

Hardware Processing Engines  
Interface Specifications

January 2018  
Revision 1.2

Francesco Conti  
[f.conti@unibo.it](mailto:f.conti@unibo.it)  
 [fconti@iis.ee.ethz.ch](mailto:fconti@iis.ee.ethz.ch)

Energy-Efficient Embedded Systems Laboratory  
University of Bologna, Italy

Integrated Systems Laboratory   
ETH Zürich, Switzerland

Copyright 2018 ETH Zurich and University of Bologna – all rights reserved.

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. |

Table of Contents

1 Introduction 6

2 HWPE Protocols 7

2.1 HWPE-Stream protocol 7

2.2 TCDM protocol 7

2.3 Exchanging data between TCDM and HWPE-Stream 8

2.4 PERIPH protocol 10

3 HWPE interface modules 11

3.1 HWPE-Stream management modules 11

3.1.1 Stream merger 11

3.1.2 Stream splitter 11

3.1.3 Static stream multiplexer 11

3.1.4 Static stream demultiplexer 11

3.1.5 Stream FIFO 11

3.1.6 Stream Buffer 11

3.1.7 Source realigner 12

3.1.8 Sink realigner 12

3.2 TCDM / HWPE-Stream interface modules 12

3.2.1 Address generator 12

3.2.2 Source 12

3.2.3 Sink 12

3.3 TCDM management modules 13

3.3.1 TCDM FIFO (loads) 13

3.3.2 TCDM FIFO (stores) 13

3.3.3 TCDM dynamic multiplexer 13

3.3.4 TCDM static multiplexer 13

3.3.5 TCDM reorder block 13

3.4 PERIPH and controller modules 13

3.4.1 Microcode processor 13

3.4.2 Slave interface and register file 14

3.4.3 Sequential multiplier 14

4 Register file map 15

4.1 Control registers and generic configuration registers 15

4.1.1 TRIGGER register 16

4.1.2 ACQUIRE register 17

4.1.3 FINISHED\_JOBS register 18

4.1.4 STATUS register 19

4.1.5 RUNNING\_JOB register 20

4.1.6 SOFT\_CLEAR register 21

5 Microcode specification and processor 22

5.1 Microcode ISA 22

6 Bibliography 23

# Introduction

*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 the 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.

For more information on HWPEs and their properties, see references [1]-[5].

