A

Convey PDK2 Reference Manual

February 11, 2015

Version 2.1

900-000046-000

1302 East Collins

Richardson, TX 75081

The Information in this document is provided for use with Convey Computer Corporation ("Convey") products. No license, express or implied, to any intellectual property associated with this document or such products is granted by this document.

All products described in this document whose name is prefaced by "Convey" or "Convey enhanced " ("Convey products") are owned by Convey Computer Corporation (or those companies that have licensed technology to Convey) and are protected by patents, trade secrets, copyrights or other industrial property rights.

The Convey products described in this document may still be in development. The final form of each product and release date thereof is at the sole and absolute discretion of Convey. Your purchase, license and/or use of Convey products shall be subject to Convey's then current sales terms and conditions.

Trademarks

The following are trademarks of Convey Computer Corporation:

The Convey Computer Logo: 

Convey Computer

HC-1

HC-1^{ex}

HC-2

HC-2^{ex}

Trademarks of other companies

Intel is a registered trademark of Intel Corporation

Adobe and Adobe Reader are registered trademarks of Adobe Systems Incorporated

Linux is a registered trademark of Linus Torvalds

Xilinx, Virtex and ISE are registered trademarks of Xilinx in the United States and other countries.

ChipScope, CORE Generator and PlanAhead are trademarks of Xilinx, Inc.

Revisions

Version	Description
1.0	January 29, 2014. Initial Release
1.01	February 4, 2014. Minor correction in accessing CSRs on HC platform. Updated links to customer support landing page.
1.02	February 10, 2014. Updated customer support link
2.0	July 30, 2014. Added support for Wolverine. Updated simulation section.
2.1	February 11, 2015. Added information for accessing CSRs on WX.

Table of Contents

1 Overview

1.1 Introduction

The Convey coprocessor is a programmable hardware solution to increase application performance beyond what is typically possible in a standard x86 system. Because of its programmable nature, the hardware allows the architecture to be reconfigured to meet the needs of the application. These reconfigurable applications are called *personalities.*

Convey provides personalities, which can be used to accelerate certain applications. Some applications, however, require specialized functionality that is not provided by these personalities. As a result, Convey has designed a framework to enable the development of custom personalities. The Personality Development Kit (PDK) provides all the hardware, software and simulation interfaces necessary to implement a custom personality on the Convey family of products.

This manual describes the Convey PDK. It is intended for hardware and software engineers developing custom personalities for the Convey coprocessor, as well as those developing or debugging custom applications that use custom personalities.

1.1.1 How to Use This Manual

This manual is intended to be both a design guide as well as a reference manual. The Coprocessor Architecture chapter describes the architecture of the Convey Coprocessor focusing on the Application Engine. Chapter 6 outlines the process of developing a custom personality and references other chapters and documents for more detailed information on each subject.

2 Coprocessor Architecture

2.1 Coprocessor Hardware

The Convey Coprocessor is made up of three major sets of components: The Application Engine Hub (AEH), the Memory Controllers (MCs) and the Application Engines (AEs). This document focuses on the Application Engines and how they interface to the system. For a high-level system overview of coprocessor architecture, refer to the Convey HC or WX Reference Manual.

The Application Engines (AEs) are the areas that are reconfigured for different personalities. Custom instructions developed for the Convey coprocessor are implemented in the Application Engines. The AEs contain 4 major interfaces to the rest of the system: the dispatch interface, memory controller interface, CSR interface and AE-to-AE interface. Those interfaces are described in detail later in this document.

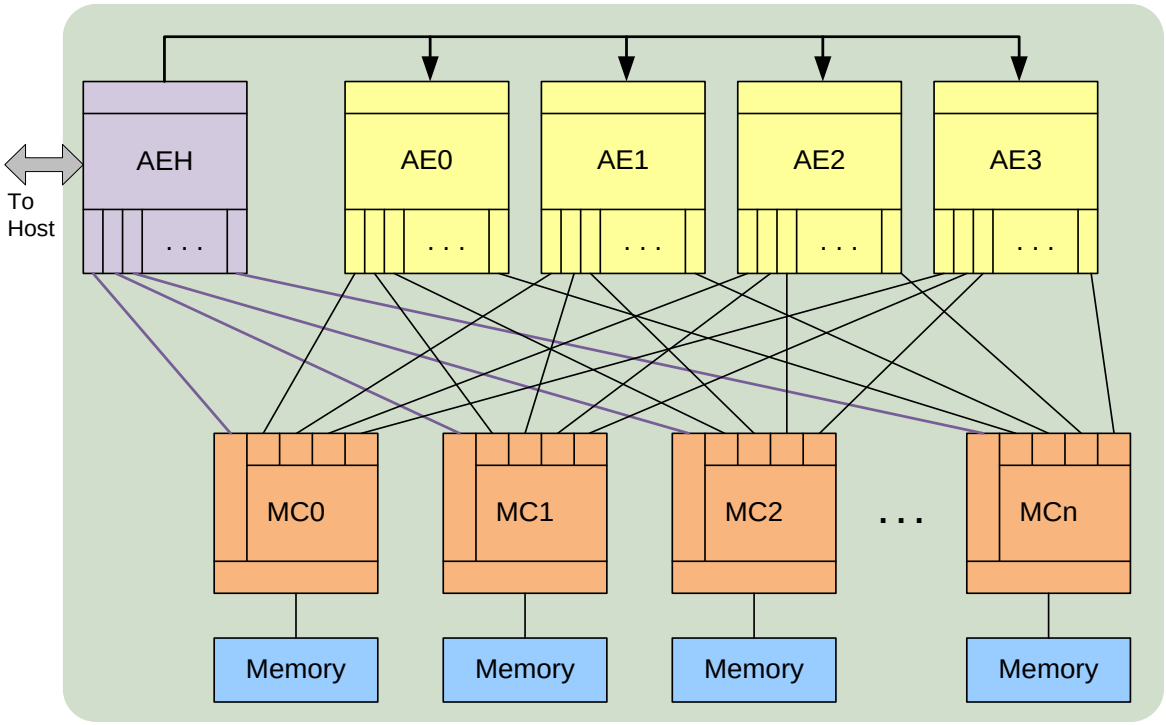

Figure 1 – Convey Coprocessor Block Diagram

3 PDK Components

The Personality Development Kit includes the following components:

- Convey PDK Reference Manual (this document)
  The PDK Reference Manual defines the hardware and simulation interfaces that make up the Personality Development kit. It provides step-by-step instructions for developing and packaging a custom personality.

- Convey HC PDK Programmers Guide
  The Convey HC PDK Programmers Guide describes the development of host applications for use with PDK personalities on the HC coprocessors.

- Convey Wolverine PDK Programmers Guide
  The Wolverine PDK Programmers Guide describes the development of host applications for use with PDK personalities on Wolverine coprocessors.

- FPGA hardware interfaces
  Provided as Verilog modules, these interfaces connect custom personality hardware to instruction dispatch, management and memory resources on the coprocessor.

- Custom personality software and hardware simulation environment
  Bus-functional models are provided to connect each of the hardware interfaces to Convey's architecture simulator.

- Sample Personality
  A sample personality illustrates how to use the hardware and simulation interfaces to develop a custom personality.

4 Requirements

The PDK is compatible with both the Covey HC and Wolverine coprocessors.

4.1 Convey Software

The Convey Development tools require various other Convey software packages, such as runtime libraries and Convey simulator. See the Customer Support webpage for requirements for the appropriate platform and usage scenario.

4.2 Other Requirements

In addition to the Convey packages listed above, the following are required for PDK development:

- Xilinx ISE Design Software for synthesis, place and route of FPGAs. The following revisions are tested:
  - Xilinx ISE Design Suite 14 for HC
    - 14.7
  - Xilinx Vivado Design Suite 14 for Wolverine
    - 2014.1
- An HDL simulator for Verilog/VHDL simulation. Mentor ModelSim and Synopsys VCS are supported. The following versions are tested:
  - ModelSim: 10.1+
  - VCS: H-2013.06.1

5 PDK Personalities

PDK personalities are generally specific to an application and are used to replace a performance critical code section with a single coprocessor instruction. The PDK infrastructure defines machine state and a means to access the machine state.

Supported Machine State

The following figure shows the machine state extensions defined for PDK personalities.

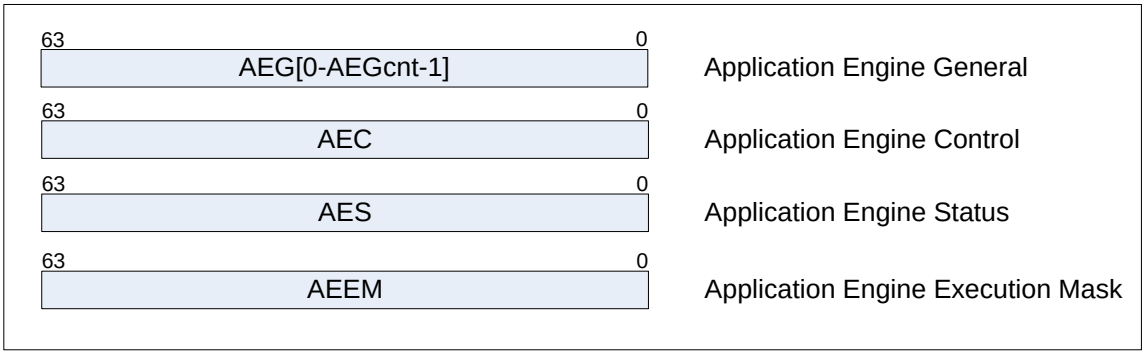

Figure 2 – PDK Personality Machine State

5.1.1 AEG Register

The AEG register is a single register with multiple elements (AEGcnt). Each element is 64-bits in size. The PDK accesses the AEG register as a single one-dimensional structure. The PDK personality may subdivide the AEG register elements as needed, with each subset organized as multi-dimensional structures as appropriate for the customer algorithm being defined.

The AEG register is non-persistent state.

5.1.2 AEC – Application Engine Control Register

The AEC register contains the exception mask field, AEM. All bits other than those within the defined AEM field are reserved. A separate copy of the AEC register is maintained per user command area. All fields are persistent state. The following figure shows the fields of the AEC register.

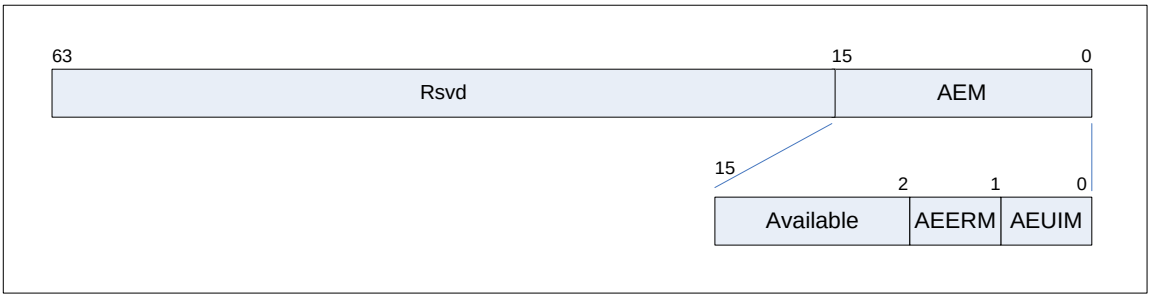

Figure 3 – Application Engine Control Register

The fields of the AEC register are defined in the following table.

