# P2012 / STHORM
# Shared Memory HWPE Wrapper
Internship Final Report

Author
Francesco Conti

## Table of Contents

# 1. Introduction

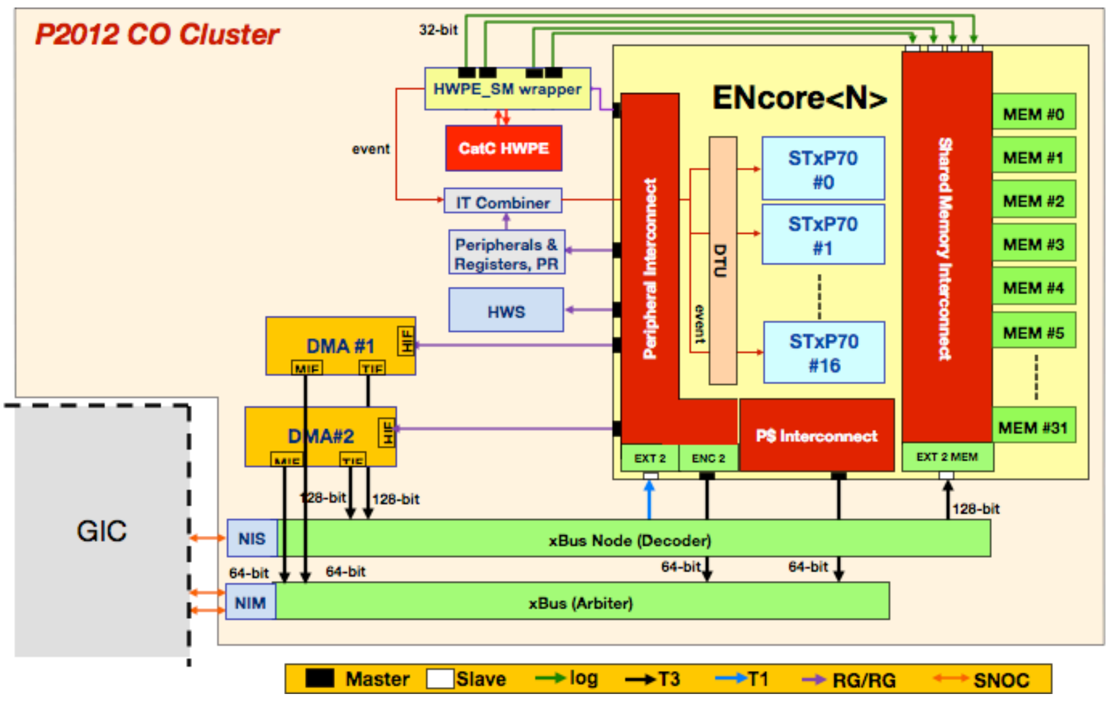

*Figure 1: Shared memory HWPE in P2012 v0.2.2 cluster*

The **Shared Memory HWPE Wrapper** module provides a standard interface to a v0.2.2 P2012 cluster for shared memory accelerators, whose peculiarity is that they can directly access the same TCDM of the PEs in the cluster, therefore cutting down communication overhead. On the ENcore side it uses a T2-like interface compatible with the logarithmic interconnect, while on the accelerator side it uses a custom memory-like interface. At the same time it implements a set of registers accessible through the peripheral interconnect, which can be used to offload jobs to the HWPE. Job offload while the HWPE is working is supported through a multi-context mechanism. A diagram of the v0.2.2 cluster with the addition of the shared memory HWPE is visible in figure 1.

The Shared Memory HWPE Wrapper is a fully parametric IP, and by tweaking parameters greatly differing configurations can be reached varying in terms of number of ports, contexts, registers etc. The interface with the accelerator has been thought to deal with accelerators developed through the CatapultC HLS tool, but nothing forbids to use it with other tools or with hand-coded accelerators.

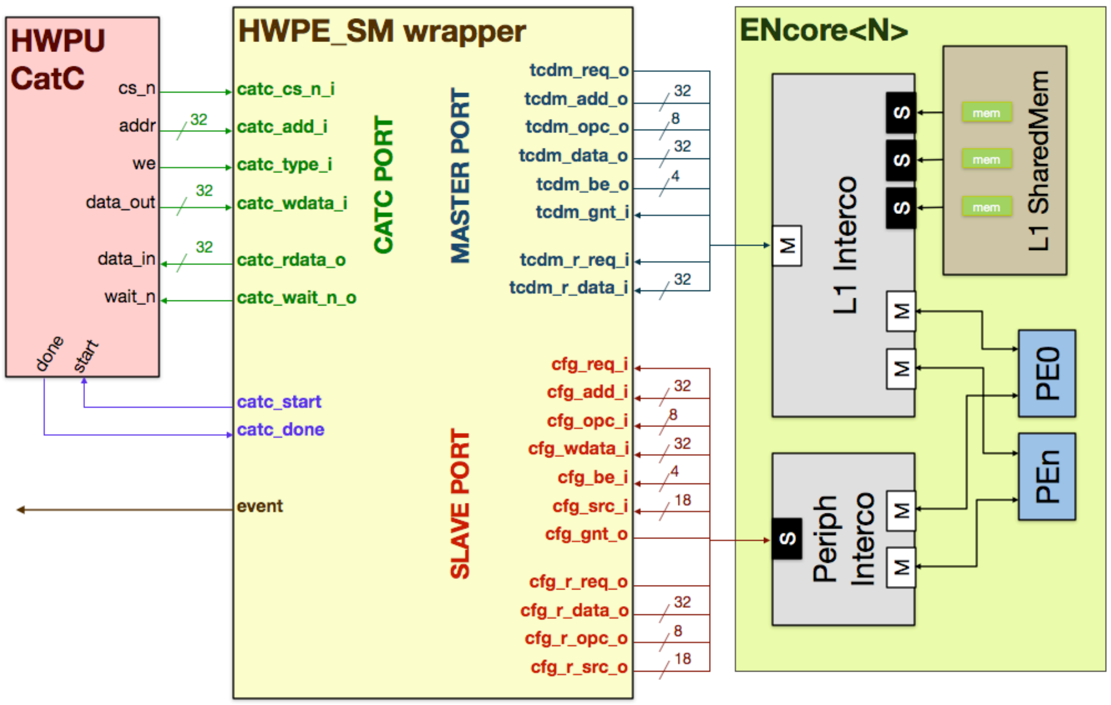

*Figure 2: Wrapper overview*

An overview of the wrapper in a simple configuration is shown in figure 2. Three ports are shown: one accelerator port, one log-interconnect port (also called "master port"), one configuration port. The top module of the Shared Memory HWPE Wrapper is the VHDL **hwpe_sm_wrapper** module, which contains some glue logic and two macroblocks: the **hwpe_sm_slave_mc** module, which implements the register file, and the **hwpe_sm_controller**, which includes the datapath transforming the accelerator requests into valid requests for the logarithmic interconnect and driving back the responses and its control logic.

The configuration port (or slave port) is a target port for the ENcore peripheral interconnect, with which it communicates via a T2-compatible protocol whose handshake follows a request & grant scheme. This port is used by the rest of the cluster to offload jobs to the HWPE by writing into its registers.

The accelerator port is used by the accelerator to access the TCDM. It provides the accelerator with the abstraction of a private memory of unknown latency by using a custom protocol modeled on the Catapult *ram_w_handshake* protocol. This protocol performs the handshake through a single wait signal which, when asserted, invalidates the current response and forces the accelerator to re-issue the request. Requests from the accelerator port are transformed in a compatible form inside the wrapper and issued to the logarithmic interconnect via the master port.