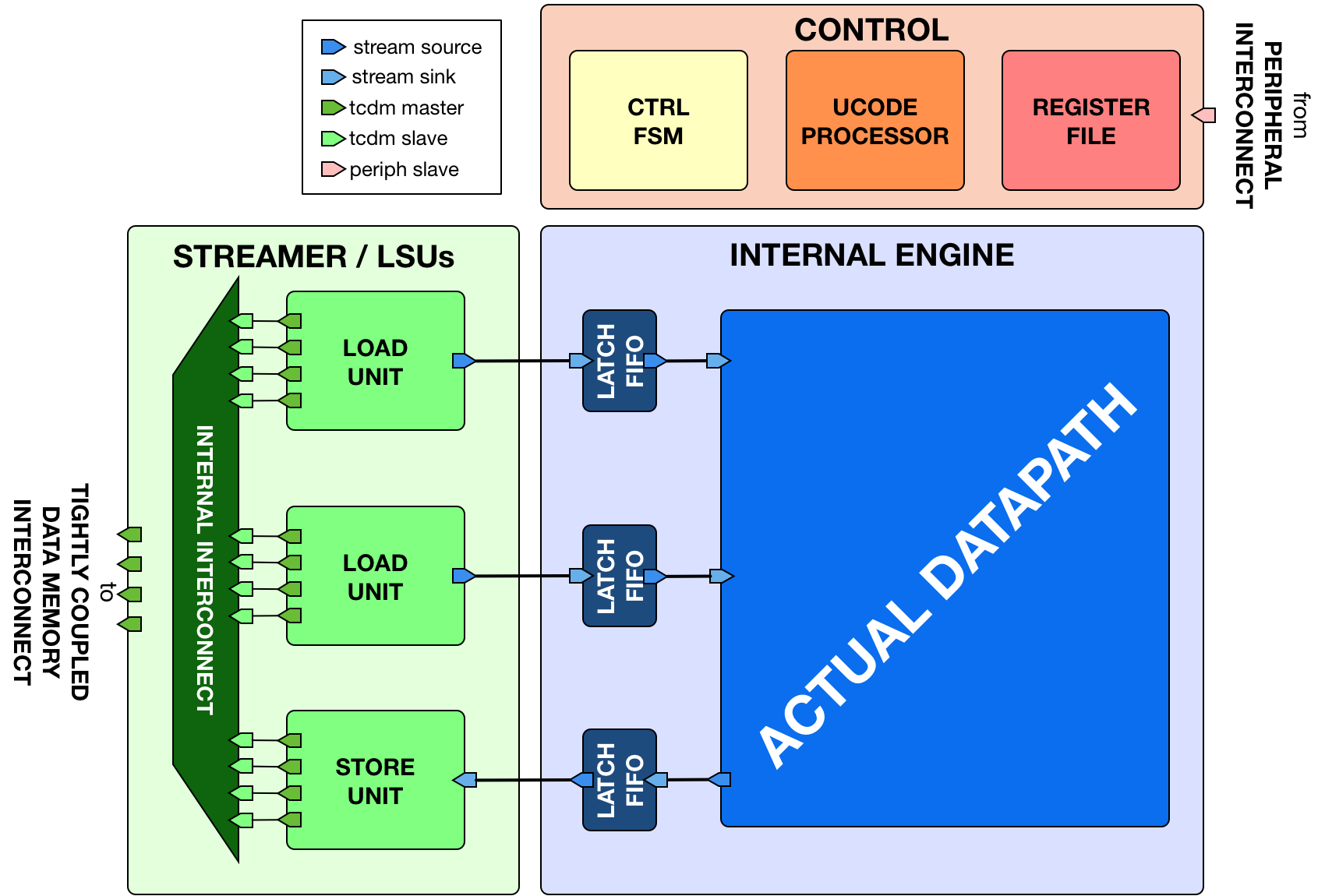


Figure 1: Example of a Hardware Processing Engine.

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.

# HWPE Protocols

## 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.

HWPE-Stream streams flow from a *source* to a *sink* direction. Table 1 reports the signals used by the HWPE-Stream protocol.

| **Signal** | **Size** | **Description** | **Direction** |
| --- | --- | --- | --- |
| *data* | Multiple of 32 bits | The data payload transported by the stream. | Source 🡪 Sink |
| *strb* | Size(*data*)/8 | Optional. Indicates which bytes in the data payload are valid (1=valid byte). | Source 🡪 Sink |
| *valid* | 1 bit | Handshake valid signal (1=asserted). | Source 🡪 Sink |
| *ready* | 1 bit | Handshake ready signal (1=asserted). | Sink 🡪 Source |

Table 1: HWPE-Stream signals.

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

1. **A valid handshake occurs in the cycle when both *valid* and *ready* are asserted**. There are no assumptions on what happens before or after, beyond those implied by the following rules.
2. **The *data* and *strb* can change their value 1) when *valid* is deasserted, 2) in the cycle after a valid handshake, even if *valid* remains asserted**. In other words, valid data must remain on the interface until a valid handshake has occurred.
3. **The assertion of *valid* (transition 0🡪1) cannot depend combinationally on the state of *ready*.** On the other hand, the assertion of *ready* (transition 0🡪1) can depend combinationally on the state of *valid*. This rule avoids deadlocks in ping-pong logic.
4. **The deassertion of *valid* (transition 1🡪0) can happen only in the cycle after a valid handshake**. In other words, valid data produced by a source must be consumed on the sink side before *valid* is deasserted.

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*.

## TCDM protocol

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 Tightly-Coupled Data Memory (*TCDM*) protocol, and it is the same as the one used by cores and DMAs operating on memories. It supports neither multiple outstanding transactions nor bursts, as HWPEs are assumed to be closely coupled to memories.