Field Name	*Bit Range*	*Description*
AEM	15:0	The **Application Engine Mask** specifies which exceptions are to be masked (i.e. ignored by the coprocessor). Exceptions with their mask set to one are ignored.

Table 1 – AEC Register Fields

Refer to section 5.1.3 for a description of each exception type.

5.1.3 AES – Application Engine Status

The application engine status register holds the status fields for an application engine. Each application engine may have different AES register values. The entire register is initialized to zero by a coprocessor dispatch. The meaning of all bits other than those within the AEE field is reserved and read as zero.

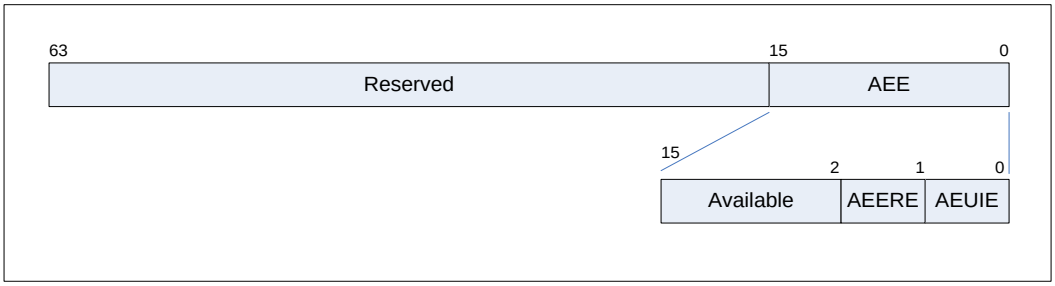

Figure 4 – Application Engine Status Register

Logic in the Application Engine checks for exceptions during instruction execution. Detected exceptions are recorded in the AEE field. The AEE field is masked with the AEC AEM field to determine if a recorded exception should interrupt the host processor. The results of the following exception checks are recorded.

Exception Name	*AEE Bit*	*Description*
AEE	15:0	The **Application Engine Exception** field indicates which exceptions have been signaled. Only exceptions with the associated mask bit set in the AEC.AEM field are sent to the host processor.
AEUIE	0	AE Unimplemented Instruction Exception. An attempt was made to execute an unimplemented instruction. The operation of the unimplemented instruction depends on the value of the instruction's opcode. If the opcode is in a range of opcodes that is defined to return a value to the AEH then the value zero is returned; otherwise the instruction is equivalent to a NOP.
AEERE	1	AE Element Range Exception. An instruction referenced an AEG register where the element index value exceeded the valid range.
	1:0	Reserved for PDK use.
	15:2	Available for specific PDK use. A PDK may define additional exceptions.

Table 2 – AES Register Fields

5.1.4 AEEM – Application Engine Execution Mask

On the HC platform the AEEM control register is used to disable some AEs from participating in instruction execution. All instructions other than moves to/from the AEEM register itself use the AEEM register to control instruction execution.

PDK directed instructions (denoted by the <.ae#> suffix, i.e. mov.ae0) can be used to target a single AE, but the IEEM bit for that AE must be set to enable that AE to participate in the instruction.

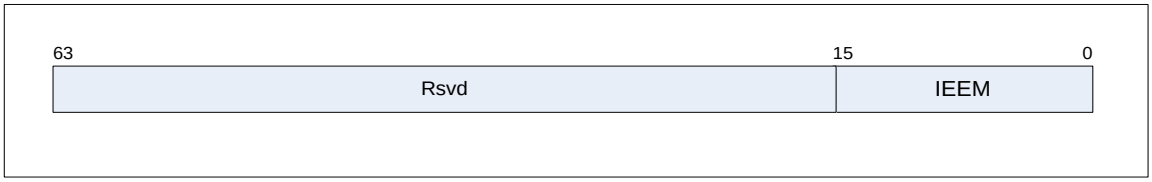

Figure 5 – AEEM Control Register Format

The fields of the AEEM register are defined in the following table.

Field Name	*Bit Range*	*Description*
IEEM	15:0	The **Instruction Execution Enable Mask** field specifies which application engines are to participate in instruction execution. Bit zero when set to a one enables application engine #0, bit one enables AE #1, etc.

Table 3 – AEEM Register Fields

6 PDK Development Steps

This chapter contains step-by-step instructions for the process of developing a custom personality for the Convey Coprocessor.

6.1 Analyze Application

The first step of personality development is to completely understand the problem to be solved. How does the current application perform on existing hardware? What bottlenecks are limiting the performance? What data structures are involved? How parallelizable is the application?

Answers to these questions provide the first insight into how the application can be accelerated in hardware. Some tools that are useful in gathering this information are **gprof** (The GNU Profiler) and **oprofile**.

6.2 Evaluate Hardware Options

With a detailed knowledge of the application and its performance limitations, the second step is to evaluate options for implementing the application in hardware. This requires a good understanding of the hardware architecture and the FPGA resources available to the custom personality.

Once a concept for hardware design is completed, the performance of the hardware can be compared relative to the existing application performance.

6.3 Develop Software Model of Custom Personality

Convey provides an architecture simulation environment to allow rapid prototyping of both the hardware and software components of a custom personality. This environment is written in C++ to emulate the rest of the system. It includes hardware models of instruction dispatch, register state and the memory subsystem.

With a hardware design concept in place a software model can be developed to emulate the hardware. The hardware model can then be simulated with the rest of the system to prove the concept before detailed design begins.

Chapter 7 describes how to develop a software model of the custom personality.

6.4 Replace Application Kernel with Call to Coprocessor

The application should be modified so that the application kernel can be called. To dispatch instructions to the coprocessor, the kernel function call is replaced with a call to dispatch the function to the coprocessor.

A detailed description of PDK host application development is contained in the Convey HC PDK Programmers Guide and the Convey Wolverine PDK Programmers.

6.5 Compile Application

A sample makefile is provided illustrating how the code should be compiled.

6.6 Simulate Application with Convey Architecture Simulator

Once the AE software model is in place and the appropriate changes to the application have been made, the application can be run against the Convey architecture simulator. This step allows the application to be debugged before the hardware is designed.

6.7 Develop FPGA Hardware

Chapter 8 of this manual describes the development steps necessary to implement a custom personality in the FPGA.

6.8 Simulate Hardware in Convey Simulation Environment

As an extension of the Convey architecture simulator, Convey provides a hardware simulation environment containing all hardware interfaces to the custom personality. Using a standard VPI interface (Verilog Procedural Interface) the architecture simulator can be used to provide stimulus to the HDL simulation.

6.9 Integrate with Convey Hardware

The final step is to run the application on the Convey Coprocessor hardware.

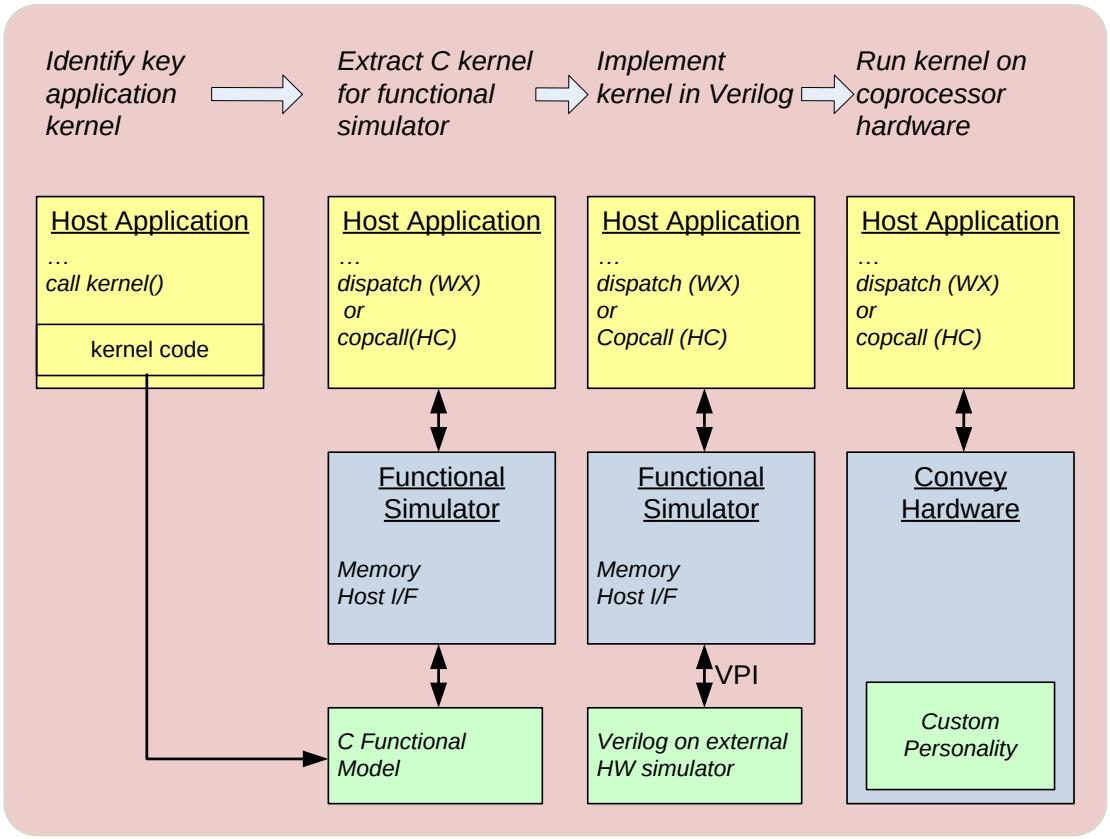

Figure 6 – PDK Development Flow

7 Custom AE Software Model Development

Convey's architecture simulator was developed to allow software to be tested and debugged in the absence of the actual Convey coprocessor hardware. It can also be used to prototype a PDK design quickly before investing the time to develop the FPGA hardware.

For custom personalities, the functionality of the AE is defined by the customer and therefore cannot be modeled in the simulator. A socket interface is designed into the simulator to allow a customer-developed the AE software model to connect to the simulation process.

7.1 Conceptual Overview of the Software Modeling Process

The application executable is a Linux executable, where host code calls to coprocessor routines are routed to the simulator. The host x86-64 code and the coprocessor simulator share the memory space of the executable, just as the real Convey coprocessor shares memory with the x86-64 host code. The Convey simulator can execute user written software emulation routines that *emulate* the user-defined functionality, or use a Verilog simulator with the user developed custom personality.

7.2 Developing the AE Software Model

The AE Software Simulation process is made up of the system interfaces (provided by Convey) and the descriptions of custom instructions and AEG registers, which are provided by the customer.

7.2.1 Functions Implemented by Custom Personality

The software model of the custom personality must implement the following functions:

```
void CCaeIsa::InitPers()

void CCaeIsa::CaepInst(int masked, int ae, int opcode, int immed, uint64 scalar)
```

7.2.1.1 InitPers

The InitPers function is called by the architecture simulator and allows the custom personality code to set variables inside the simulator. Inside this function, the personality must call the following function to set the max AEG index used by the custom personality.

```
SetAegCnt(MAX_AEG_INDEX);
```

The personality must also call the following function to provide the simulator the personality number of the custom personality.

```
SetPersNum(PERS_NUM)
```

7.2.1.2 CaepInst

The CaepInst function is called by the architecture simulator anytime a custom AE instruction (CAEP00-CAEP1F) is called. This call models the instruction dispatch interface in the hardware, and includes as arguments all information needed by the custom personality to decode an instruction:

- aeId: AE ID (0-3)

- opcode: instruction opcode (0x00 – 0x3f)
- immed: 18-bit immediate
- inst: 32-bit instruction
- data: 64-bit scalar

The customer must model all instructions implemented by the custom personality by decoding the opcode field. For opcodes that are not implemented, the function SetException(<ae>, AEUIE) must be called to assert the unimplemented instruction exception. Note for Wolverine coprocessors, the coprocessor dispatch generates an opcode of 0x00.