The wrapper currently supports configurations with 1, 2 or 4 master ports and 1, 2, 4 or 8 accelerator ports; the number of ports on accelerator side must be greater than or equal to the number of master ports. Table 1 shows the parameters that can be used to change the number of ports.

| Parameter | Type | Values | Description |
|---|---|---|---|
| N_MASTER_PORT | natural | 1,2,4 | Number of ports on the ENcore side |
| N_ACCELERATOR_PORT | natural | 1,2,4,8 | Number of ports on the accelerator side |

*Table 1: General wrapper parameters*

To start the accelerator, the wrapper asserts the starts signals, then it waits on the done signal to understand when the accelerator has finished execution. The done signal is registered and then broadcasted as an event signal to the interrupt combiner, so that the PEs can acknowledge the end of an offloaded job.

In the following sections both the controller and the slave module will be described in detail, along with their submodules and the parameters that can be used to configure them.

## 2. Slave module

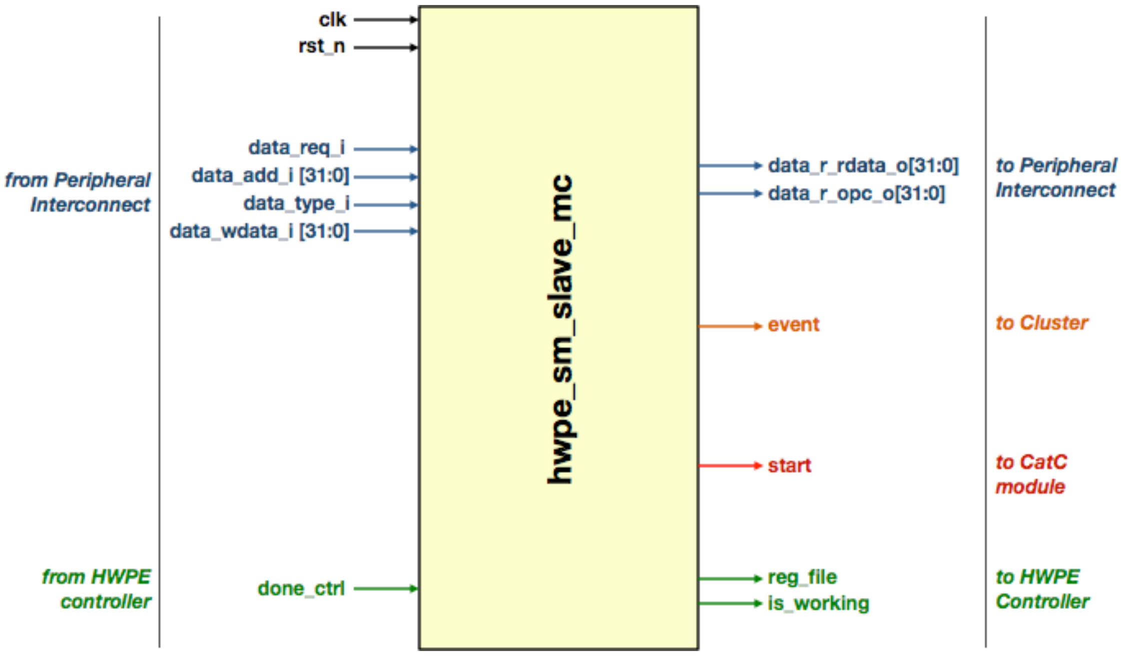

*Figure 3: Slave pinout*

The slave module implements the multi-context register file and the logic necessary to start the accelerator when needed. The multi-context functionality is aimed at allowing the offload of a new job while a current job is in execution, i.e. to perform a kind of double buffering for job offloading. After offload, contexts are executed in a FIFO queue fashion. To avoid conflicts, no job offloads are accepted while the context queue is full or when another offload has not yet ended.

The registers in the slave module are divided in two groups: **mandatory** registers, which are used for general control of the HWPE, and **contexted** registers, which contain parameters relative to a specific job offload procedure. Registers 0 to 7 are mandatory and can be accessed both in read and write through address from 0x000 to 0x01c. Contexted registers are shadowed in write, meaning that during a normal offload parameters are always written to addresses 0x020 to 0x07c; the slave then directs the write to the physical registers of the appropriate context (the one indicated by the context pointer), while the other registers are unaccessible in write. However, for debugging purposes, it is possible to access all the registers from all contexts in read; context #0 is associated to the same set of addresses as in write, while the other contexts have a mirrored address space which is obtained by adding context_id*32*4 to the normal address. Mandatory registers are not mirrored and a read to addresses in range (context_id*32+[0-7])*4 will result in an error code. Table summarizes the names, addresses and content of the registers.

| Register | Address | Type | Name | Description |
|---|---|---|---|---|
| 0 | 0x000 | W/O | Trigger | Ends the offload sequence when is set to **1** |
| 1 | 0x004 | R/O | Context | Returns **-1** if there are no free contexts; **-2** if the offload is locked by another PE; else initiates an offload sequence and returns context id as a handle |
| 2 | 0x008 | R/W | Sync mode | Must be set to **1** to activate *event* signal generation |
| 3 | 0x00c | R/O | Status | Byte *i* contains the status of context #*i*, **1** if offloaded and not completed, **0** else |
| 4 | 0x010 | R/O | Running context | Contains the id of the currently running context |
| 5 | 0x014 | R/O | Pointer context | Contains the id of the next granted context id, without initiating an offload |
| 6 | 0x018 | R/O | Context owner | Byte *i* contains the id+1 of the PE who last offloaded context #*i* |
| 7 | 0x01c | - | Unused | - |
| 8 - 15 | 0x020 - 0x03c | R/W | Generic registers | N_MAX_GENERIC_REGS registers to contain general purpose parameters for context #0 |
| 16 - 31 | 0x040 - 0x07c | R/W | Input/output registers | N_MAX_IO_REGS registers to contain addresses of input/output data in TCDM for context #0 |
| 32 - 63 | 0x080 - 0x0fc | - | Context #1 | Address space for context #1 |
| 64 - 95 | 0x100 - 0x17c | - | Context #2 | Address space for context #2 |
| 96 - 127 | 0x180 - 0x1fc | - | Context #3 | Address space for context #3 |

*Table 2: Register file address map*

The job offload sequence proceeds in the following way:

1. the offloading PE reads the context register; if it gets -1 or -2 the offload procedure can't continue and the PE must either poll the context register or go in idle and wait for an event to be sent.

2. when the PE reads a valid *context_id*, it can write the context registers if necessary.
3. if necessary, the PE can save the *context_id* and use it as a handle to read the status and context owner registers or one of the contexted registers.
4. after all the parameters are correctly set, the PE writes 1 to the trigger register.
5. after previous jobs have been executed, the slave starts the job with id *context_id*.
6. when execution has finished, the slave notifies the PE with an event.
7. if necessary, the PE can check if the event was related to its job by reading the *context_id* byte of the context owner register.

It is worth to be noted that, since the event signal is only sent after the completion of a job, if a PE goes to idle after reading -2 from the context register some time might be lost (because -2 only means that some other PE was offloading a job, not that the context queue was full). A possible software way to avoid this is to poll on the context register when the PE gets -2, since the offload sequence is generally short, and go to idle only when it gets -1.

