagliavini, S. Mach, M. Guermandi, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini, "Vega: A 10-Core SoC for IoT End-Nodes with DNN Acceleration and Cognitive Wake-Up from MRAM-Based State-Retentive Sleep Mode," in IEEE Journal on Solid-State Circuits. [arXiv]
- **Vega**: D. Rossi, F. Conti, M. Eggimann, S. Mach, A. Di Mauro, M. Guermandi, G. Tagliavini, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini, "A 1.3TOPS/W @ 32GOPS Fully Integrated 10-Core SoC for IoT End-Nodes with 1.7W Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode," in Proceedings of the 2021 International Solid State Circuit Conference
- **Quentin**: A. Di Mauro, F. Conti, P. D. Schiavone, D. Rossi, and L. Benini, "Always-On 674uW @ 4GOP/s Error Resilient Binary Neural Networks With Aggressive SRAM Voltage Scaling on a 22-nm IoT End-Node.," IEEE Trans. Circuits Syst., vol. 67–I, no. 11, pp. 3905–3918, 2020, doi: 10.1109/TCSI.2020.3012576.
- **GAP8**: E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, "GAP-8: A RISC-V SoC for AI at the Edge of the IoT.," in 29th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2018, Milano, Italy, July 10-12, 2018, 2018, pp. 1–4, doi: 10.1109/ASAP.2018.8445101.
- **Mia Wallace**: A. Pullini, F. Conti, D. Rossi, I. Loi, M. Gautschi, and L. Benini, "A Heterogeneous Multicore System on Chip for Energy Efficient Brain Inspired Computing.," IEEE Trans. Circuits Syst. II Express Briefs, vol. 65–II, no. 8, pp. 1094–1098, 2018, doi: 10.1109/TCSII.2017.2652982.
- **Fulmine**: F. Conti et al., "An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics.," IEEE Trans. Circuits Syst. I Regul. Pap., vol. 64–I, no. 9, pp. 2481–2494, 2017, doi: 10.1109/TCSI.2017.2698019 (**IEEE CAS Darlington Award 2020**). [arXiv]
- **Fulmine**: F. K. Gürkaynak, R. Schilling, M. Muehlberghuber, F. Conti, S. Mangard, and L. Benini, "Multi-core data analytics SoC with a flexible 1.76 Gbit/s AES-XTS cryptographic accelerator in 65 nm CMOS.," in Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems, CS2 at HiPEAC 2017, Stockholm, Sweden, January 24, 2017, 2017, pp. 19–24, doi: 10.1145/3031836.3031840.
- **Mia Wallace**: A. Pullini, F. Conti, D. Rossi, I. Loi, M. Gautschi, and L. Benini, "A heterogeneous multi-core system-on-chip for energy efficient brain inspired vision.," in IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22-25, 2016, 2016, p. 2910, doi: 10.1109/ISCAS.2016.7539213.

5.4 HWPE template

- F. Conti, C. Pilkington, A. Marongiu, and L. Benini, "He-P2012: Architectural heterogeneity exploration on a scalable many-core platform.," in IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2014, Zurich, Switzerland, June 18-20, 2014, 2014, pp. 114–120, doi: 10.1109/ASAP.2014.6868645.
- P. Burgio, G. Tagliavini, F. Conti, A. Marongiu, and L. Benini, "Tightly-coupled hardware support to dynamic parallelism acceleration in embedded shared memory clusters.," in Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014, 2014, pp. 1–6, doi: 10.7873/DATE.2014.169.
- F. Conti, A. Marongiu, and L. Benini, "Synthesis-friendly techniques for tightly-coupled integration of hardware accelerators into shared-memory multi-core clusters.," in Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013, Montreal, QC, Canada, September 29 - October 4, 2013, 2013, p. 5:1-5:10, doi: 10.1109/CODES-ISSS.2013.6658992

- M. Dehyadegari, A. Marongiu, M. R. Kakoee, S. Mohammadi, N. Yazdani and L. Benini, "Architecture Support for Tightly-Coupled Multi-Core Clusters with Shared-Memory HW Accelerators," in IEEE Transactions on Computers, vol. 64, no. 8, pp. 2132-2144, 1 Aug. 2015, doi: 10.1109/TC.2014.2360522.

5.5 Other authors

- **TinyVers** from Marian Verhelst's team at KU Leuven: V. Jain, S. Giraldo, J. De Roose, B. Boons, L. Mei, M. Verhelst, "TinyVers: A 0.8-17 TOPS/W, 1.7 W-20 mW, Tiny Versatile System-on-chip with State-Retentive eMRAM for Machine Learning Inference at the Extreme Edge", VLSI 2022, doi: 10.1109/VLSITechnologyandCir46769.2022.9830409.
- **DIANA** from Marian Verhelst's team at KU Leuven: K. Ueyoshi et al., "DIANA: An End-to-End Energy-Efficient Digital and ANAlog Hybrid Neural Network SoC," 2022 IEEE International Solid- State Circuits Conference (ISSCC), 2022, pp. 1-3, doi: 10.1109/ISSCC42614.2022.9731716.
- **PULPO** from Christoph Studer's team at ETH Zurich: O. Castañeda, L. Benini and C. Studer, "A 283 pJ/b 240 Mb/s Floating-Point Baseband Accelerator for Massive MU-MIMO in 22FDX," ESSCIRC 2022-IEEE 48th European Solid State Circuits Conference (ESSCIRC), 2022, pp. 357-360, doi: 10.1109/ESSCIRC55480.2022.9911311.
- **Bit-Serial NE**: M. Capra, F. Conti, and M. Martina, "A Multi-Precision Bit-Serial Hardware Accelerator IP for Deep Learning Enabled Internet-of-Things", in IEEE MWSCAS 2021.