The complete list of functions available to the custom personality is in the next section.

7.2.2 Functions Callable by Custom Personality

AeMemLoad and AeMemStore are used to load data from memory and store data to memory, respectively.

ReadAeg and WriteAeg are used to access the Application Engine General (AEG) registers in the simulation model. These registers are defined by the personality.

SetException is used to send exceptions to the architecture simulator.

SetAegCnt is used to send the simulator the number of AEG registers implemented in the personality.

7.2.2.1 AeMemLoad

Memory load requests are sent across the socket interface to the coprocessor simulator thread, which maintains memory state. A 64-bit value is returned.

bool CCaeIsa::AeMemLoad(int aeId, uint64 addr, int size, bool bSigned, uint64 &data)

- int aeId is the Application Engine Identifier, which is a number from 0 to 3 indicating the index of the AE sending the request.
- int size is the size of the request to the MC in bytes. Valid sizes are 1, 2, 4 and 8. Note that the simulator expects logical size values, not encoded values as in the FPGA signal interface.
- bSigned indicates whether the data is signed (1) or unsigned (0)
- unit 64 &data will contain the data loaded from memory. The data is right justified.

7.2.2.2 AeMemStore

Memory store requests are sent across the socket interface to the coprocessor simulator thread, which maintains memory state. No return value.

bool CCaeIsa::AeMemStore(int aeId, uint64 addr, int size, bool bSigned, uint64 data)

- int aeId is the Application Engine Identifier, which is a number from 0 to 3 indicating the index of the AE sending the request
- int size is the size of the request to the MC in bytes. Valid sizes are 1, 2, 4 and 8. Note that the simulator expects logical size values, not encoded values as in the FPGA signal interface.
- bSigned indicates whether the data is signed (1) or unsigned (0).
- unit 64 &data contain the data to be stored to memory. The data is right justified

7.2.2.3 ReadAeg

Read a 64-bit AEG register.

uint64 CCaeIsa::ReadAeg(int aeId, int aegIdx)

- int aeId is the Application Engine Identifier, which is a number from 0 to 3 indicating the index of the AE sending the request.
- int aegIdx is the index of the AEG to be read or written.

7.2.2.4 WriteAeg

Write a 64-bit AEG register.

void CCaeIsa::WriteAeg(int aeId, int aegIdx, uint64 data)

- int aeId is the Application Engine Identifier, which is a number from 0 to 3 indicating the index of the AE sending the request.
- int aegIdx is the index of the AEG to be read or written.
- unit64 data is the data to be written into the AEG

7.2.2.5 SetException

Exceptions defined by the custom personality can be set by calling SetException with an integer value of the exception bit number. For instance, an unimplemented instruction exception (AEUIE, defined in section 5.1.3) can be set by calling

void CCaeIsa::SetException(int aeId, int bitNum)

- int aeId is the Application Engine Identifier, which is a number from 0 to 3 indicating the index of the AE sending the request.
- int bitNum is the bit number of the exception bus to set. Setting bitNum to 4 sets bit 4 of the bus.

7.2.2.6 SetAegCnt

The AEGCnt defines the maximum AEG register index used in the custom personality. The custom personality software model must call this function in the InitPers() routine to set AEGCnt appropriately.

void CCaeIsa::SetAegCnt(int AegCnt)

7.2.2.7 SetPersNum

The PersNum defines the personality number of the personality for the simulator.

void CCaeIsa::SetPersNum(int PersNum)

7.3 Compiling the Model

The AE software model is compiled into an executable using the source code for the custom personality and the Convey-supplied libraries. This executable is automatically run by the Convey architecture simulator when an application is run on the architecture simulator and dispatches a PDK personality. The software model is also compiled into a shared library to be used in the Verilog simulation.

8 FPGA Development

8.1 Introduction

The custom personality hardware is implemented in the Application Engine (AE) on the Convey coprocessor. This chapter describes the FPGA development process.

8.2 FPGA Technology

The PDK supports development for the HC-1, HC-2, HC-1ex, HC-2ex, WX-690 and WX-2000 platforms. The Convey coprocessor uses the following FPGAs for the Application Engine FPGAs:

- HC1 / HC2: Xilinx Virtex 5 LX330, FFG1760C package
- HC1ex / HC2ex: Xilinx Virtex 6 LX760, FFG1760C package
- WX690: Xilinx Virtex 7 XC7VX690T, FFG1761 package
- WX2000: Xilinx Virtex 7 XC7VX2000T, FFG1761 package

More information about the FPGA technology is available at Xilinx.

8.3 Hardware Interfaces

The hardware interfaces described in this section are designed to enable simple integration of a custom personality into the Convey coprocessor. Each of these interfaces corresponds to a Verilog module provided by Convey. Together with the custom personality module(s) developed by the customer, these modules make up the design that will be synthesized into the AE.

8.3.1 Application Engine (AE) block diagram

The diagram below shows the Application Engine with the interfaces to the rest of the coprocessor:

- Dispatch Interface
- Memory Controller Interface
- AE-AE Interface
- CSR Interface

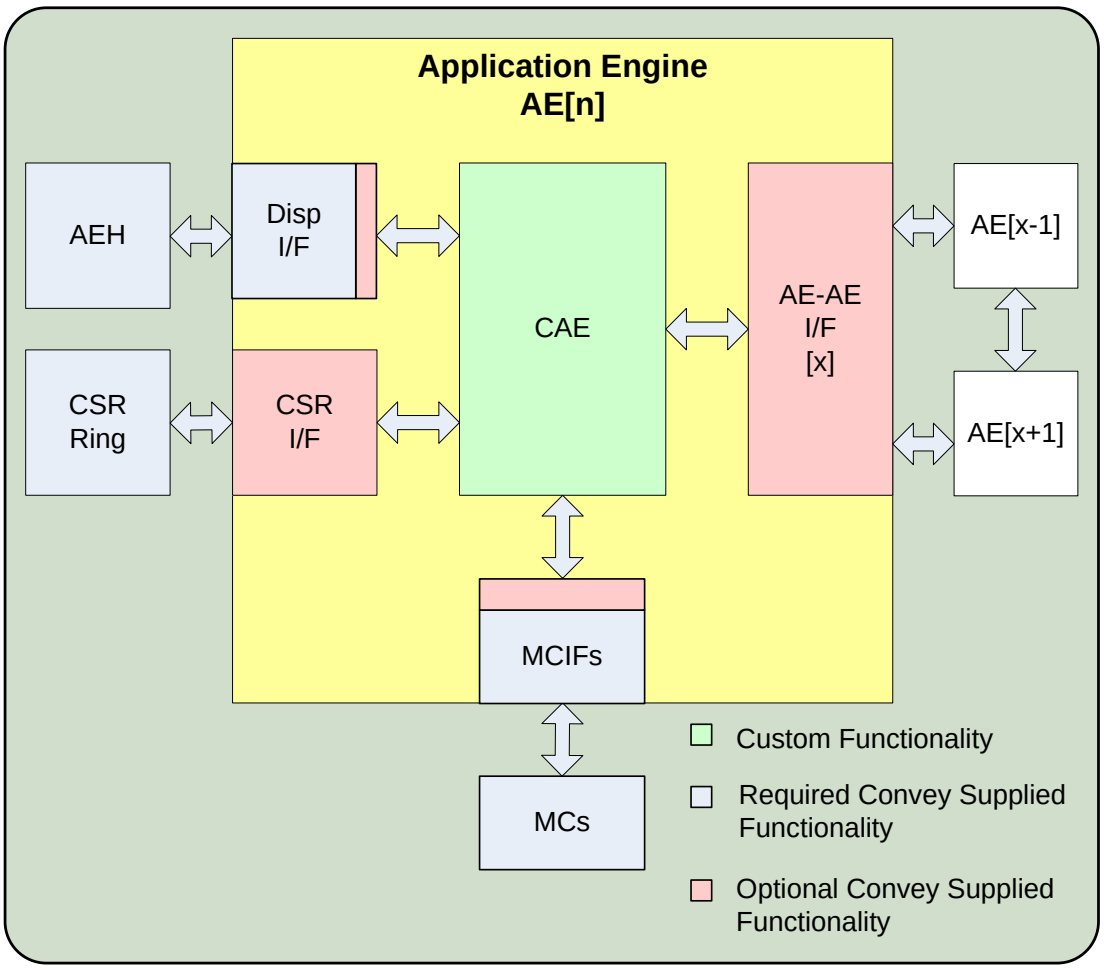

Figure 7 – Application Engine (AE) Block Diagram

8.3.2 Dispatch Interface

The dispatch interface is the hardware interface through which a host application sends coprocessor instructions to be executed by the AE. The dispatch interface receives instructions from the Application Engine Hub (AEH). Instructions intended for the Custom Application Engine (CAE) are passed to the custom personality through the signals described below. The dispatch interface also ensures that data is returned to the AEH when required by the instruction.

Figure 8 shows the data flow through the dispatch interface:

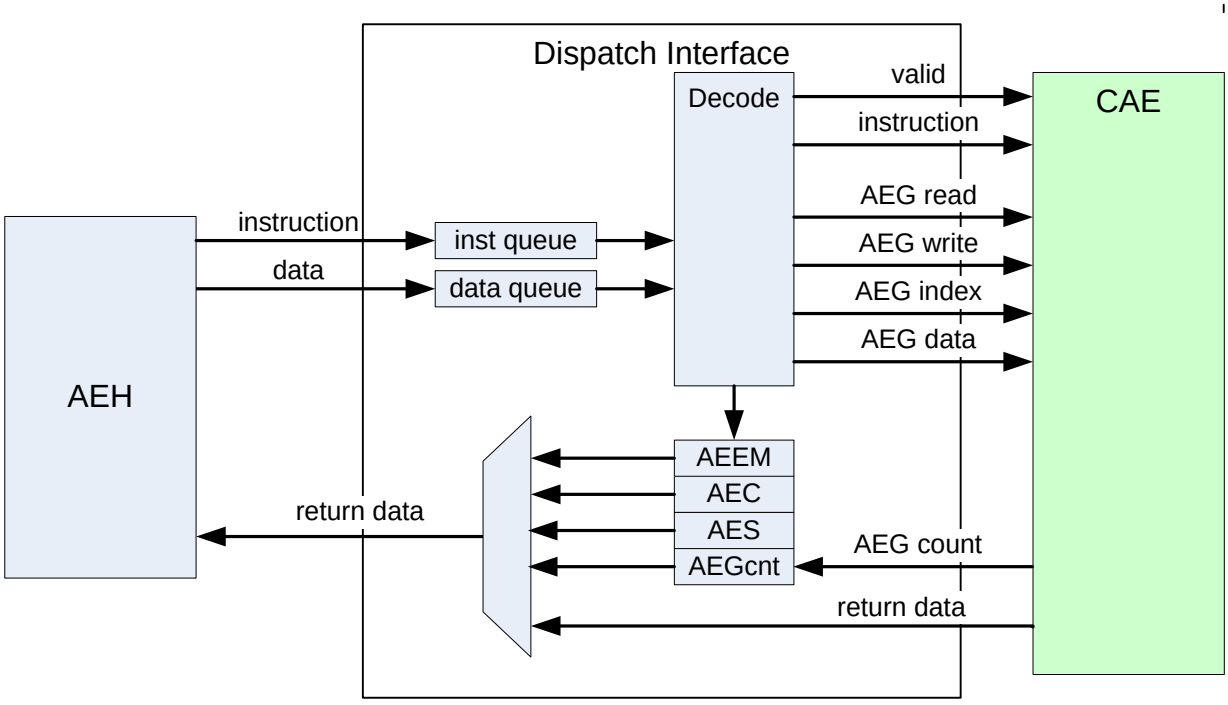

Figure 8 – Dispatch Interface Diagram

8.3.2.1 Registers

The dispatch module implements the AEEM (Application Engine Execution Mask), AEC (Application Engine Control) and AES (Application Engine Status) registers defined in Section 0.