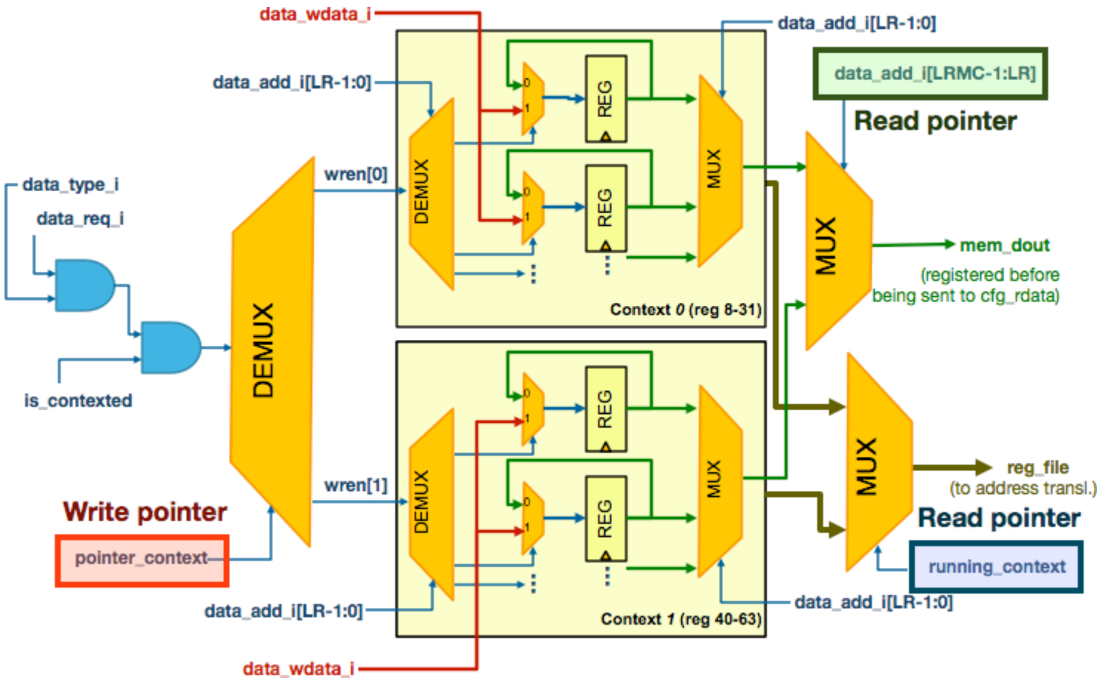

*Figure 4: Register file: contexted registers microarchitecture*

The RTL code is structured in such a way unused registers are not physically implemented, but at the same time same exact address configuration is mantained. By changing some parameters, the configuration can be tailored to actual needs. Figure 4 shows the microarchitecture of the contexted registers, implementing a FIFO of N_CONTEXT sets of registers with a single write pointer (the pointer context) and two read pointers: the part of the address specifying the context id for data going to the peripheral interconnect and the running context for use inside the wrapper. For write and reads through the peripheral interconnect, the lower log2(N_REGISTERS) bits are used to select the correct register inside the context, while for use inside the wrapper (more specifically inside the address translation module) the whole context is passed to the reg_file output of the slave module.

The rest of the microarchitecture consists mainly in:

- two counters with enable, *pointer_counter* and *running_counter*;
- a bidirectional counter, *pending_counter*, to evaluate the number of pending jobs;
- two FSMs, *fsm_running* and *fsm_context*, to control respectively the start procedure and the lock of the offload procedure;
- the mandatory R/W and R/O registers
- logic to generate the values of R/O mandatory registers

Table 3 reports the parameters which may be used to modify the structure of the slave module and the register file. Note that the minimal configuration with just one context is not supported at the moment, but will be in a future version of the wrapper.

| Parameter | Type | Values | Description |
|---|---|---|---|
| N_CONTEXT | natural | 2,4 | Number of ports on the ENcore side |
| HW_CONTEXT_SYNC | boolean | T,F | If False, there is no HW lock on the offload sequence: PEs have to synchronize in software |
| N_REGISTERS | natural | 32 | Number of register addresses reserved for a context |
| N_MANDATORY_REGS | natural | 7 | Number of mandatory registers |
| N_MAX_GENERIC_REGS | natural | 8 | Number of register addresses reserved for generic parameters in a context |
| N_MAX_IO_REGS | natural | 16 | Number of register addresses reserved for input/output parameters in a context |
| N_GENERIC_REGS | natural | 0-8 | Actual number of generic parameters in a context |
| N_IO_REGS | natural | 0-16 | Actual number of input/output parameters in a context |

*Table 3: Slave parameters*

## 3. Controller module

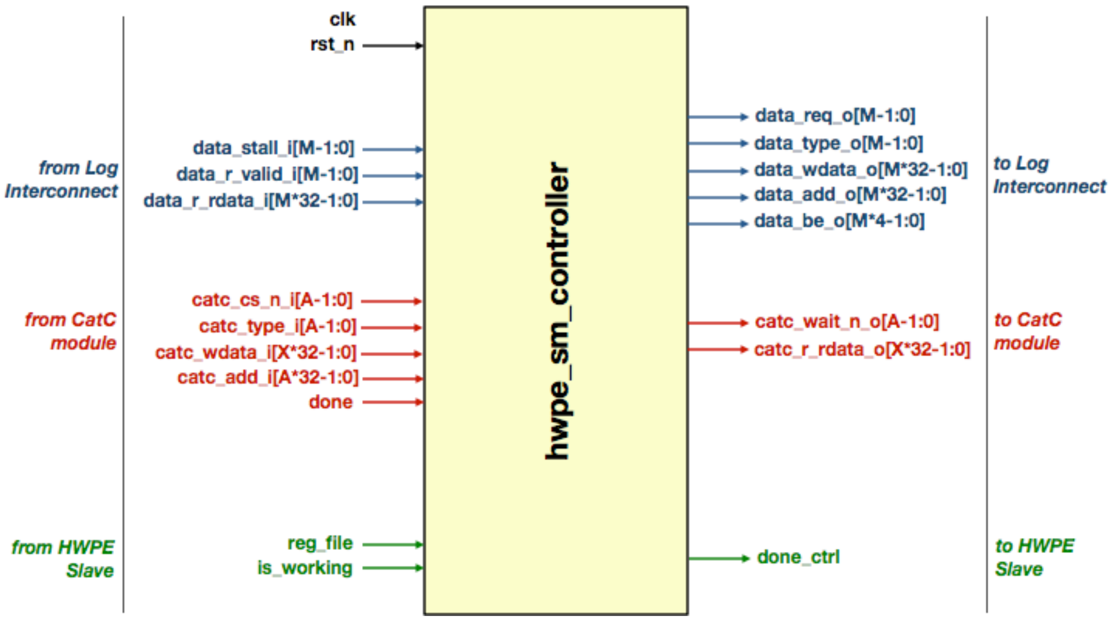

*Figure 5: Controller pinout*

The objective of the controller module, whose pinout is shown in figure 5, is to translate the requests from the accelerator into valid logarithmic interconnect transactions, in which the HWPE takes the role of the master. The first design guideline was of course to ensure correctness from the accelerator's point of view, that is to grant that every valid request is validly responded. The second guideline, given the tightly-coupled nature of the shared memory HWPE wrapper, was to provide the maximum performance in terms of throughput, at least in the common case.

Since the first accelerators targeted for shared memory HWPE are synthesized through CatapultC, the design of the controller has been focused on providing correctness in its case. Due to some inherent limits of the *ram_w_handshake* protocol and of CatapultC, some workarounds - which can be activated or not depending on the value of parameters - have been inserted into the controller. These workarounds are currently the only way to ensure correct operation (particularly in case of memory stall); an initial collaboration with Calypto to have a real fix to the related bugs has been set up.