The TCDM protocol is used to connect a *master* to a *slave*. Table 2 reports the signals used by the TCDM protocol.

| **Signal** | **Size** | **Description** | **Direction** |
| --- | --- | --- | --- |
| *req* | 1 bit | Handshake request signal (1=asserted). | Master 🡪 Slave |
| *gnt* | 1 bit | Handshake grant signal (1=asserted). | Slave 🡪 Master |
| *add* | 32 bit | Word-aligned memory address. | Master 🡪 Slave |
| *wen* | 1 bit | Write enable signal (1=read, 0=write). | Master 🡪 Slave |
| *be* | 4 bit | Byte enable signal (1=valid byte). | Master 🡪 Slave |
| *data* | 32 bit | Data word to be stored. | Master 🡪 Slave |
| *r\_data* | 32 bit | Loaded data word. | Slave 🡪 Master |
| *r\_valid* | 1 bit | Valid loaded data word (1=asserted). | Slave 🡪 Master |

Table 2: TCDM protocol.

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. **The *r\_valid* signal 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🡪1) cannot depend combinationally on the state of *gnt*.** On the other hand, the assertion of *gnt* (transition 0🡪1) can depend combinationally on the state of *req* (and typically it does). This rule avoids deadlocks in ping-pong logic.

## Exchanging data between TCDM 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 the two protocols (i.e., exchanging data between the memory-based and the stream-based worlds) is one of the main tasks to be accomplished. The HWPE-Stream and TCDM protocols are similar by design, which makes the handling of handshakes easier.

Three objectives have to be met:

* HWPE-Stream has no notion of address: to produce a stream out of TCDM loads, or consume a stream in a series of TCDM 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.
* TCDM 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.

Figure 2: From TCDM loads to a HWPE-Stream

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:

* The ***hwpe\_stream\_addressgen*** module is responsible of generating addresses according to a pattern of 3D blocks characterized by width, height and depth.
* The ***hwpe\_stream\_merge*** and ***hwpe\_stream\_split*** modules can be used to merge/split HWPE-Stream streams. In this way, on the module boundary 32-bit streams can be converted in TCDM accesses.
* The ***hwpe\_stream\_source\_realign*** and ***hwpe\_stream\_sink\_realign***modules can be used to transform a strobed stream into unstrobed ones and to transform unstrobed streams into strobed ones. In this way, misaligned TCDM accesses can be already transformed in streams with a strobe to indicate what data is meaningful.

Figure 2 and Figure 3 show two examples of transactions going (respectively) from a series of loads on the TCDM interface to internal HWPE-Streams and from an internal HWPE-Stream to a series of stores on TCDM. The example focuses on the realignment behavior.

## PERIPH protocol

Figure 3: from a HWPE-Stream to TCDM stores

To enable control of HWPEs, they typically expose a slave port to the peripheral system interconnect. The slave port follows an extension of the TCDM protocol which we can call PERIPH. The PERIPH protocol is the same exposed by most peripherals in a PULP system and used by the core to communicate with them.

| **Signal** | **Size** | **Description** | **Direction** |
| --- | --- | --- | --- |
| *req* | 1 bit | Handshake request signal (1=asserted). | Master 🡪 Slave |
| *gnt* | 1 bit | Handshake grant signal (1=asserted). | Slave 🡪 Master |
| *add* | 32 bit | Word-aligned memory address. | Master 🡪 Slave |
| *wen* | 1 bit | Write enable signal (1=read, 0=write). | Master 🡪 Slave |
| *be* | 4 bit | Byte enable signal (1=valid byte). | Master 🡪 Slave |
| *data* | 32 bit | Data word to be stored. | Master 🡪 Slave |
| *id* | ID\_WIDTH bits | ID used to identify the master (request). | Master 🡪 Slave |
| *r\_data* | 32 bit | Loaded data word. | Slave 🡪 Master |
| *r\_valid* | 1 bit | Valid loaded data word (1=asserted). | Slave 🡪 Master |
| *r\_id* | ID\_WIDTH bits | ID used to identify the master (reply). | Slave 🡪 Master |

Table 3: PERIPH protocol.

The PERIPH protocol is distinguished by the TCDM protocol by the *id* and *r\_id* side channels. They 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.