CHAPTER

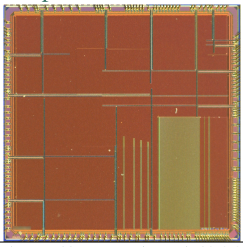
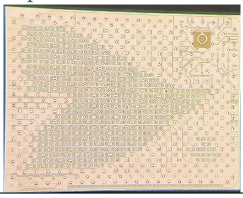
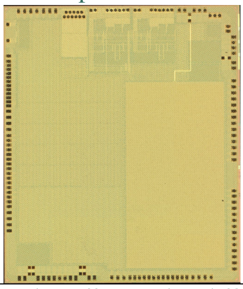
SIX

CHIP GALLERY

Disclaimer: most of these references are the effect of the point of view of myself, Francesco Conti, maintainer and dictator of this site. Most chips shown here are related to work on HWPEs performed in the context of the PULP project during my activity at University of Bologna (2012-ongoing) and ETH Zurich (2015-2020). Although there is a Other authors section, there may be several missing chips using the HWPE IPs and/or a similar template. In case you spot a missing reference, let me know.

6.1 PULP chips

Table 6.1: PULP chips

Chip	Year	Notes
Siracusa http://asic.ethz.ch/2022/Siracusa.html 	2022	PULP cluster with *NEureka*
Darkside http://asic.ethz.ch/2021/Darkside.html Angelo Garofalo Matteo Perotti Luca Valente Yvan Tortorella Alessandro Nadalini Alexandre Levisse Riccardo Gandolfi Carla Ohanesian Francesco Conti Andrew Simon Pietro Maltoni Marco Rios DARKSIDE	2021	PULP cluster with *RedMulE*, *DepthWise Engine*, *DataMover Engine*
Echoes http://asic.ethz.ch/2021/Echoes.html ECHOES a PULP chip by Mattia Sinigaglia, Luca Valente, Luca Bertaccini, Angelo Garofalo, Francesco Conti, Davide Rossi	2021	PULPissimo with *FFT HWPE*
Kraken http://asic.ethz.ch/2021/Kraken.html	2021	PULP cluster with *SNE* and *PULPO* (they do not directly use HWPE IPs, but they follow the same HWPE template)
Marsellus http://asic.ethz.ch/2021/Marsellus.html 	2021	PULP cluster with *RBE* (https://github.com/pulp-platform/rbe)
Vega http://asic.ethz.ch/2020/Vega.html 	2020	PULP cluster with *HWCE* v4
Xavier http://asic.ethz.ch/2019/Xavier.html	2019	PULPissimo with *QNE*

CHAPTER SEVEN

TEAM

7.1 Current team

- **Francesco Conti**, team lead, Tenure-Track Assistant Professor @ UNIBO, Italy
- **Arpan Prasad**, N-EUREKA Neural Engine designer, PhD student @ ETHZ, Switzerland
- **Yvan Tortorella**, RedMulE (FP16 MatMul Engine) designer, PhD student @ UNIBO, Italy
- **Luca Bertaccini**, FFT-HWPE designer, PhD student @ ETHZ, Switzerland
- **Alessio Burrello**, SW integration in DORY, Postdoc @ POLITO, Italy
- **Luka Macan**, SW integration in DORY/Deeploy, PhD student @ UNIBO, Italy
- **Alessandro Nadalini**, course tutoring, PhD student @ UNIBO, Italy
- **Luigi Ghionda**, FP FFT-HWPE designer, master thesis student @ UNIBO, Italy
- **Lorenzo Greco**, in-memory computing acceleration, master thesis student @ UNIBO, Italy

7.2 Past members / Members at large

- **Gianna Paulin**, Reconfigurable Binary Engine designer, former PhD student @ ETHZ, now at Axelera AI
- **Pietro Maltoni**, former master thesis student @ UNIBO, now at GreenWaves Technologies
- **Riccardo Gandolfi**, former master thesis student @ UNIBO, now at GreenWaves Technologies
- **Aurora Di Giampietro**, former master thesis student @ UNIBO+ETHZ, now at OnSemi

CHAPTER

EIGHT

DOCUMENT REVISIONS

Rev.	Date	Author	Description
1.0	14/01/18	Francesco Conti	First draft of the specifications.
1.1	19/01/18	Francesco Conti	Added description of *hwpe-stream*, *hwpe-ctrl* modules.
1.2	26/01/18	Francesco Conti	Added specification of the microcode processor.
1.3	10/02/18	Francesco Conti	Removed some unnecessary constraints on TCDM prot.
1.4	27/03/19	Francesco Conti	Switched to RST; major rehaul.
2.0	16/06/22	Francesco Conti	Adding HCI; chips; papers.
2.0.1	28/11/23	Francesco Conti	Minor updates.
2.0.2	20/01/24	Francesco Conti	Minor updates.
3.0	16/03/24	Francesco Conti	Update to HCIv2.

LIST OF FIGURES

LIST OF TABLES