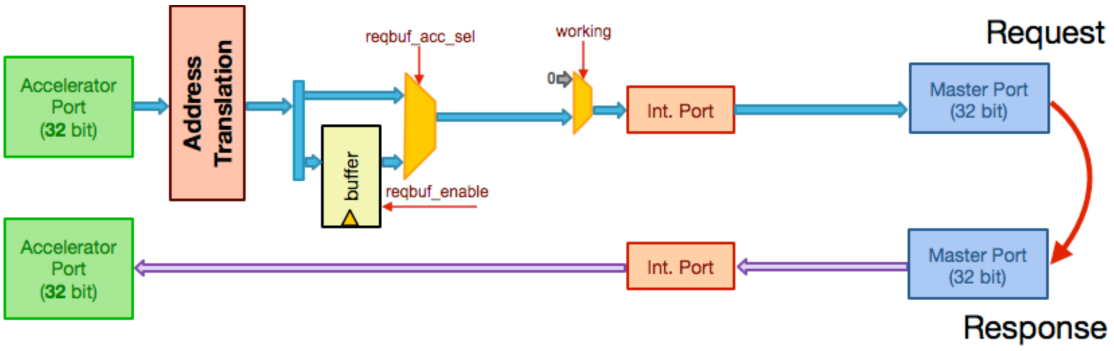

*Figure 6: Controller datapath in simple configuration*

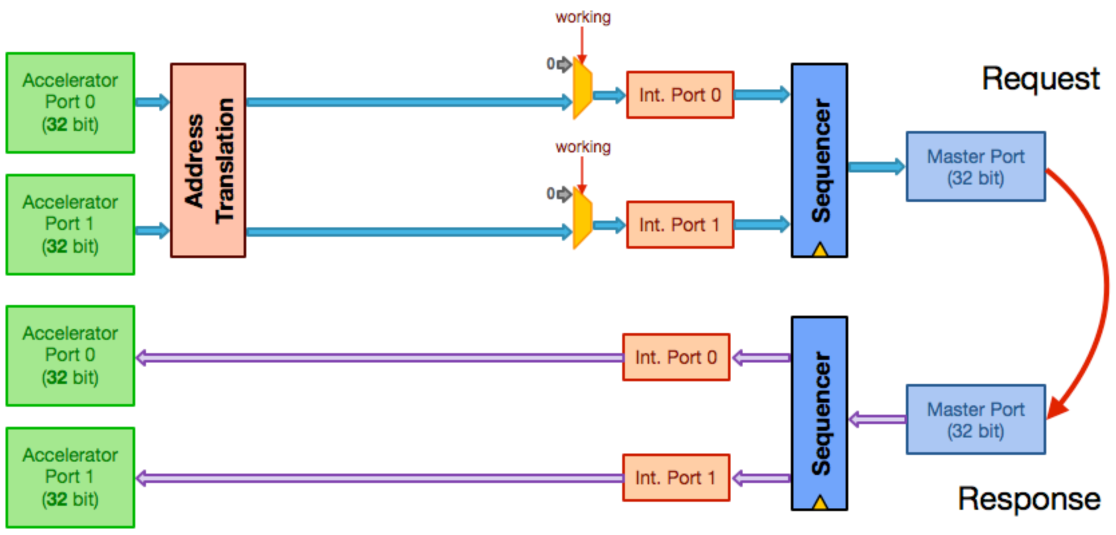

*Figure 7: Controller datapath in sequenced configuration*

The controller module is basically made by two datapaths, the request one and the response one, plus the control logic needed to drive them. The request datapath is composed by some basic components that can be configured to make up different configurations at design time, two of which are shown in figure . These basic components are the **address translation** module, a set of buffers and the **sequencer** module; they can be composed to form two different configurations of the controller.

The **simple** configuration is that in which N_MASTER_PORT is equal to N_ACCELERATOR_PORT, and therefore there is a 1-1 correspondance between master and accelerator ports. In this case the sequencer is left out and the datapath is composed by the address translation module, a bypassable set of buffers, and a set of multiplexers which can be used to "silence" all request traffic in case the accelerator is not working (sometimes spurious requests are generated when the module is reset). The response datapath, except from the stall signal, is nothing more than a direct connection of TCDM signals back to the accelerator.

On the contrary, in the **sequenced** configuration N_ACCELERATOR_PORT is greater than N_MASTER_PORT and therefore multiple accelerator ports are associated to a single master port. In this case, the sequencer module is used to time-multiplex the requests to the TCDM. Since the sequencer performs buffering when necessary internally, the set of buffers present in the simple configuration is not needed. In this case the response datapath also includes the sequencer, which assigns the response to the right accelerator port depending on its previous request and the stall conditions.

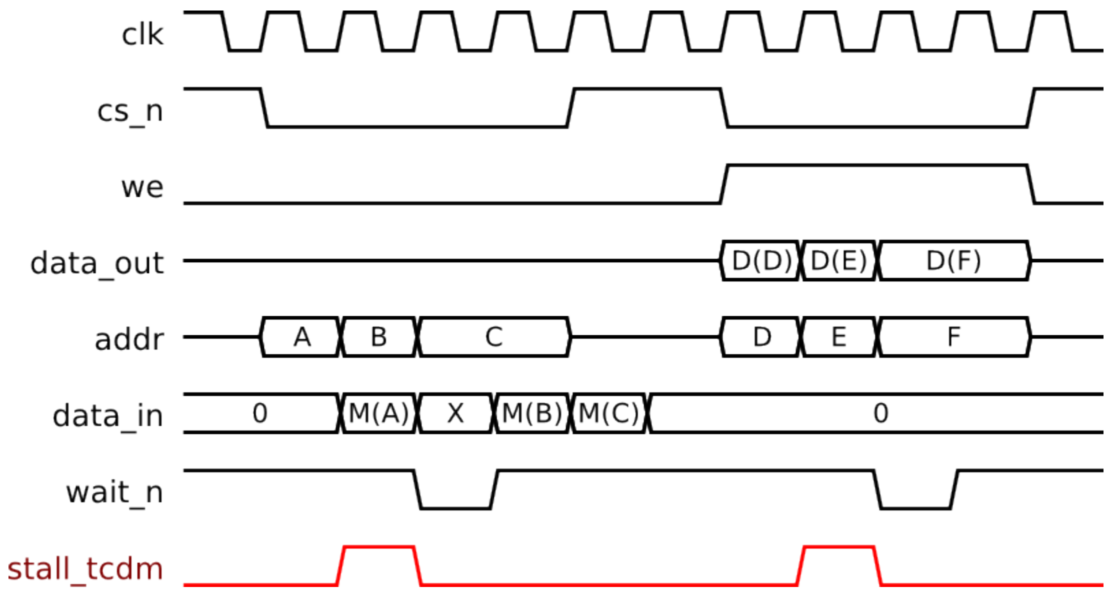

*Figure 8: ram_w_handshake protocol*

Figure 8 illustrates the *ram_w_handshake* protocol that is used by the accelerators generated through Calypto CatapultC. The accelerator uses a cs_n chip select to signal its request and a we to indicate whether the operation is a read or a write. The handshake is performed through a wait_n signal, which orders the accelerator to repeat its current request in the next cycle and invalidates the current datum in data_out; this is different from the stall signal coming from the TCDM which also orders to repeat the current request, but invalidates *next cycle's* datum. In the simple configuration, there are two possible solutions for this inconsistency:

- buffer the read datum and feed data_in with the buffered value in the cycle after a stall, feed wait_n with the unbuffered (negated) stall coming from TCDM;
- buffer the request (cs_n, addr, we, data_out) and feed the signals going to the TCDM with the buffered request in the cycle after a read stall, always feed the unbuffered read datum and always buffer wait_n.

The second solution, which is the one indicated in figure 8, was the one chosen. The main rationale behind the choice is that, while both the solutions grant the correctness of the transaction in non-pathological conditions, the second is more easily integrated with the sequencer module for the sequenced configuration, and it also has the advantage that it cuts one of the only signal chains (since the stall signal is generated combinatorially in the logarithmic interconnect). Since the sequencer module relies on heavy use of the stall signal for its inner functioning, in the sequenced configuration the wait_n signal is generated internally in the sequencer.

Unfortunately, there are many limitations and pathological conditions that can affect this otherwise simple protocol when CatapultC is used to generate an accelerator using it:

1. CatapultC does not always treat separated ports as independent. For example, stalling one of the ports related to the same CatapultC

memory resource results in all of them waiting, which causes a loss of performance.

2. Deeply related to the previous point is the case two different ports related to the same resource suffer a single-cycle stall in consecutive cycles: the second stall is not correctly acknowledged by the CatapultC module. The buffering technique explained in the previous paragraph ensures correctness of the transaction even in this case, but if the consecutive stalls and the ports involved are three or more instead of two (not an uncommon situation), correctness is lost.
3. A pathological condition which can arise is the stall of the last write of a sequence. When no cs_n signal is active, the accelerator disregards the wait_n signal; in this case the stalled request is concluded normally, but the following one is lost.
4. Sometimes the CatapultC module begins buffering input data internally, with no clear reason. This of course invalidates all that is done outside the module to mantain correctness; the buffering has to be directly eliminated inside the accelerators (see the Accelerator synthesis section).