# HWPE interface modules

## HWPE-Stream management modules

HWPE-Stream IPs management modules, found within the *hwpe-stream* Git repository, are used to connect multiple streams, delay them, merge / split them, or otherwise morph them. Stream splitter

### Stream merger

The ***hwpe\_stream\_merge*** module is used to merge NB\_IN\_STREAMS 32-bit 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 ANDof 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.

### Stream splitter

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.

### Static stream multiplexer

The ***hwpe\_stream\_mux\_static*** module is used to statically propagate one of two input streams of size DATA\_SIZE into a single output stream. The multiplexer is static as the selection bit *cannot be changed* when there are transactions in flight; if the selection bit is changed when transactions are in flight, the result is undefined.

### Static stream demultiplexer

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 *cannot be changed* when there are transactions in flight; if the selection bit is changed when transactions are in flight, the result is undefined.

### Stream FIFO

The ***hwpe\_stream\_fifo*** module implements a FIFO for HWPE-Stream streams of size DATA\_WIDTH and depth FIFO\_DEPTH. It is optionally implemented with latches if LATCH\_FIFO=1. The ***hwpe\_stream\_fifo\_earlystall*** variant lowers the ready signal one cycle before the FIFO is actually full.

### Stream Buffer

The ***hwpe\_stream\_buffer*** implements a shallow pipeline stage of size DATA\_WIDTH for cases where a full stream FIFO is not required, i.e. when the only important feature required is to cut forward propagation combinational paths.

### Source realigner

The ***hwpe\_stream\_source\_realign*** module is used to transform a strobed (misaligned) stream of size DATA\_WIDTH into a realigned stream of the same size, taking as input a strobe generated from an address generator (see below).

The module does not work for generic strobes, but rather it assumes that strobes result in a *rotation*, which is what happens for streams generated from a batch of misaligned transfers.

### Sink realigner

The ***hwpe\_stream\_sink\_realign*** module is used to transform a stream of size DATA\_WIDTH into a realigned strobed stream of the same size, taking as input a strobe generated from an address generator (see below).

The module does not work for generic strobes, but rather it assumes that strobes result in a *rotation*, which is what happens for streams used to generate from a batch of misaligned transfers.

## TCDM / HWPE-Stream interface modules

At the interface between the TCDM and HWPE-Stream modules, the main necessity is to generate an address for the streams. They also reside in the *hwpe-stream* repository.

### Address generator

The ***hwpe\_stream\_addressgen*** module is used to generate addresses to load or store HWPE-Stream streams. The REALIGN\_TYPE parameter is used to generate appropriate strobes to realign the streams in the sink and source cases.

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.

While useful in accelerators (e.g. in the HWCE [1][2][5]) the multiple loops are essentially supersed by the functionality provided by the microcode processor that can be embedded in HWPEs. The usage of more than a single loop is discouraged, i.e. the HWPE designer should statically set line\_stride=0, feat\_length=1, feat\_stride=0.

### Source

The ***hwpe\_stream\_source*** puts together an address generator, a stream merger, and a source realigner to create an interface between NB\_TCDM\_PORTS memory ports using the TCDM protocol (for loads alone) and a stream of size DATA\_WIDTH=NB\_TCDM\_PORTS\*32.

Typically it is sufficient to instantiate directly this module instead of the address generator, stream merger and source realigner alone.

### Sink

The ***hwpe\_stream\_sink*** puts together an address generator, a stream splitter, and a sink realigner to create an interface between a stream of size DATA\_WIDTH=NB\_TCDM\_PORTS\*32 and NB\_TCDM\_PORTS memory ports using the TCDM protocol (for store alone).

Typically it is sufficient to instantiate directly this module instead of the address generator, stream merger and sink realigner alone.

## TCDM management modules

Modules to manage TCDM streams with address also reside within the *hwpe-stream* repository.

### TCDM FIFO (loads)

The ***hwpe\_stream\_tcdm\_fifo\_load*** module can be used to decouple loads with two FIFOs (one for requests, one for responses). It is currently not fully tested.

### TCDM FIFO (stores)

The ***hwpe\_stream\_tcdm\_fifo\_store*** module can be used to decouple stores with a FIFO (for requests). It is currently not fully tested.