8.3.2.2 Signal Interface to Custom Personality

Figure 9 shows the signal interface between the dispatch block and the custom AE personality. Detailed signal definitions are in Section 8.3.2.9.

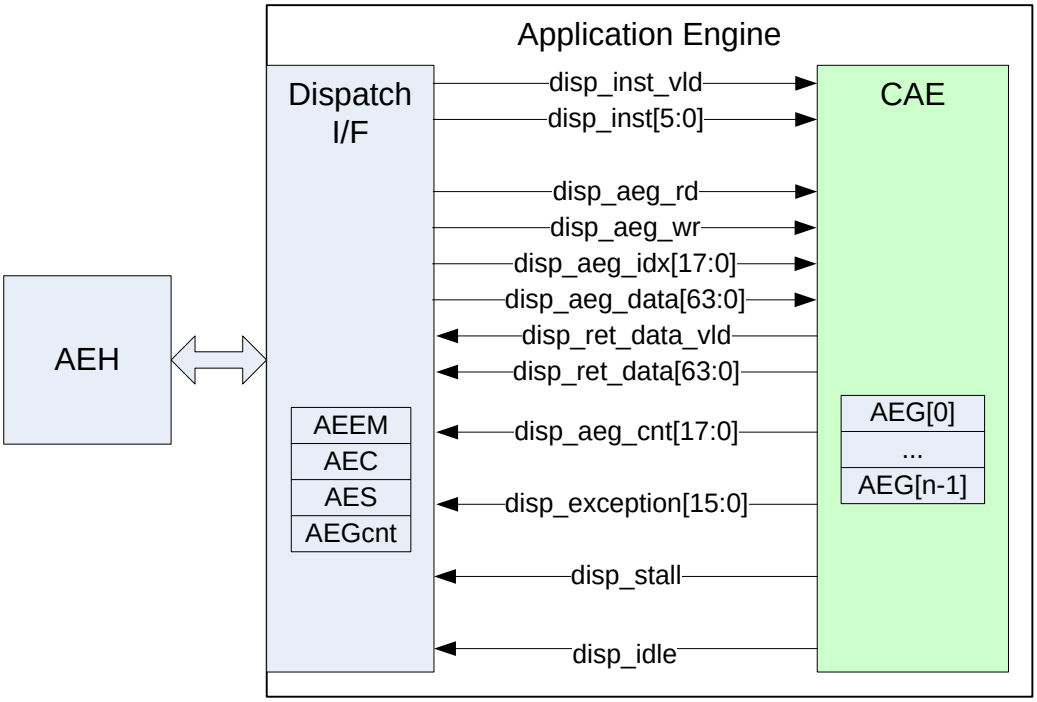

Figure 9 – Dispatch Interface Signals

8.3.2.3 Instruction decode

Instructions intended for the Custom Application Engine (CAE) personality are presented on the *dist_inst<5:0>* bus and are valid when *disp_inst_vld* is high. The dispatch interface asserts *dist_inst_vld* when the following are true:

- the instruction is a custom instruction
- the instruction is intended for the AE
- the AE is enabled in the AEEM register

Note: On Wolverine coprocessors a coprocessor dispatch always results in instruction 0 being sent to the CAE.

The dispatch module handles all reads and writes to the AEEM, AEC and AES registers, as well as reads of AEGCnt. In addition, the dispatch module has an unimplemented instruction filter that drops all instructions that cannot be decoded and provides the unimplemented instruction exception.

AEG reads and writes are decoded by the dispatch module. When an AEG read or write is decoded the personality is presented with the AEG index number on the *disp_aeg_idx<17:0>* bus. For reads *disp_aeg_rd* indicates the index is valid for a read. For writes *disp_aeg_wr* indicates the index is valid for a write and the data is valid on *disp_aeg_data<63-0>* on the same cycle.

The custom personality should generate an unimplemented instruction exception, by asserting *disp_exception<0>*, to cover all CAEP space that it does not use.

8.3.2.4 Return data

Data is returned to the AEH on the *disp_ret_data<63:0>* bus. The custom AE should drive *disp_ret_data_vld* high when the data is valid.

The ordering of return data must be preserved. The Convey-supplied dispatch module ensures that AEC/AES/AEGcnt/AEG data is returned in the correct order. However, the CAE personality must guarantee that reads of AEG registers stay ordered appropriately.

8.3.2.5 AEG_CNT

The CAE sets the AEG_CNT field in the dispatch module by driving the count value on *disp_aeg_cnt<17:0>*, which remains static for a given personality.

8.3.2.6 Dispatch stall

The custom personality can stall the dispatch of new instructions by asserting the *disp_stall* signal. The *disp_stall* signal is registered in the dispatch module, so instruction dispatch is stalled one clock cycle after *disp_stall* is asserted.

8.3.2.7 CAE Idle Status

The CAE must indicate idle status via the *disp_idle* signal. This signal must indicate true idle status—all pending memory operations in the CAE must be processed before *disp_idle* goes high.

8.3.2.8 Exceptions

Exceptions are sent by the custom personality on the *disp_exception<15:0>* bus. Bit 0 is defined to be the unimplemented instruction exception (see Section 0). This exception can be set by the personality as well as by the dispatch module. When any of the *disp_exception* signals are asserted, the exception is logged in the AES register in the dispatch interface. If the exception is not masked in the AEC register, a trap is sent back to the host processor. The program can clear the exception by writing the AES register. The AES register is automatically cleared when a new program is dispatched to the coprocessor.

8.3.2.9 Signal Definitions

The table below lists all the signals that connect the dispatch interface to the custom personality.

Signal Name	*Type*	*Description*
disp_inst<5:0>	input	Custom instruction to CAE
disp_inst_vld	input	CAE instruction valid. Instruction must be latched by the CAE when disp_inst_vld is high.
disp_aeg_idx<17:0>	input	AEG index for AEG read or write
disp_aeg_rd	input	AEG read. AEG index must be latched by the CAE when active.
disp_ret_data<63:0>	output	CAE AEG read data returned to AEH.
disp_ret _data_vld	output	CAE AEG read data valid
disp_aeg_wr	input	AEG write. AEG index must and data must be latched by the CAE when active.
disp_aeg_data<63:0>	input	AEG write data
disp_stall	output	CAE instruction dispatch stall. When high, the dispatch module stalls the instruction path to the CAE.
disp_idle	output	CAE is idle – all memory operations must be complete
disp_aeg_cnt<17:0>	output	Number of AEG registers implemented in the CAE.
disp_exception<15:0>	output	CAE exception

Table 4 – Dispatch Interface Signal Definitions

8.3.3 Memory Interface

The memory interface provides the CAE with access to both coprocessor and host memory. Each of the AEs is connected to each of the MCs (Memory Controllers) as shown in Figure 10.

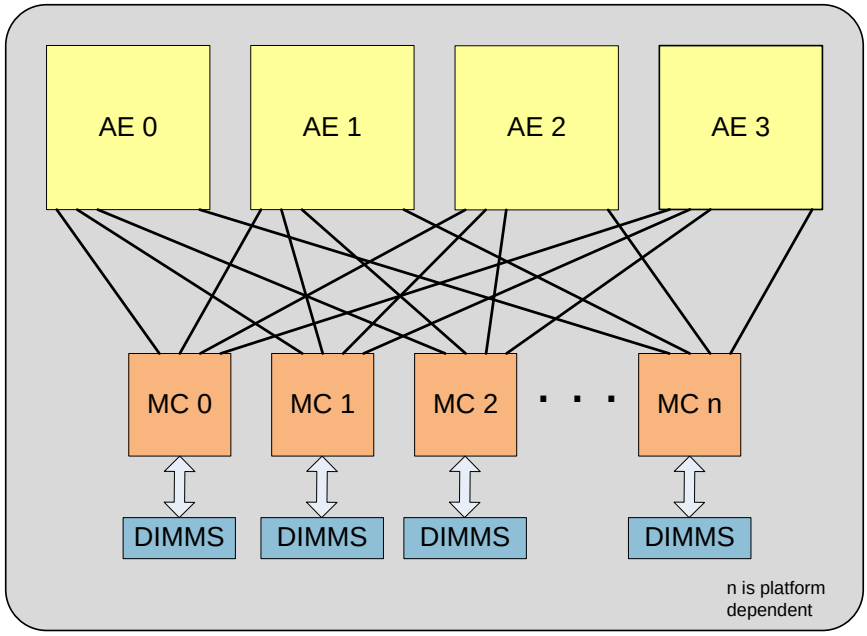

Figure 10 – Coprocessor AE Memory Connections

Convey provides a Memory Controller Interface (MCIF) which connects the CAE to the MCs providing the CAE access to both coprocessor and host memory. The complexity in the link between the AE and the MC is hidden from the custom personality by the MCIF.

The MCIF provides memory ports to the custom personality. A crossbar is used to connect each port of the custom personality to every MC. It allows the personality to maintain an abstracted view of memory, since the address decode and request/response routing is handled by the crossbar.