In addition, due to the very nature of the protocol it is impossible to pipeline it, because it is impossible to make the accelerator issue new requests while waiting the response for a previous one. This is an intrinsic limitation that will have to be addressed in the future.

Issues 2 an 3 can be workarounded with the following methods, which are currently active by default:

2. By using the OR of all stall signals instead of the signals themselves problem 2 is bypassed, because multiple consecutive stalls on different ports become a single longer stall on all ports (which is a correctly working condition).
3. When the last write of a sequence is issued, the controller enters a special mode, the accelerator is disabled and it is re-enabled only when all the requests issued in the last cycle have valid responses. Though many simpler solutions were tried, only this one could work in any condition without causing other issues.

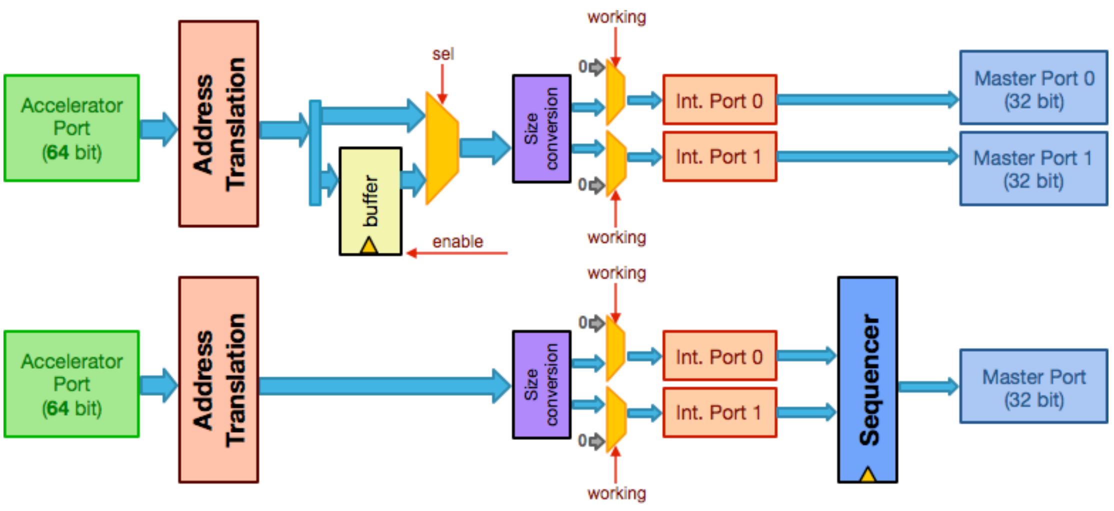

*Figure 9: Controller request datapath with size conversion*

The controller has also one unsupported parameter to use a **size converter**, a simple module that transforms a single accelerator ports of data width multiple of 32 bits into a set of 32-bit intermediate ports in the request and packs it back in the response datapath. Figure 9 shows the request datapath in simple and sequenced configuration with size conversion.

Table shows the parameters which can be used to modify the configuration of the controller block. Non-primitive parameters are also included for additional clarity; they are signaled with * sign in the "Values" column.

| Parameter | Type | Values | Description |
|---|---|---|---|
| ORG_ACCELERATOR_PORT | array of natural | any | A vector of N_ACCELERATOR_PORT elements containing the size of each accelerator port divided by 32 |
| N_INTERMEDIATE_PORT | natural | * | Number of intermediate ports (the sum of all values in ORG_ACCELERATOR_PORT) |
| PORT_SEQUENCER | boolean | * | If True the controller is in sequenced configuration, if False in simple configuration (it is True when N_INTERMEDIATE_PORT is greater than N_MASTER_PORT) |
| STALL_OR_FLAG | boolean | T/F | If True, activates the workaround to issue 2 by OR'ing all stall signals |
| LAST_WRITE_FLAG | boolean | T/F | If True, activates the workaround to issue 3 by adding a FSM that disables the accelerator in the last cycle if there are pending writes |
| SIZE_CONVERSION | boolean | T/F | If True, activates size conversion (currently unsupported) |

*Table 4: Controller parameters*

## 4. Address translation module

In the tightly coupled shared memory acceleration model the accelerator accesses vectors in the TCDM without any particular requirement in terms of structure and positioning inside the shared memory. Since on the contrary the positioning of vectors in memory must be known at compile time in CatapultC (there is no support for pointer arithmetics), an address translation stage is needed to remap the internal addresses known by Catapult to the real ones passed through the input/output registers. By using the right CatapultC directives, it is possible to map I/O arrays such that they occupy base-aligned addresses, that is such that each vector is identified by a unique base value in the upper part of its address.

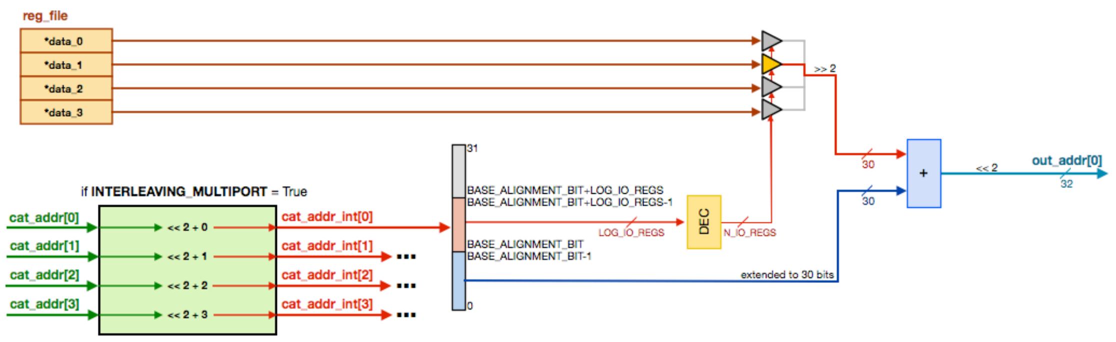

*Figure 10: Address translator module*

Figure 10 shows an example of address translation for a single port. If a group of N ports have been generated in CatapultC using the INTERLEAVING directive (see the Accelerator synthesis section), it is necessary to shift their input addresses by log2(N) bits to the left and add the number of the port in the group in the log2(N) least significant bits, generating the internal address. Otherwise, this preliminary shift is not performed. In any case, the internal address is split in two vectors, one containing the lowest BASE_ALIGNMENT_BIT bits, which are the offset, and the other containing the LOG_IO_REGS bits immediately higher than the offset, which indicate the base. Bits from BASE_ALIGNMENT_BIT+LOG_IO_REGS up to 31 are not used.

The base bits are transformed in one-hot formed by means of a decoder and used to select the right base address in the TCDM, which was previously put in the register file during the offload sequence. Selection of the true base is performed through three-state buffers and a wired or; since the address in the registers is byte-based the two lowest bits are skipped to obtain a 30-bit word-based address. Since the true base could be anywhere inside the TCDM (there is no base alignment requirement), to put together base and offset it is necessary to use a full 30-bit adder; the address is then transformed again in byte-based by adding two zeros as least significant bits.

Table 5 shows the main parameters used to configure the address translation module.

| Parameter | Type | Values | Description |
|---|---|---|---|
| BASE_ALIGNMENT_BIT | natural | 1-31 | Lowest bit of the base address in the input (internal) address |
| INTERLEAVING_MULTIPORT | natural | T/F | If True, shifting is needed because the ports are generated through the INTERLEAVED Catapult directive |
| N_INTERLEAVING_SETS | natural | any | Number of sets of ports generated through the INTERLEAVED Catapult directive |
| ORG_VIRTUAL_PORT | array of natural | any | A vector of N_INTERLEAVING_SETS elements containing the number of ports associated with each set |

*Table 5: Address translation parameters*

## 5. Sequencer module

The sequencer module is responsible of sequentializing the memory access requests coming from the accelerator module with the objective of time-multiplexing them in the lower-bandwidth channel to the TCDM, using a simple round robin scheme with no arbitration. Each set of intermediate ports are associated uniquely to a master port on TCDM side; the sets must be disjoint. The datapath of the sequencer is shown on figure 11; to enhance readability the figure does not show all control, which greatly depends on the solution of the Catapult issues (as happens in the controller module).