### TCDM dynamic multiplexer

The ***hwpe\_stream\_tcdm\_mux*** module can be used to dynamically share NB\_IN\_CHAN channels using the TCDM protocol into NB\_OUT\_CHAN channels, with NB\_OUT\_CHAN < NB\_IN\_CHAN. The multiplexer is not “optimal” in the sense that there is no reorder buffer, so transactions cannot be swapped in-flight. In practice this limitation is compensated by the fact that the cost of the reorder buffer is saved, and it works well in practice in the Fulmine HWCE [1].

### TCDM static multiplexer

The ***hwpe\_stream\_tcdm\_mux\_static*** module is used to statically share NB\_CHAN ports using the TCDM protocol between two sets of NB\_CHAN input ports. It works similarly to the ***hwpe\_stream\_mux\_static*** and similarly requires a strictly static selector.

### TCDM reorder block

The ***hwpe\_stream\_tcdm\_reorder*** module is used to shuffle the order of NB\_CHAN channels using the TCDM protocol according to an external order, that can be changed arbitrarily (e.g. with a counter). This is useful in some cases (e.g. [1]) so that the probability of a transaction is equalized between multiple ports.

## PERIPH and controller modules

The control interface of HWPEs exposes a PERIPH interface that is used to program a memory-mapped register file. The *hwpe-ctrl* repository contains several IPs that can be used to compose the control interface; apart from the PERIPH interface, these modules are optional – and the main control finite-state machines are accelerator-specific and have to be designed from scratch in any case.

### Microcode processor

The ***hwpe\_ctrl\_ucode*** module is a microcode processor that can be used to execute the main computation block of an HWPE (implemented within the “engine”) multiple times according to several rules, at the same time adapting the value of several internal parameters. The microcode processor can be used to execute a default number of 6 nested loops.

The microcode supports four R/W registers and twelve R/O registers (by default); the microcode has two instructions: an **add** operation and a **move** operation. The **add** operation performs RA := RA + RB; the **move** operation performs RA := RB. R/O registers can only be used as RB. The R/W registers can be used to generate offsets to program the address generators, or for other purposes.

The microcode can be specified in a “high-level” fashion in terms of YAML description, which can then be “compiled” by the *ucode\_compile.py* Python script, also within the *hwpe-ctrl* repository. The compiler provides the two bit fields to be used to program the HWPE microcode processor, typically this is either hardwired or passed through job-independent registers.

### Slave interface and register file

The ***hwpe\_ctrl\_slave*** module implements the PERIPH slave interface. The ***hwpe\_ctrl\_regfile***, which is instantiated inside it, implements the actual register file. The register file contains N\_GENERIC\_REGS registers which are non-contexted, i.e. their value stays constant between consecutive job offloads; and N\_IO\_REGS registers which are contexted, i.e. which are used to implement a queue of jobs that can be offloaded also when the HWPE is active. The slave module also generates the events that are propagated in the PULP platform.

### Sequential multiplier

The ***hwpe\_ctrl\_seq\_mult*** module is a utility module to implement a sequential multiplier; it can be used to produce derivative parameters e.g. for usage as read-only registers in the microcode processor. When the *start* input is asserted, the multiplier will start compute the product of the two inputs *a* and *b*. The sequential multiplier takes *width(a)* cycles to compute the output and asserts a valid bit when the product has been computed.

# Register file map

The HWPE configuration ports are visible in the cluster memory map. Typically they are divided between *control registers*, which are the same in all HWPEs; *generic registers* or *job-independent*, which are used to hold parameters that are assumed not to change within consecutive offloads; and *job-dependent registers*, which are used to hold parameters that change between consecutive offloads. The latter kind of registers can be modified even when the HWPE is executing, because the HWPE supports at least two contexts, or equivalently a FIFO queue of at least two jobs.

The rest of this section does not deal with *generic* and *job-dependent* registers, as their semantic varies from a HWPE to another.