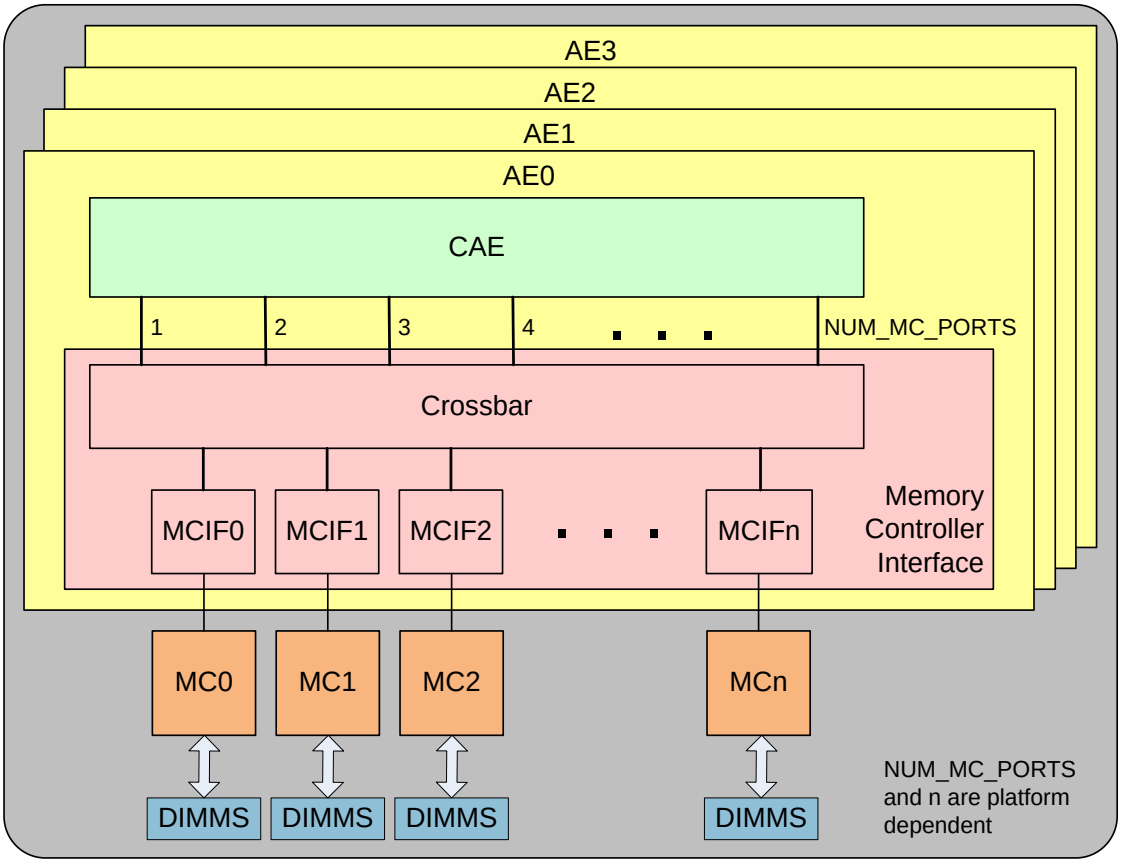

Figure 11 – PDK Memory Interface

8.3.3.1 Platform Specific Memory Subsystem

There are several differences in the HC and Wolverine memory subsystems. These differences are summarized in Table 5.

Feature / Characterisic	*Platform*	
	HC	*Wolverine*
Number of MC Interface Ports	Configurable from 1 - 16	Configurable from 1 - 8
Multi-Quad Word Access	Not Supported	Host or coprocessor memory

Feature / Characterisic	*Platform*	
	HC	*Wolverine*
Atomic Support	HC-1: None HC-2: ATOM_EXCH only	Support for: ATOM_ADD ATOM_EXCH ATOM_AND ATOM_OR
Maximum Memory Bandwidth (1)	76.8GB/sec	42.6Gb/sec
Memory Access Optimization	8 byte (host or coprocessor)	64 byte (host or coprocessor)

(1) Theorectical maximum

Table 5 – Platform Memory Subsystem Comparison

8.3.3.2 Signal Interface to Custom Personality

The signals between the AE personality and the MC interface for each memory port are divided into the following categories:

- Request Ports
- Response Ports
- Write Flush

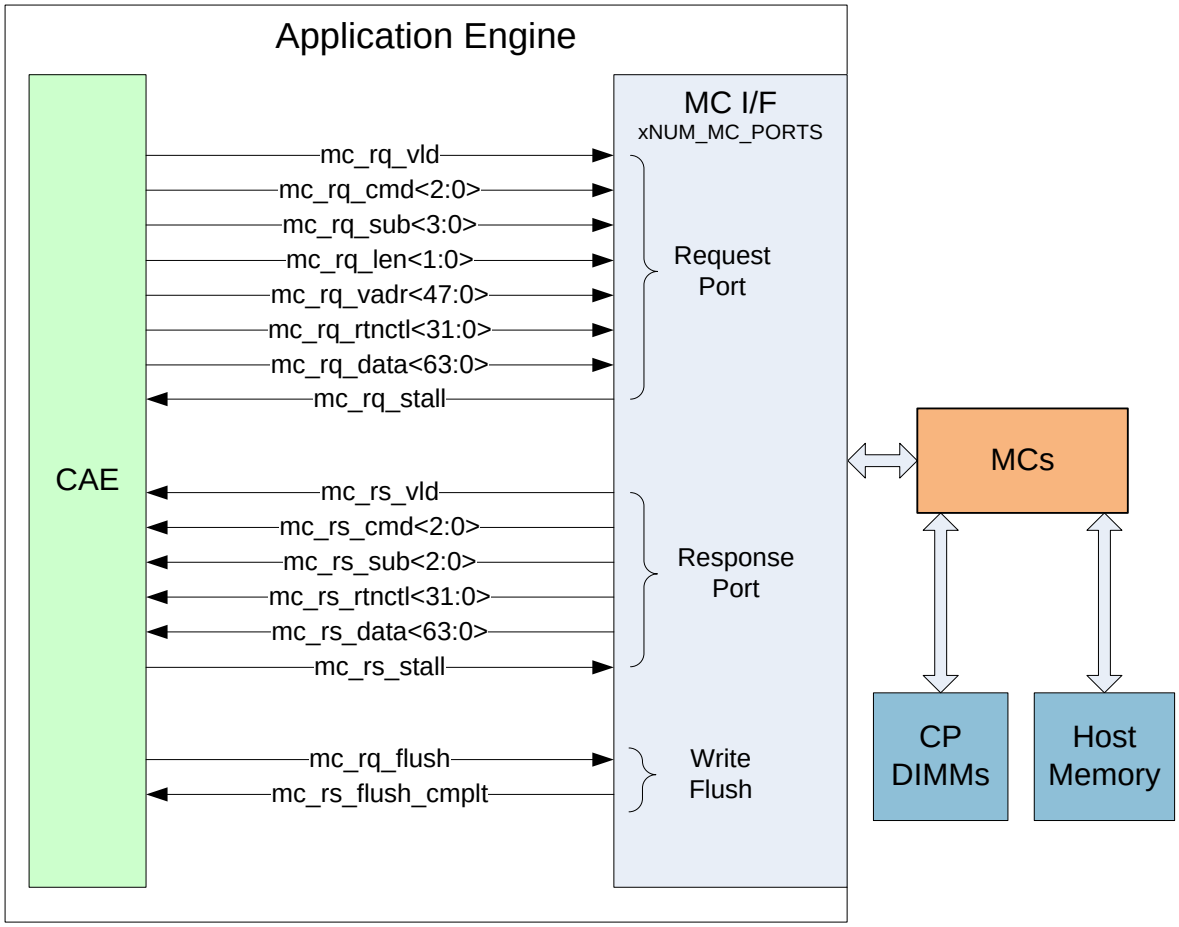

Figure 12 –Memory Interface Port Signal Interface

8.3.3.2.1 MC Interface Signal Definitions

MC Interface signals are defined in the tables below. Tables are divided by request/response interface port.

Signal Name	*Type*	*Description*
mc_rq_vld	output	Valid request indication
mc_rq_cmd<2:0>	output	Request type
mc_rq_sub<3:0>	output	Request type subcommand
mc_req_size<1:0>	output	Request size
mc_rq_vadr<47:0>	output	48-bit virtual address
mc_rq _rtnctl<31:0>	output	Request return control (width is programmable)
mc_rq_data<63:0>	output	Request data
mc_rq_flush	output	Write flush request
mc_rq_stall	input	Stall requests to this memory interface port

Table 6 – Memory Interface Port Request Signal Definition

Signal Name	*Type*	*Description*
mc_rs_vld	input	Valid response indication
mc_rs_cmd<2:0>	input	Response type
mc_rs_sub<3:0>	input	Response type subcommand
mc_rs _rtnctl<31:0>	input	Response return control (width is programmable)
mc_rs_data<63:0>	input	Response data
mc_rs_flush_cmplt	input	Write flush complete
mc_rs_stall	output	Stall responses to this memory interface port

Table 7 – Memory Interface Port Response Signal Definition

The width of the request and response return control is set using the MC_RTNCTL_WIDTH parameter.

Parameter Name	*Description*
MC_RTNCTL_WIDTH	Width of the mc_rq_rtnctl and mc_rs_rtnctl busses. 1 – 32 (default = 32)

Table 8 – MC_RTNCLT_WIDTH Parameter

8.3.3.2.2 Memory Ports

The number of memory interface ports used by the personality is selectable using the NUM_MC_PORTS parameter as shown in Table 9. Unused memory interface ports are tied off, conserving FPGA resources.

Parameter Name	*Description*
NUM_MC_PORTS	1 – 8 (Wolverine) 1 – 16 (HC)

Table 9 – NUM_MC_PORTS Parameter

All memory interface port signals in the personality are vectored to enable the use of generated loops inside the personality. An example is shown below:

```
genvar i;
generate for (i=0; i<16; i=i+1) begin : fp
    vadd add (
       .mc_rq_vld      (mc_rq_vld[i]),
       .mc_rq_cmd      (mc_rq_cmd[i*3 +: 3]),
       .mc_rq_size     (mc_req_size[i*2 +: 2]),
       .mc_rq_vadr     (mc_rq_vadr[i*48 +: 48]),
       .mc_rq_data     (mc_rq_data[i*64 +: 64]),
       .mc_rq_rtnctl   (mc_rq_rtnctl[i*32 +: 32]),
       .mc_rs_stall    (mc_rs_stall[i]),
       .mc_rq_stall    (mc_rq_stall[i]),
       .mc_rs_rtnctl   (mc_rs_rtnctl[i*32 +: 32]),
       .mc_rs_data     (mc_rs_data[i*64 +: 64]),
       .mc_rs_vld      (mc_rs_vld[i]),
    );
end endgenerate
```

8.3.3.3 Memory Requests

A memory request is sent to an MC interface on the request port. A request can be sent every clock cycle unless the MC interface has stalled requests by asserting *mc_rq_stall*.

A request is sent when *mc_rq_vld* is active. The request type is encoded on the *mc_rq_cmd<2:0>* and *mc_rq_sub<3:0>* busses. The *mc_req_size<1:0>* field indicates whether it is a byte, word, double-word or quad-word request. The length, command type, subcommand type, 48-bit address, write data and return control signals are valid on the same cycle as the request valid indication.

8.3.3.3.1 64 Bit or Sub 64-bit Stores

64-bit or sub-64-bit stores to memory are done by driving *mc_rq_vld* while *mc_rq_cmd<2:0>* is 2 and *mc_rq_sub<3:0>* is 0. The appropriate 64-bit data value is driven on the *mc_rq_data<63:0>* bus. Four request sizes are supported, byte, word, double-word and quad word, determined by the *mc_rq_len<1:0>* field. For store sizes less than quad word, the data must be correctly aligned on the data bus and all other bits are don't-cares. If write control is used, the control data is driven on *mc_rq_rtnctl<31:0>.* This field is used by the custom personality to uniquely identify request/response pairs.

Request Type	*mc_rq_cmd <2:0>*	*mc_rq_sub <3:0>*	*mc_req_len <1:0>*	*Reponse(1)*
Write Byte	2	0	0	WR_CMP
Write Word	2	0	1	WR_CMP
Write Long	2	0	2	WR_CMP
Write Quad Word	2	0	3	WR_CMP

(1) See Section 8.3.3.4 for details on memory responses

Table 10 – Traditional Write Requests

8.3.3.3.2 Multi-Quadword Stores

Multi-quadword accesses can be used to maximize memory bandwidth in some scenarios. Refer to Section 8.3.3.1 for platform specific memory information.