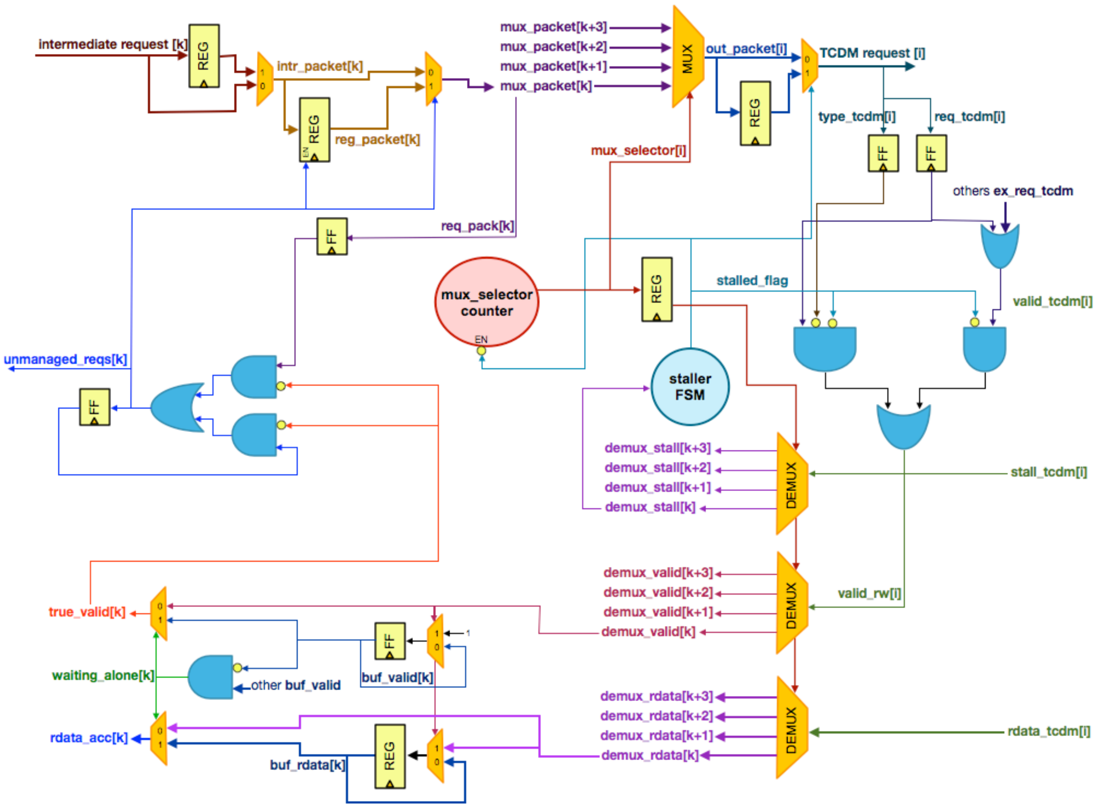

*Figure 11: Sequencer module microarchitecture*

A request from the intermediate port *k* forms an intr_packet[k] packet (depending on control flags it might also be registered before to tackle some corner cases). Depending from a flag unmanaged_reqs[k], generated in the response stage, the request might be buffered in a reg_packet[k] buffer which otherwise continues to store the previous buffered request. The unmanaged_reqs[k] flag indicates whether a request in the previous cycle could not be managed in the current cycle; in this case it has to remain in the buffer to be managed in one of the next cycles. In case the STALL_OR_FLAG is not active, the OR between all the unmanaged_reqs in the same sets is used to enable the sampling, while when it is active the OR between unmanaged_reqs from all ports must be used. unmanaged_reqs[k] is also used to select between the unbuffered intr_packet[k] (if there are no former requests unmanaged) and the buffered reg_packet[k]. The selected packet, mux_packet[k], is the request that intermediate port *k* will bring to the TCDM.

To select a single request to be brought to master port *i* from a set of intermediate ports, a counter (stopped only when the stalled_flag is active) generates a mux_selector[i] signal. The out_packet[i] which is obtained is the signal containing the request to be brought to TCDM. In case of a TCDM stall in the previous cycle, indicated by the stalled_flag, a buffered version of the request is sent to TCDM.

For the response side, the first step is to use the req signals sent to TCDM in the previous cycle to generate a valid_rw[i] signal; the response is always considered valid for writes and is the valid_tcdm[i] signal for reads, except when there was a stall in the previous cycle. The response, composed by this valid signal, the read data and the current stall signal is then assigned to its intermediate port with a demuxer fed with the mux_selector[i] signal of the previous cycle; the result are the demux_stall[k], demux_rdata[k] and demux_valid[k] signals.

Then demux_valid[k] and demux_rdata[k] signals can't always be directly sent to the intermediate port, because the response of the ports associated to the same master port must happen at the same cycle. If demux_valid[k] is active, the buf_valid[k] register is set to 1 and buf_rdata[k] registers demux_rdata[k]. The buf_valid[k] bit is then used to generate a waiting_alone[k] flag, which is 1 only when buf_valid[k] is the only register in its set to be still not active. The true_valid[k] and rdata_acc[k] signals, which are the ones that must be sent to the intermediate port, are always the buffered signals except in case waiting_alone[k]: in this case, to spare a clock cycle, the unbuffered version is used.

The true_valid[k] is used to generate the unmanaged_reqs[k] for the request stage; though it indicates logical validity of the value on rdata_acc[k] it can't be directly sent to the intermediate port: all stall signals in a set of intermediate ports must be mantained to 1 until all their data is valid. Depending on the STALL_OR_FLAG, the stall signal might be generated with some elaboration on the true_valid signals of a single group and the unmanaged_reqs[k] signal (if the stall OR workaround is not active), or with just an OR of all unmanaged_reqs[k] signals (if the workaround is active).

When all requests are always active, there are no TCDM stalls and N_INTERMEDIATE_PORT is an even number of times N_MASTER_PORT the sequencer is able to work without spending additional cycles with respect to those needed for pure memory communication; the upper limit for the throughput is given by the ratio of N_INTERMEDIATE_PORT divided by N_MASTER_PORT, multiplied by the throughput in case the number of master ports is N_INTERMEDIATE_PORT. When one of the demux_stall[k] signals is activated, however, the whole sequencer is frozen with the stalled_flag and the last requests are repeated until the stall is lifted. Note that the current implementation has a single stalled_flag but, whenever it is not any more necessary to use the STALL_OR_FLAG workaround, it will actually be more convenient to implement a stalled_flag[i] for each master port. The stalled_flag is lowered all data is valid or all demux_stall signals are lowered

As the controller module, also the sequencer must ensure correctness and currently this is achieved through the two workaround flags STALL_OR_FLAG and LAST_WRITE_FLAG. The STALL_OR_FLAG changes the generation of the output stall signal and the buffering of requests, while the LAST_WRITE_FLAG is used to invalidate the content of the response buffers in case a stall on a last write of a group happens.

Table 6 shows the parameters used to configure the sequencer module.

| Parameter | Type | Values | Description |
|---|---|---|---|
| ORG_SEQUENCER_PORT | array of natural | any | A vector of N_MASTER_PORT elements containing the number of intermediate ports associated with each master port |

*Table 6: Sequencer parameters*

## 6. Integration in P2012 v0.2.2

Both the wrapper and test HWPEs were integrated into the P2012 cluster_soft module, i.e. inside the cluster and out of the ENcore module containing the PEs. Four ports for the HWPE wrapper were added to the logarithmic interconnect in the same channel as DMAs; as a consequence currently the cluster supports either 4 wrappers with one master port each, or 2 wrappers with 2 ports each, or one wrapper with 4 master ports. The code related to Shared Memory HWPEs is generated with a similar flow to the current Streaming HWPEs; by modifying the input architecture XML file it is possible to create versions of the cluster with any variety of HWPEs. However, since all of the parameters (except for BASE_ALIGNMENT_BIT) are defined inside a package and not through generics, at the moment all HWPE wrappers must share the same structure (number of ports, configuration flags). This will be fixed in a future version.