## Control registers and generic configuration registers

The control registers are not replicated between different jobs and are used to start and finish a job offload sequence, check the status of the HWPE and soft-clear the HWPE.

| **Reg. #** | **Name** | **R/W** | **Description** |
| --- | --- | --- | --- |
| 0x0 | *TRIGGER* | W/O | Trigger the execution of an offloaded job. |
| 0x1 | *ACQUIRE* | R/O | Acquire the lock to offload a job. |
| 0x2 | *FINISHED\_JOBS* | R/O | Returns the number of concluded jobs since last read. |
| 0x3 | *STATUS* | R/O | Returns the status of the HWPE (1=busy, 0=idle). |
| 0x4 | *RUNNING\_JOB* | R/O | Returns the ID of the currently running job. |
| 0x5 | *SOFT\_CLEAR* | W/O | Resets the HWPE to its known idle state. |
| 0x6-7 | - | - | Reserved. |
| 0x8-F | *HWPE dependent* | R/W | Depends on actual HWPE. |

Table 4: Control registers and generic configuration registers.

### TRIGGER register

Write-only register; any write to this register will close the current offload phase by releasing the job offload lock and inserting the currently offloaded job in the control queue.

### ACQUIRE register

Read-only register; any read to this register has the “side effect” of initiating an offload sequence by acquiring the job offload lock. Until the offloading core releases the lock by writing to the TRIGGER register, no other core can start a job offload.

A read to the ACQUIRE register can return:

* the *id* of the job to be offloaded (a number from **0** to **255**)
* an *error* code if one of the following conditions apply:
  1. else, if the job offload lock has been established, it will answer **0xfffffffe** (**-2**)
  2. else, if the job queue is full, it will answer **0xffffffff** (**-1**)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| **ERR** | | | | | | | | | | | | | | | |
| R/O | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **ERR** | | | | | | | | **ID / ERR** | | | | | | | |
| R/O | | | | | | | | R/O | | | | | | | |

| **Bit #** | **R/W** | **Description** |
| --- | --- | --- |
| 31:8 | R/O | **ERR**: error code. |
| 7:0 | R/O | **ID / ERR:** if ERR is nil, ID of the offloaded job. Otherwise, part of error code. |

Table 5: ACQUIRE register bit fields.

### FINISHED\_JOBS register

Read-only register containing an abstraction of the number of jobs that the HWPE executed and finished since the last time the same *FINISHED\_JOBS* register was accessed. A read to *FINISHED\_JOBS* returns:

* **0x0** if no job was completed since the last access;
* **0x1** if a single job was completed since the last access;
* **0x2** if two or more jobs were completed since the last access.

### STATUS register

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | | 16 |
| *Reserved* | | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | | 0 |
| *Reserved* | | | | | | | | | | | | | | | **ST** | |
| R/O | |

| **Bit #** | **R/W** | **Description** |
| --- | --- | --- |
| 31:1 | - | *Reserved.* |
| 0 | R/O | **ST:** 1 if the HWPE is currently running a job. |

Table 6: STATUS register bit fields.

### RUNNING\_JOB register

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| *Reserved* | | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| *Reserved* | | | | | | | | **ID** | | | | | | | | |
| R/O | | | | | | | | |

| **Bit #** | **R/W** | **Description** |
| --- | --- | --- |
| 31:8 | - | *Reserved.* |
| 7:0 | R/O | **ID:** the ID of the job currently running. |

Table 7: RUNNING\_JOB register bit fields.

### SOFT\_CLEAR register

Write-only register; any write to this register will reset the HWPE to its idle state and clear all internal flip-flops (but not the standard-cell latch-based memories

# Microcode specification and processor

The microcode processor provides a general way to specify relatively complex loop patterns without hard-coding them, and to keep the actual code of the loop modifiable post design time. The microcode is used to execute nested loops such as this example (in pseudo-Python code):

for i in range(0, NI):  
 c = c+1  
 d = a  
 b = b\_init  
 for j in range(0, NJ):  
 b = b+1  
 for k in range(0, NK):  
 a = a+1

The loops must be static (i.e. they must have a known number of iterations).

## Microcode ISA

The microcode processor uses a tiny ISA that supports two types of registers:

| **Reg #** | **R/W** | **Description** |
| --- | --- | --- |
| 0-3 | R/W | Used to store offsets to be used to compute new address generation bases. |
| 4-15 | R/O | Used to store parameters to compute new values for reg. 0-3. |

Table 8: microcode ISA registers

The microcode ISA consists in two imperative instructions, similar to an extremely tiny subset of a RISC ISA, and one declarative instruction.