Multi-quadword stores to memory require multiple quad word write cycles to a single memory interface port. The cycles may be consecutive cycles or may have unused cycles between them, but other memory access can not occur between cycles associated with 64 byte accesses.

During each cycle of the multi-quadword write, *mc_rq_vld* is driven active, while *mc_rq_cmd<2:0>* is 6, *mc_rq_sub<3:0>* contains the total number of quad words to be written and *mc_req_len* is 3. The appropriate 64-bit data values are driven on the *mc_rq_data<63:0>* bus during each cycle with the first cycle containing the starting quadword offset. The first request cycle's virtual address bits[5:3] indicate the starting quadword offset. The personality should increment bits[5:3] each cycle. If write control is used, the control data is driven on *mc_rq_rtnctl<31:0>* and is the same for each cycle of

the multi-quadword access. This field is used by the custom personality to uniquely identify request/response pairs.

Request Type	*mc_rq_cmd <2:0>*	*mc_rq_sub <3:0>*	*mc_req_len <1:0>*	*Reponse*[2]
Write – Multi-quadword[1] (N requests)	6	N (# of Quad Words)*	3	WR_CMP (1 response)

(1) Platform dependent. See section Section 8.3.3.1 for platform specific memory information.

(2) See Section 8.3.3.4 for details on memory responses

Table 11 –64 Byte Write Requests

8.3.3.3.3 64 Bit or Sub 64 Bit Memory Loads

64 bit or sub 64 bit loads from memory are done by driving *mc_rq_vld* while *mc_rq_cmd<2:0>* is 1 and *mc_rq_sub<3:0>* is 0. Loads from memory use the 32-bit control bus (*mc_rq_rtnctl<31:0>*) to send read control information. The 32 bits of this data are stored by the MC and returned as *mc_rs_rtnctl<31:0>*. This field may be used by the custom personality to uniquely identify request/response pairs.

For load requests, the four request sizes are supported, byte, word, double-word and quad word, determined by the *mc_req_size<1:0>* field.

Request Type	*mc_rq_cmd <2:0>*	*mc_rq_sub <3:0>*	*mc_req_len <1:0>*	*Reponse*[1]
Read Byte	1	0	0	RD_DATA
Read Word	1	0	1	RD_DATA
Read Long	1	0	2	RD_DATA
Read Quad Word	1	0	3	RD_DATA

(1) See Section 8.3.3.4 for details on memory responses

Table 12 – Read Requests

8.3.3.3.4 Multi-Quadword Memory Loads

Multi-quadword accesses can be used to maximize memory bandwidth in some scenarios. Refer to Section 8.3.3.1 for platform specific memory information.

Multi-quadword loads from memory utilize a single request cycle. Multi-quadword loads from memory are done by driving *mc_rq_vld* while *mc_rq_cmd<2:0>* is 7. Loads from memory use the 32-bit control bus (*mc_rq_rtnctl<31:0>*) to send read control information. The 32 bits of this data are stored by the MC and returned as *mc_rs_rtnctl<31:0>*. This field may be used by the custom personality to uniquely identify request/response pairs.

Request Type	*mc_rq_cmd <2:0>*	*mc_rq_sub <3:0>*	*mc_req_len <1:0>*	*Reponse*[2]
Read – Multi-quadword[1] (1 request)	7	0	3	RD64_DATA (8 responses)

(1) Platform dependent. See section 8.3.3.1 for platform specific memory information.

(2) See Section 8.3.3.4 for details on memory responses

Table 13 – 64 Byte Read Requests

8.3.3.3.5 Atomic Requests

Support of atomics varies with the different Convey Platforms. Refer to Section 8.3.3.1 for platform specific information.

Atomic instructions perform read modify write operations to coprocessor memory location. A target address and operation is specified. The value of the target address before the operation is returned. The value after the operation is stored in the target address.

Atomic requests are done by driving *mc_rq_vld* while the target coprocessor memory location is specified in the *mc_rq_adr<63:0>* and the atomic request is encoded in *mc_rq_cmd<2:0>* and *mc_rq_sub<3:0>*. The *mc_req_size<1:0>* is set to 3. The *mc_rq_data<63:0>* bus is used to pass the value that is used to perform the atomic operation on the memory location.

Atomic operations use the 32-bit return control bus, *mc_rq_rtnctl<31:0>* to send control information. The 32 bits of this data are returned as *mc_rs_rtnctl<31:0>*. This field is used by the custom personality to uniquely identify request/response pairs

Request Type	*mc_rq_cmd <2:0>*	*mc_rq_sub <3:0>*	*mc_req_len <1:0>*	*Repons*[2]
ATOM_ADD[1]	5	0	3 QUAD	ATOMIC_DATA
ATOM_EXCH[1]	5	2	3 QUAD	ATOMIC_DATA
ATOM_AND[1]	5	7	3 QUAD	ATOMIC_DATA
ATOM_OR[1]	5	8	3 QUAD	ATOMIC_DATA

(1) Platform dependent support. See Section 8.3.3.1.

(2) See Section 8.3.3.4 for details on memory responses

Table 14 – Atomic Requests

8.3.3.3.6 Request Stalls

The stall indication, *mc_rq_stall* is sent from the MC interface to stall requests from the personality. When a stall asserts, the personality must stop sending requests within two cycles to avoid overflowing buffers in the MC interface.

8.3.3.4 Memory Responses

Responses are returned for all requests types. Response data is always returned aligned at byte 0, regardless of the address offset within the 64-bits. For instance, a single byte load from offset 0x2 will be returned in the least-significant byte, and the other 7 bytes of data are invalid.

Responses are valid from the MCIF on every cycle that *mc_rs_vld* is asserted. The response type is encoded in *mc_rs_cmd<2:0>* and *mc_rs_sub<2:0>*, as shown in the table below. The *mc_rs_rtnctl* bus is set to the corresponding request *mc_rq_rtnctl* bus. This field may be used by the custom personality to uniquely identify request/response pairs.

Response Type	*mc_rs_cmd <2:0>*	*mc_rs_sub <3:0>*	*mc_rs_len <1:0>*
RD_DATA	2	0	rq_rtnctl<31:0>
WR_CMP	3	0	rq_rtnctl<31:0>
ATOMIC_DATA[1]	6	ms_rq_sub<3:0>	rq_rtnctl<31:0>
RD64_DATA[1] (8 resposes)	7	Quad Offset	rq_rtnctl<31:0>

(1) Platform dependent. See Section 8.3.3.1 for platform specific memory information.

Table 15 – Memory Reponses

8.3.3.4.1 Read Response (RD_DATA)

Read responses are returned to the personality when read requests are complete. The read response is valid when *mc_rs_vld* is asserted and *mc_rs_cmd<2:0>* is 2. The read data, *mc_rs_data<63:0>* is valid on the same cycle. The response control bus, *mc_rs_rtnctl<31:0>* contains the *mc_rq_rtnctl<31:0>* of the corresponding request.

8.3.3.4.2 Write Response (WR_CMP)

Write responses are returned to the personality, when write requests are complete. The response is valid when *mc_rs_vld* is asserted and *mc_rs_cmd<2:0>* is 3. The response control bus, *mc_rs_rtnctl<31:0>* contains the *mc_rq_rtnctl<31:0>* of the corresponding request.

Note, a single write response is returned to the personality, when a write 64 request is complete.

8.3.3.4.3 Atomic Response (ATOMIC_DATA)

Support of atomics varies with the different Convey Platforms. Refer to Section 8.3.3.1 for platform specific information.

Atomic responses are returned to the personality when atomic requests are complete. An atomic response is valid when *mc_rs_vld* is asserted and *mc_rs_cmd<2:0>* is 6. On the same cycle *mc_rs_sub<2:0>* will reflect mc_*rq_sub<2:0>* of the corresponding request. The atomic data, *mc_rs_data<63:0>* is valid on the same cycle. The response control bus, *mc_rs_rtnctl<31:0>* contains the *mc_rq_rtnctl<31:0>* of the corresponding request.

8.3.3.4.4 Multi-Quadword Read Response (RD64_DATA)

Multi-quadword accesses can be used to maximize memory bandwidth in some scenarios. Refer to Section 8.3.3.1 for platform specific memory information.

Multi-quadword read responses are sent to the personality when a multi-quadword read is executed. 8 quadword read responses are sent to the personality, in quadword order for each multi-quadword read request (offset 0 first and offset 7 last). The responses can be interleaved with other unrelated responses.

Multi-quadword read responses are valid when *mc_rs_vld* is asserted and *rs_cmd<2:0>* is 7. The quadword offset is contained in the *rs_sub<2:0>* bus and the read data, *mc_rs_data<63:0>* is valid on the same cycle. The response control bus, *rx_rtnctl<31:0>* contains the *rq_rtnctl<31:0>* of the corresponding request.

8.3.3.4.5 Response Stall

Responses from the MC can be stalled by the custom personality using the *mc_rs_stall* signal. The MC can send up to eight additional responses after the stall is asserted, so this signal should be connected to a FIFO “almost full” signal to allow space for responses in-flight.

8.3.3.5 Coprocessor Memory Order

The Convey Coprocessor supports a weakly ordered memory model. All coprocessor memory reads and writes are allowed to proceed independently to memory through different interconnect paths to provide increased memory bandwidth. Requests that follow different interconnect paths may arrive in a different order than they were issued. The weak memory ordering applies between load and store requests, as well as between requests of the same type.

The CAE must assure proper memory ordering as required by the personality. Since there is a response associated with all requests, the CAE can determine if a request has completed. Response ordering is also available and is described in Section 8.3.6

In some circumstances, the order of memory accesses must be assured when accessing different coprocessor memory locations. For example, writing data to a buffer, then setting a valid bit. In this case a write flush may be set between the data buffer writes and the valid bit write, to assure the data is written prior to the valid bit.

8.3.3.5.1 Write Flush

A write flush is requested by asserting a flush request following the last write that needs to be complete. After a write flush has been issued at an MC interface request port, the corresponding flush complete signal will be asserted for one cycle when all outstanding writes to that port have completed.

The write flush function is useful when the AE stores to a buffer in memory, then follows those stores with a store to a location in memory that indicates the data is valid. To prevent the "valid" store from passing data stores, a write flush is sent, and the valid store is held until the flush is complete.

In some designs, performance can be increased by allowing multiple write flushes to be outstanding at one time. A maximum of 4 flushes can be outstanding at each MC interface. The CAE is responsible for keeping track of the number of outstanding flushes.

8.3.3.6 Memory Bandwidth

In order to optimize memory bandwidth, a memory request should be made from every AE to every MC port on every cycle (unless stalled by the MC interface). Requests should be distributed across the memory. Section 8.3.8.1. contains information on the memory address scheme and should be refered to to determine efficient strides for non-random memory accesses.

8.3.4 CSR Interface

The Control Status Register (CSR) interface provides alternate communication path to the AE. Since this path is independent of the instruction dispatch path from the host processor, it can be useful in debugging, by allowing visibility into internal state of the personality, even when the application is busy.

CSRs are connected to a ring through CSR agents. CSR agents provide a register interface to the custom personality, as shown in Figure 13.