## 7. Verification and testbench

The shared memory HWPE wrapper has been verified both with the P2012 cosimulation environment and a specialized testbench using a very simple IDCT accelerator module. Other simulations will be performed using more complex accelerators in the near future.

The cosimulation environment is used mainly to verify the offload procedure (and thus the slave module) and address translation, and to check the functionality of the shared memory HWPE in a real program (albeit a small one). Various software tests have been implemented using the HAL (described in a later section), each of them trying a different set of the features of the wrapper or a different offloading scheme in an environment where multiple PEs try to offload jobs (such as schemes trying to explicitly mantain job balancing between the different PEs versus one that do not enforce it). All tests are passing at the moment.

The specialized testbench is designed to offload 100 equal jobs and let the wrapper and accelerator work on a small unsynthesizable memory which dumps data written to it in a text file and randomly generates stall with probability defined in a parameter. In this way, it is possible to stress the controller and sequencer module and hunt for corner-cases due to particular sequences of stalls; by checking the output dump against a "golden" dump obtained through execution in software it is possible to detect errors.

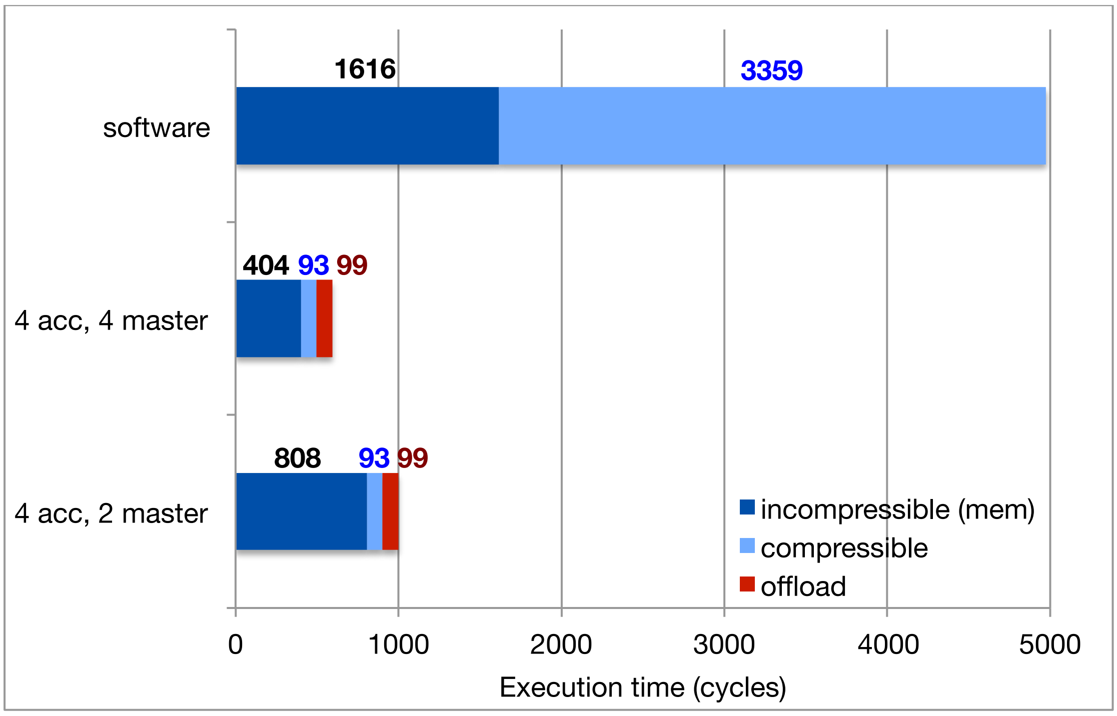

*Figure 12: Execution time of 6 blocks IDCT*

Figure 12 presents the simulation results obtained for the acceleration of an IDCT kernel on a very small grayscale image (six 8x8 blocks). The image shows clearly the two mechanisms that are used in tightly-coupled acceleration to enhance performance: *compression* of execution time due to pure computation and *parallelism*. The amount of time that is devoted to communication with memory is incompressible and it can be enhanced only by parallelism, i.e. by improving memory throughput as happens in the 4 and 2 master ports accelerators. Pure computational time, on the contrary, can be compressed through execution on hardware. Thanks to the multiple context mechanism, the HWPE can be maintained almost always working and therefore the overhead of the offload procedure is small even in this example’s tiny image.

The next step in verification is to try the wrapper with different (and more complex) accelerators and to simulate and debug the size converter module. Some passing tests have already been conducted with a version of this wrapper on a HOG (histogram of oriented gradients) accelerator, but a more thorough examination is certainly needed.

## 8. Accelerator HLS

High level synthesis of accelerator modules is currently performed with the Calypto Catapult SL tool, version 2011a.161. To interface correctly with the wrapper, the accelerator must use the *ram_w_handshake* protocol described in the controller section and provide a start, a done and an enable signal. To use the *ram_w_handshake* protocol, all inputs and outputs must be expressed in terms of pointers to input/output vectors passed as parameters to the C function used to generate the accelerator. Due to the number of internal registers, up to 16 input/output parameters. A future version of the wrapper will also support up to 8 configuration registers.

Each set of input/output parameters must be associated to an interface resource of type *ram_w_handshake* inside Catapult, which on turn will be transformed in a single port or a set of interleaved ports in case the INTERLEAVE directive is used. To ensure that all addresses inside a resource are base aligned (as required by the address translation module) the BASE_ALIGNED directive must be set for all interface resources; at the moment only accelerators with a single memory resource are supported, with a free number of interleaved ports.

The RTL Verilog code generated through Catapult SL must pass through a very simple postprocessing stage which was designed to automatically comment out the lines causing bugs (if this is needed) and possibly change the name of the module to a custom one, to make it possible to connect multiple different accelerators to the cluster.

# 9. HAL and Programming model

A simple hardware abstraction layer, reported in table 7, has been developed which makes use of the register in the wrapper slave module to control the HWPE.

| Return type | Name | Parameters | Description |
|---|---|---|---|
| **int** | checkContextHWPE | **int** hwpe_id | Checks if there is a free context and initiates the offload sequence; returns **-1** if there is no free context, **-2** if the wrapper is locked in another offload procedure, otherwise **context_id** of the current offload |
| **void** | triggerHWPE | **int** hwpe_id | Ends the offload sequence |
| **void** | setSyncModeHWPE | **int** hwpe_id, **int** sync_mode | Sets synchronization mode to polling if sync_mode is 0, to event if 1 |
| **int** | isWorkingHWPE | **int** hwpe_id | Returns 0 if all jobs have been executed, the content of the status register otherwise |
| **char** | isWorkingContextHWPE | **int** hwpe_id, **int** context_id | Returns 0 if job in context context_id has been executed, 1 otherwise |
| **int** | getRunningContextHWPE | **int** hwpe_id | Returns the ID of the context currently running on the accelerator |
| **int** | getPointerContextHWPE | **int** hwpe_id | Returns the ID of the next context for the offload procedure (similar to checkContextHWPE(), but without |

| | | | |
|---|---|---|---|
| | | | initiating the offload) |
| **char** | getOwnerContextHWPE | **int** hwpe_id, **int** context_id | Returns 0 if context context_id has never been offloaded, otherwise the ID+1 of the latest PE who offloaded a job to context context_id |
| **void** | setGenericRegHWPE | **int** hwpe_id, **int** reg, **int** value | Sets generic register reg (from 0 to N_GENERIC_REGS-1) to value |
| **void** | setIORegHWPE | **int** hwpe_id, **int** reg, **int** value | Sets IO register reg (from 0 to N_IO_REGS-1) to value |
| **int** | setGenericRegHWPE | **int** hwpe_id, **int** reg | Gets value of generic register reg (from 0 to N_GENERIC_REGS-1) |
| **int** | setIORegHWPE | **int** hwpe_id, **int** reg | Gets value of IO register reg (from 0 to N_IO_REGS-1) |