Imperative instructions are R-R operations. They have two operands, *RA* and *RB*. *RA* is always a register of type R/W; *RB* can be any register.

| **Instruction** | **Operation** | **Description** |
| --- | --- | --- |
| **ADD** | *RA* **←** *RA + RB* | Replaces *RA* with the sum of *RA* and *RB*. |
| **MV** | *RA* **←** *RB* | Replaces *RA* with *RB*. |

Table 9: microcode ISA imperative instructions

The two operations are encoded using a single bit to select between the two operations, 5 bits to select the *RA* address, and 5 bits to select the *RB* address.

The microcode also supports a single declarative **LOOP** instruction. The amount of loops supported by the microcode processor is a parameter (*NB\_LOOPS*, default is 6), and at the beginning of the microcode there are as many **LOOP** instructions as are supported, even if only one is actually used.

| **Field** | **Size** | **Description** |
| --- | --- | --- |
| **BASE** | 5 bits | Address in the microcode of the first imperative instruction of this loop. |
| **NB.OPS** | 3 bits | Number of operations included in the loop declarative instruction. |
| **RANGE** | *CNT* bits | Number of iterations in the loop (see details below). |

Table 10: fields of **LOOP** declarative instruction

For details on how the instructions are serialized and encoded, see *hwpe\_ctrl\_package.sv*. The **RANGE** field is special: typically it is not set as part of the microcode (which might be in a generic, job-independent register); instead, its value is computed from parameters in job-dependent registers.

## Microcode FSMs

Internally, the microcode processor is constituted of a set of finite-state machines that collectively implement the behavior specified in the previous section. The current state of the microcode processor is defined by four registers, described in Table 11.

| **Register** | **Size** | **Description** |
| --- | --- | --- |
| **loop** | clog2(*NB\_LOOPS*) bits | Index of the currently considered nested loop. |
| **op** | 3 bits | Index of the currently executing operation within the loop. |
| **addr** | clog2(*LENGTH*) bits | Address of the operation to execute in the imperative microcode. |
| **idx[l]** | *NB\_LOOPS* × *CNT* bits | Vector of iteration indeces. |

Table 11: state registers of microcode processor

Each register (and in the case of **idx**, each element) is conceptually managed by a separate finite-state machine[[1]](#footnote-1). All these FSMs share transitions triggered by the same conditions, which result in coherent actions. Some of these transitions trigger the execution of a micro-instruction in the following cycle, which updates the value of a register as discussed in the previous section. Transitions are reported in Table 12.

| **Transition** | **Conditions** | **Exec?** |
| --- | --- | --- |
| UPDATE | **idx**[**loop**] < decl\_instr[**loop**].range-1 **&&** **op** < decl\_instr[**loop**].nb\_ops-1 | Y |
| ITERATE & GOTO 0 | **!** UPDATE **&& idx**[**loop**] < decl\_instr[**loop**].range-1 **&& loop**  > 0 | Y |
| ITERATE | **!** UPDATE **&&** **!** ITERATE & GOTO 0 **&& idx**[**loop**] < decl\_instr[**loop**].range-1 | Y |
| GOTO NEXT | **!** UPDATE **&& !** ITERATE & GOTO 0 **&& !** ITERATE **&& loop** < *NB\_LOOPS* | N |
| TERMINATION | **else** | N |