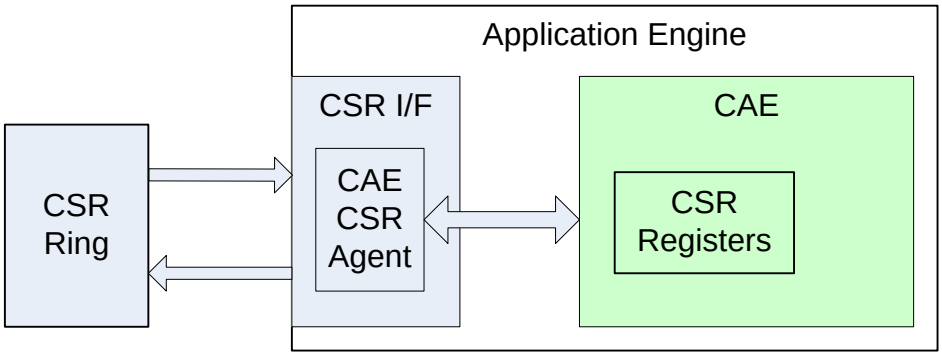

Figure 13- CAE CSR Registers

Instantiation of the appropriate CSR logic in the infrastructure is configured using the CSR_IF parameter, as shown in Table 16

Variable Name	*Description*
CSR_IF	OFF – No CSRs in personality REG – Instantiate a single CAE agent in the infrastructure. (Figure 13))

Table 16 – CSR_IF Variable

8.3.4.1 CSR Register Interface

The CSR Register Signals are shown below.

Signal Name	*Type*	*Description*
csr_wr_vld	Input	Single cycle pulse that indicates the csr_address and csr_wr_data are valid for a write request
csr_rd_vld	Input	Single cycle pulse that indicates the csr_address is valid for a read request
csr_address<15:0>	Input	Zero-based address for CSR read or write request. Only valid if csr_wr_vld or csr_rd_valid is asserted
csr_wr_data<63:0>	Input	Write data for write requests. Only valid if csr_wr_vld is asserted.
csr_rd_data<63:0>	Output	Read response data for read requests, latched when the csr_ack signal is asserted.
csr_ack	Output	Asserted for a single cycle to indicate successful completion of a read. Latches csr_rd_data on read requests.

Table 17 – CSR Agent Register Signal Definitions

A write to a CSR register occurs when *csr_wr_vld* is asserted. The data on *csr_wr_data<63:0>* is written to the CSR register at *csr_address<15:0>*.

A read of a CSR register occurs when *csr_rd_vld* is asserted indicating *csr_address<15:0>* has a valid read address. *csr_ack* is asserted indicating read data is valid on *csr_rd_data<63:0>.*

8.3.4.2 Address Range for CSRs

Address space is reserved for CAE CSR registers. The CSR address space is platform dependent.

8.3.4.2.1 HC Platform Address Range

The user addressable CSR address space on the HC platform is 0x8000 – 0xFFFF for each AE.

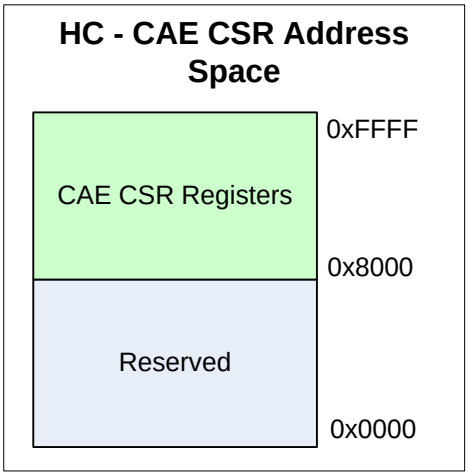

Figure 14 – HC CSR Address Space

8.3.4.2.2 Wolverine Platform Address Range

The user addressable CSR address space on the Wolverine platform is 0x6000 – 0x7FFF and is divided between the AEs as shown in Figure 15.

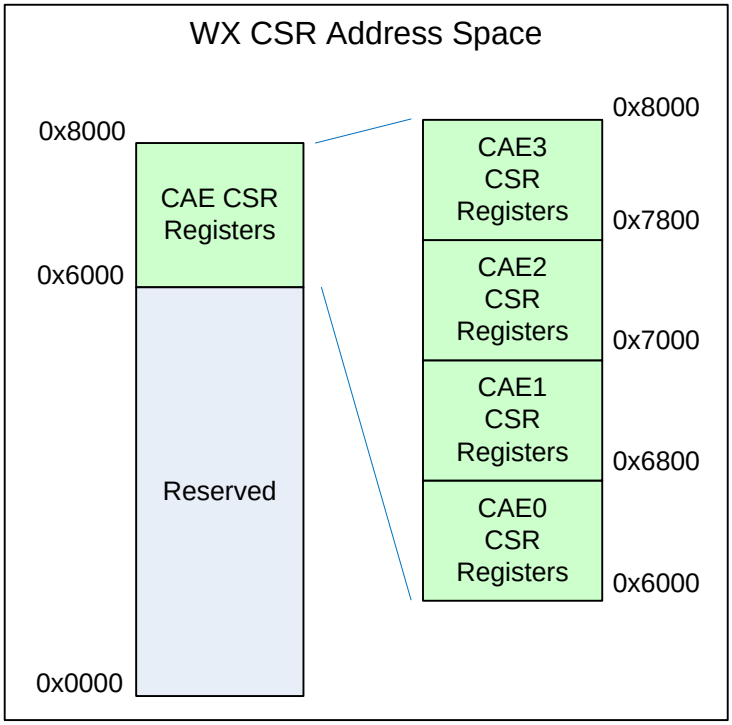

Figure 15 – Wolverine CSR Address Space

8.3.4.3 Accessing CSR Registers

8.3.4.3.1 HC Platform

PDK CSR Registers are accessed from the host.

The host communicates with the MP FPGA via telnet. To establish the telnet connection, type the following commands at the host shell prompt:

```
/opt/convey/sbin/mpip 2543&
telnet localhost 2543
```

Once the telnet session is established, commands can be sent to the MP.

The following commands are used to read and write the AE CSR Registers:

```
ae_csr_write ae<0-3> <csr address> <value> {<mask>}
ae_csr_read ae<0-3> <csr address>
```

When finished, use the telnet escape sequence (normally CTRL-]q) to exit the telnet session, then kill the mpip program.

To kill the mpip program first find the process ID

```
pgrep mpip
```

This will return the ID of the process. Then kill the process using

```
kill <process ID>
```

An example application program which opens a MP socket, accesses CSR registers then closes the socket is available in the PDK release in the following location:

```
/opt/convey/pdk/<rev>/<platform>/diag
```

The example program (*ae_perf.c*) is used to access the performance monitor registers described in Section 8.3.7.1. This program can be used as a starting point in developing other applications accessing CSRs.

8.3.4.3.2 WX Platform

PDK CSR Registers are accessed from the host, using the **wxcsr** utility found in **/opt/convey/utils.**

```
./wxcsr <coproc> [cmd] [device] [offset] [value]
```

where

coproc: the physical coprocessor ID (0-63 or wxpfwa0-63). Default is 0. The physical ID can be found using the **wxinfo** utility.

cmd: command type

read: **read|rd|rdb**

write: **write|wr|wrb**

device: specifies the AE containing the CSR. Valid entries are **ae[0-3]** to specifiy a specific AE or **ae*** for all AEs.

offset: specifies the offset in the AE's CSR space. Valid entries are 0-0x7FF the offset will be converted to a byte address (*8) unless the command is rdb or wrb.

value: 64 bit write data. Writes are only allowed to a single AE at a time (device can not be ae*)

Example:

To read the first CSR in CAE0:

```
/opt/convey/utils/wxcsr wxpfwa0   rd ae0 0
```

results in

```
AE 0 0x0 (0x0): 0x0000000000000000
```

or to read the first CSR in all CAEs

```
/opt/convey/utils/wxcsr wxpfwa0 rd ae* 0
```

results in

```
AE 0 0x0 (0x0): 0x0000000000000000
AE 1 0x0 (0x0): 0x0000000000000000
AE 2 0x0 (0x0): 0x0000000000000000
AE 3 0x0 (0x0): 0x0000000000000000
```

To write all ones to the first CSR in CAE0

```
/opt/convey/utils/wxcsr wxpfwa0 wr ae0 0 0xffffffff
```

8.3.5 General Resources

8.3.5.1 Static Inputs

Signal Name	*Type*	*Description*
i_aeid<1:0>	input	AE Identification (0 – 4)
csr_31_31_intlv_dis	input	Interleave Configuration Indication (boot option for HC only) 0: 31/31 Interleave 1: Binary Interleave

Table 18 - Static Signals

The *csr_31_31_intlv_dis* signal reflects the interleave boot option. This signal is only applicable to the HC platform.

8.3.5.2 Clocks

The clocks, shown in Table 19 are and are available to the custom personality.

Signal Name	*Type*	*Description*
clk	input	General system clock
clkhx	input	General half system rate clock
clk2x	input	General 2x system rate clock

Table 19 – Clock Signals

The clocks rates shown in Table 20 are generated by a PLL in the AE and are the default clock rates. A PLL can be instantiated in the personality generating a different frequency for *clk* and the other associated clocks. See Section 8.3.5.2.1 for more information on configuring other clock rates.

Signal	*HC Default Rates*	*WolverineDefault Rates*
clk	150 MHz	167 MHz
clkhx	75 MHz	83.5 MHz
clk2x	300 MHz	334 MHz

Table 20 – Default Clock Rates

The Dispatch and Memory and AE – AE Interfaces to the custom personality use *clk*. The CSR Interface to the custom personality used *clkhx*.

Clocks are phase aligned on the rising edge. These clocks are run through global clock buffers and fanned out across the chip by the Xilinx tools. Any asynchronous crossings must be handled by the FPGA designer.

8.3.5.2.1 Other Clock Rates

The designer can specify clock rates other than the default using the CLK_PERS_FREQ parameter. The CLK_PERS_FREQ parameter can be set in the makefile to a value between 50 and 250 (50 – 250 MHz) that is divisible by 25. This value will be the frequency of the *clk* input. The *clkhx*, *clk2x* and *clk4x* inputs are based on the *clk* input.

Variable Name	*Description*
CLK_PERS_FREQ	50 – 250: Frequency of the *clk* input

Table 21 – CLK_PERS_FREQ Variable

The CLK_PERS_FREQ also enables the asynchronous interface between the personality and the PDK interfaces, as shown in Figure 16. The asynchronous interface handles the asynchronous crossings of all signals in the Dispatch, Memory AE-AE and CSR

Interfaces. Note the interface handles crossings between the specified *clk* and the default *clk* for the Dispatch, Memory and AE-AE Interfaces and the specified *clkhx* and the default *clkhx* for the CSR Interface. The AE-AE and CSR Asynchronous Interfaces are only instantiated when the optional CSR or AE-AE Interfaces are implemented.

On the HC-1/HC-2 platforms, the resources available to generate and distribute additional clocks are the same resources used for the AE-AE Interface, so only one of these features can be implemented. HC-1ex, HC-2ex and Wolverine support both asynchronous clocks and the AE-AE Interface.