*Table 7: HAL*

A standard offload sequence, after checkContextHWPE has acquired the lock, uses the following scheme:

```
static inline void offloadHWPE_foo(int hwpe_id, int sync_mode,
int d_in, int d_out) {
   setSyncModeHWPE(hwpe_id, sync_mode);
   setIORegHWPE(hwpe_id, 0, d_in);
   setIORegHWPE(hwpe_id, 1, d_out);
   triggerHWPE(hwpe_id);
}
```

If function foo() is substituted by execution on a HWPE, this piece of code

```
int main() {
   int d_in[SIZE], d_out[SIZE] __attribute__((section(".cluster
")));
   // ...
   // Loop
   while(i<NBLKS) {
      foo(d_in, d_out);
   };
   // ...
}
```

can be substituted by this other piece, which shows a complete offload sequence and the PE going idle and waiting for an event from the HWPE.

```
int main() {
   int d_in[SIZE], d_out[SIZE] __attribute__((section(".cluster
")));
   // ...
   // Offloading loop
   while(i<NBLKS) {
     // Poll if context is locked by another PE
     while((check_ctxt = checkContextHWPE(HWPE_TO_USE)) == -2);
     // If all contexts are busy, wait for new barrier
     if(check_ctxt == -1) {
       rise_itcomb_status();
       wait_event(8);
     }
     // If context is free, offload new task
```

```
        else {
          offloadHWPE_foo(FOO_HWPE_ID, SYNC_MODE, (int) &d_in, (in
t) &d_out);
          i=i+2;
        }
      };
      // ...
}
```

This basic HAL allows using the tightly coupled HWPE with two different abstractions in mind:

- In the lower level abstraction, the shared memory HWPE can be seen as a **peripheral resource** similar to a DMA controller, which can be configured by reading/writing some registers and performs some work on behalf of a PE. This abstraction is of course very near to the way the wrapper is implemented.
- In the higher level abstraction, the HWPE is itself a processing element, not dissimilar in principle from a processor, implementing a **hardware thread**. This abstraction is the one at the basis of the whole tight coupling idea, in which the software developer does not need to know from the start which parts of his code will work on software and which will be HW accelerated.

While the first abstraction has been the main guide for the development of the slave module and the HAL, the second one is probably nearer to the optimal way of using tightly coupled acceleration, and therefore one of the possible next steps is to provide a software layer making this abstraction more transparent. This will need tighter integration between the software stack and the HLS tools.

## 10. Preliminary synthesis results

A preliminary synthesis of the wrapper in ST 32 nm technology was performed during the design cycle - to identify critical points - and at the end of the internship.

In terms of area, most occupation is due to sequential logic in the register file and in various buffers inside the controller and sequencer blocks. A great optimization in this regard has been performed by defining the register file in such a way that only registers that are useful for the specific HWPE are physically present, even if the register file maintains the same structure. Figure 12 shows the scaling of area occupation while varying the number of accelerator ports, master ports and contexts.

From the point of view of timing, it is quite difficult to evaluate the wrapper from a standalone synthesis because it contains fully combinational paths which would need a synthesis of the whole "accelerator + wrapper + logarithmic interconnect + TCDM" block to be characterized. However, it is possible to give an estimate of the critical paths by posing reasonable but relaxed delay constraints on the input and output ports of the wrapper in such a way that it meets timing with a 500 MHz target frequency, and then tighten them until reaching a point where the timing is not met any more, to evaluate the limit critical delay introduced by the wrapper.

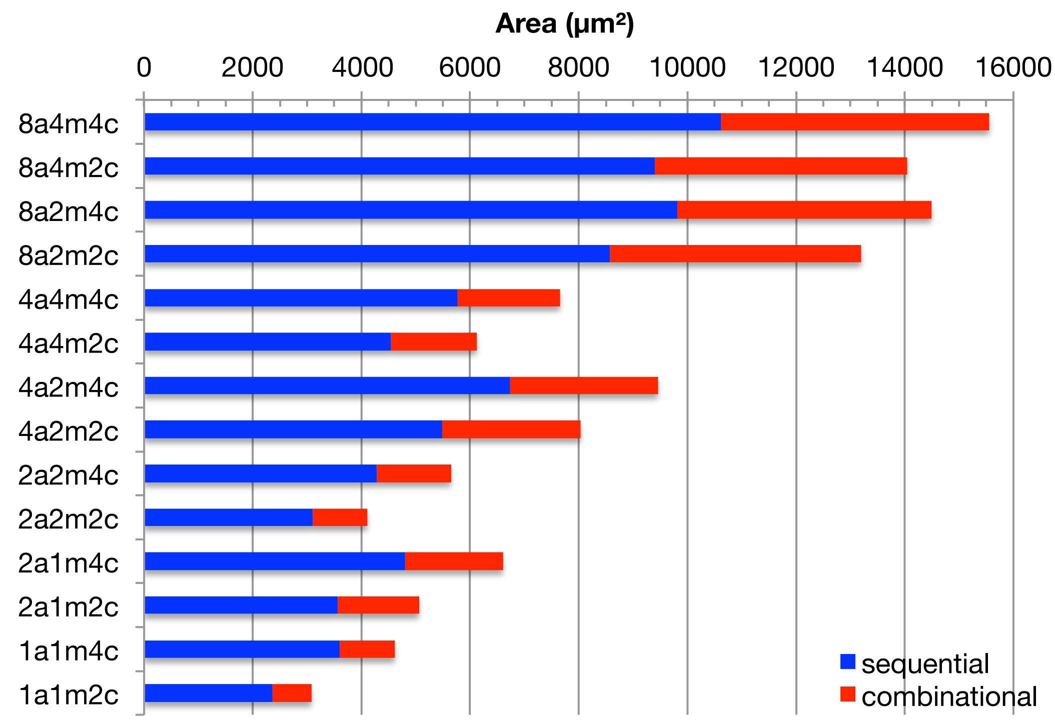

*Figure 13: Area of wrapper after preliminary synthesis run in ST 32nm technology*

The "relaxed" constraints that were used in the preliminary synthesis of the wrapper are the following; they were obtained over a 2 ns clock period with a clock uncertainty of 0.4 ns:

- all inputs from the accelerator module were constrained to be ready after **0.6 ns** from the clock edge, and all outputs towards the accelerator module were constrained to be ready **0.6 ns** before the clock edge.
- all outputs to the logarithmic interconnect were constrained to be ready **0.7 ns** before the clock edge, and all inputs except from the stall/grant signal were constrained to be ready after **0.7 ns** from the clock edge.
- the grant input from the logarithmic interconnect was constrained to be ready **1.2 ns** before the clock edge.

Note that this - considering the 0.4 ns of clock uncertainty - gives a 0.3 ns window for signals in the request path. The tightening of the constraints was tested on the wrapper with 4 accelerator, 4 master ports and 2 contexts, which is the configuration best known and most often used also in simulations.

It appears from synthesis results that the critical path is the one that starts from the *catc_add_i* signal, passes through the address translation block, the rest of the controller and then outputs to the *tcdm_add_o* signal. By tightening the constraints on the outputs to the logarithmic interconnect, it is possible to reduce the total time of this path to about **0.2 ns**.

Another very dangerous path is the one coming back combinationally from the logarithmic interconnect through the *tcdm_gnt_i* signal; while the grant/stall signal is always registered in some way inside the wrapper, the time budget in this case is extremely tight because this signal depends combinationally upon the *catc_add_i* signal itself (this is the reason for which its delay was modeled differently than the others). Timing constraints were met for a *tcdm_gnt_i* delay of up to **1.5 ns**.