Table 12: transitions of the microcode processor FSMs

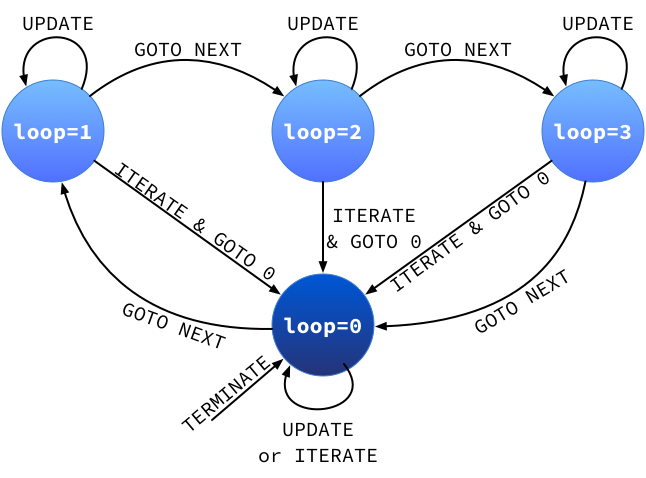


Figure 4: **loop** FSM

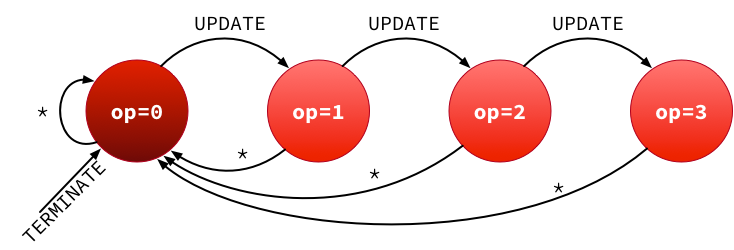


Figure 5: **op** FSM

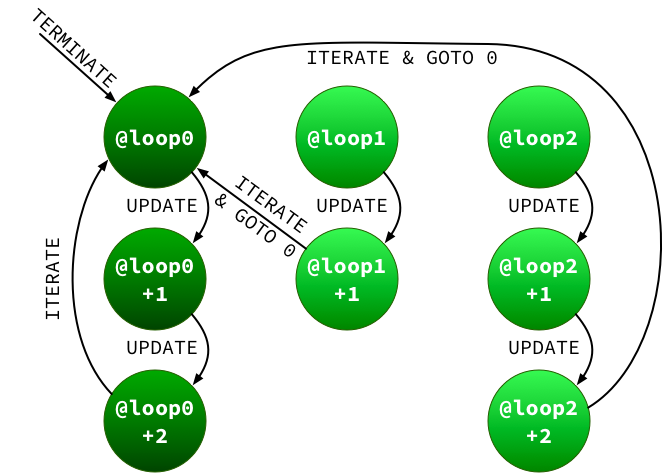


Figure 6: **addr** FSM

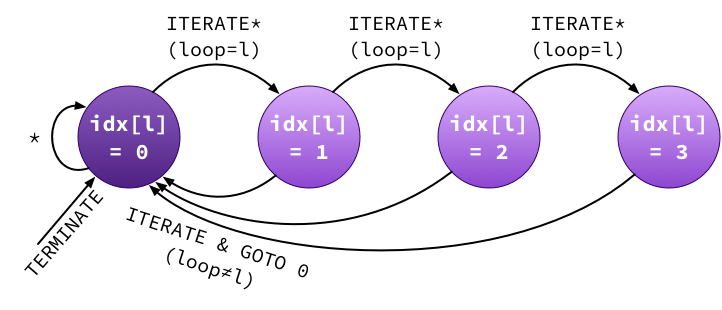


Figure 7: **idx[l]** FSM

# Bibliography

[1] F. Conti et al., “**An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics**,” IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 64, no. 9, pp. 2481–2494, Sep. 2017.

[2] 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, San Jose, CA, USA, 2015, pp. 683–688.

[3] F. Conti, A. Marongiu, and L. Benini, “**Synthesis-friendly techniques for tightly-coupled integration of hardware accelerators into shared-memory multi-core clusters**,” in 2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013, pp. 1–10.

[4] F. Conti, A. Marongiu, C. Pilkington, and L. Benini, “**He-P2012: Performance and Energy Exploration of Architecturally Heterogeneous Many-Cores**,” Journal of Signal Processing Systems, pp. 1–16, 2015.

[5] 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 Computing**,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. PP, no. 99, pp. 1–1, 2017.

1. This statement is referred to the conceptual view, which is explained below. The current version of the RTL code distributed in the *hwpe\_ctrl* repository uses only two processes – one always\_comb and one always\_ff – so technically it might be considered a single state machine, whose state is composed by all the states in Table 11 and whose diagram is considerably more complex. [↑](#footnote-ref-1)