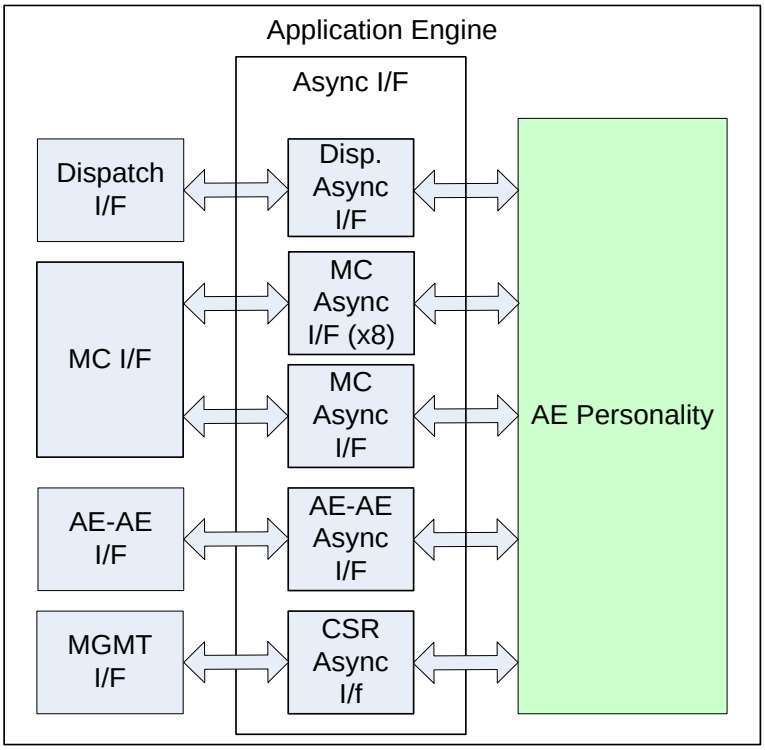

Figure 16 - Asynchronous Interface

8.3.5.3 Resets

The reset signals below are available to the custom personality.

Signal Name	*Type*	*Description*
reset	input	Power up reset, synchronous to *clk*

Table 22 – Reset Signals

The *reset* signal is on a global clock network (connected through a BUFG global clock buffer). This signal can be connected directly to the reset input of flip-flops, or it can be locally fanned out through registers If it is registered locally, it should be connected to the reset input of the flip flop (not the D input) as shown in the figure and the sample code below.

```
Always @(posedgd clk) begin
        if (reset)
                local_reset <=1
            else
                local_reset<=0
        end
```

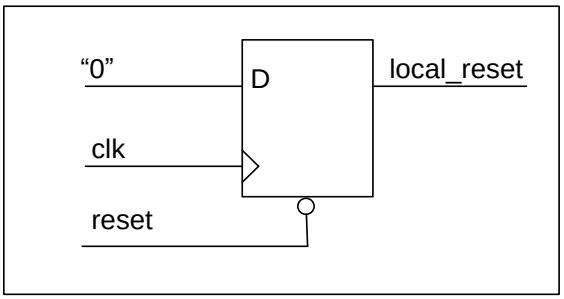

Figure 17 – Local Reset

8.3.5.3.1 AE Reset

The coprocessor cannot be reset without rebooting the server. The coprocessor memory system and host interface must always be available while the system is booted.

The AE may be reset without affecting the host, by

- forcing the image to reload, using the mpcache –f command.
- <ctrl>-C may be issued, while an application is running, to force the image to be reloaded and the AE to be reset.

The custom personality may generate a local reset to initialize between routines based on idle state or a start signal.

8.3.6 Optional MC Interface Functionality

The optional memory interface functionality is divided in two sections: response data ordering and request data ordering. Only functions needed should be instantiated to avoid using FPGA resources and possibly adversely affecting performance.

The diagram below shows the AE memory interface with the optional functionality:

- Response Data Ordering

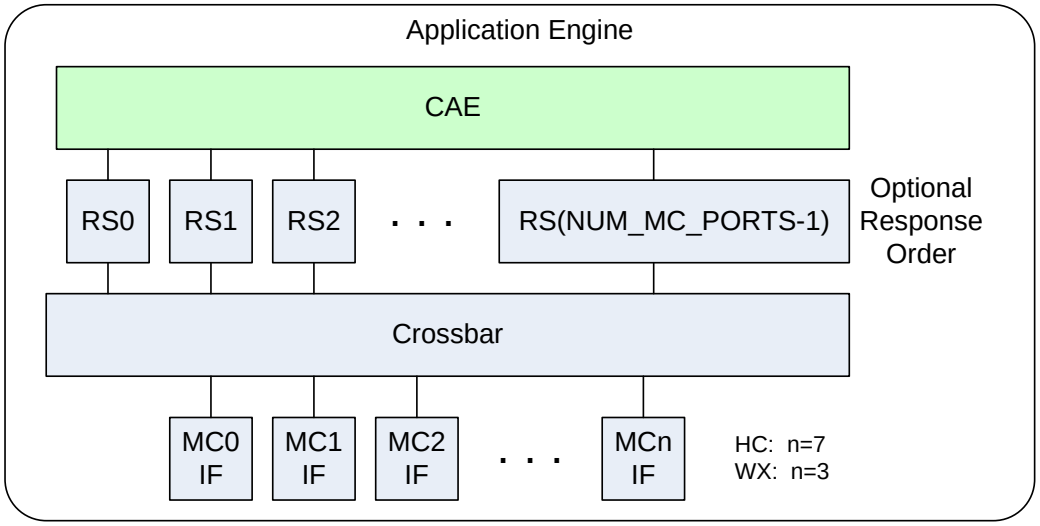

Figure 18 - Optional MC IF Functionality

8.3.6.1 Optional Response Order MC Interface

The optional response order module assures all responses from the associated request port are returned to the custom personality in order. The return control and read data returned from memory is queued and ordered before being returned to the personality. Order is only assured for requests from the same request port.

The response order module does not impact memory content. The order of reads and writes is not affected, so additional functionality may need to be implemented in the personality to assure request data order.

The response order module is instantiated using the MC_RSP_ORDER variable.

Variable Name	*Description*
MC_RSP_ORDER	0 – Disable the optional response order module (default). 1 – Instantiate the optional response order module

Table 23 – Response Order Variable

8.3.7 Diagnostic Resources

8.3.7.1 Performance Monitor

The optional Performance Monitor may be instantiated in the custom personality to provide data on memory bandwidth utilization. The Performance Monitor implements three sets of counters in CSRs located in the Convey Reserved space.

- Absolute counters are located at 0x4000. These counters count loads, stores and stalls for each port, while the CAE is active (!cae_idle)

- Latency counters are located at 0x4100. These counters hold the results of load and store latency calculations
- Histogram counters are located at 0x4200.

The Performance Monitor is instantiated using the PERFMON variable.

Variable Name	*Description*
PERFMON	0 – Disable the optional Performance Monitor (default). 1 – Instantiate the optional Performance Monitor

Table 24 – Performance Monitor Variable

After a dispatch has completed the application (ae_perf) found in

```
/opt/convey/pdk2/<rev>/<platform>/diag
```

can be run to provide memory bandwidth utilization for the dispatch.

8.3.8 Advanced Features

8.3.8.1 Memory System

8.3.8.1.1 HC Memory System

The Convey HC memory system has 1024 memory banks. The banks are spread across eight memory controllers (MCs). Each memory controller has two 64-bit busses, and each bus is accessed as eight sub busses (8-bits per sub bus). Finally, each sub bus has eight banks. The 1024 banks is the product of 8 MCs * 2 DIMMs/MC * 8 sub bus/DIMM * 8 bank/sub bus. The diagram below shows the coprocessor memory hierarchy.

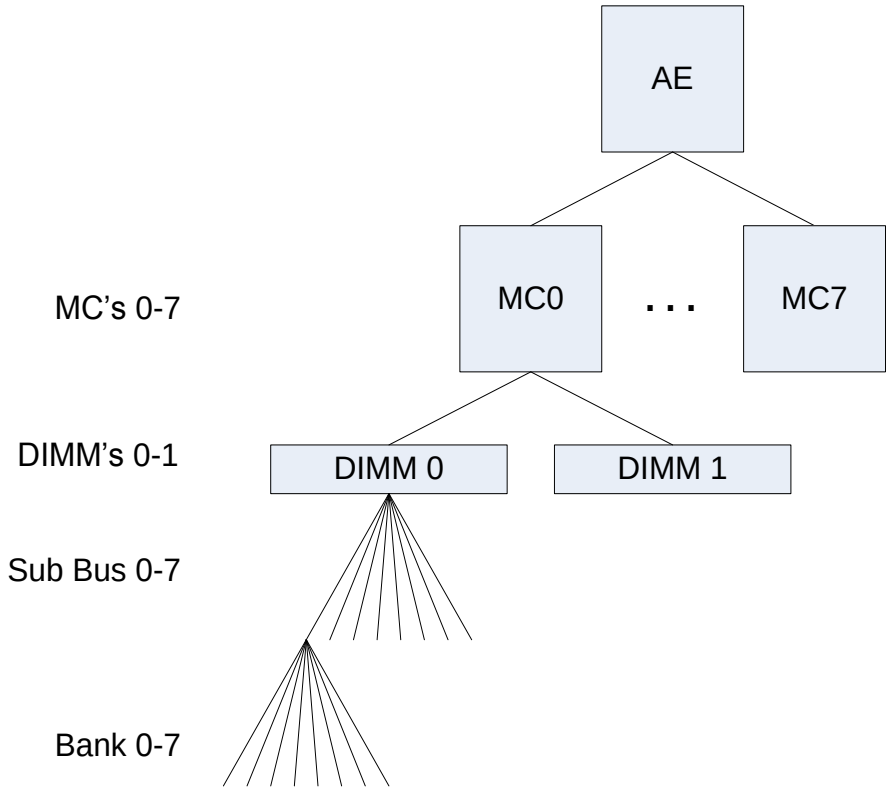

Figure 19 – HC Memory Hierarchy

Binary interleave is the recommended interleave mode for most applications. The HC platforms also support 31-31 interleaving. 31-31 Interleaving is covered under Advanced Features in Section 8.3.8.3.

8.3.8.1.2 HC Binary Interleave

Binary Interleave is the recommended option for most applications. When 31-31 interleaving is disabled, the 1024 banks are accessed with the following virtual address assignments:

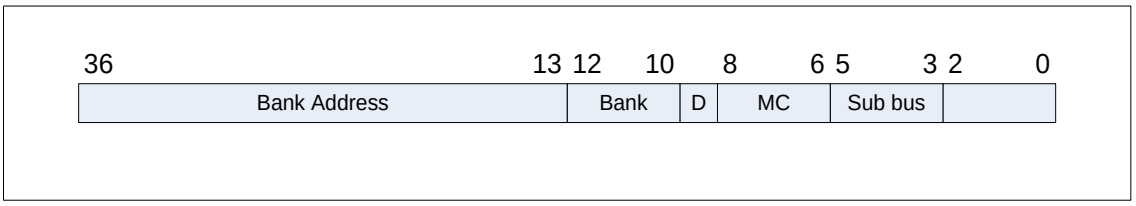

Figure 20 – HC Binary Interleave

Note that in the above figure, bit 9 is used to select which of the two DIMMs on a memory controller is to be accessed.

8.3.8.1.3 Wolverine Memory System

The Convey Wolverine memory system has 64 memory banks. The banks are spread across four memory controllers (MCs). Each memory controller has one DIMM, containing 2 ranks, and each rank has eight banks. The 64 banks is the product of 4 MCs * 1 DIMM/MC * 2 ranks * 8 banks/rank. The diagram below shows the coprocessor memory hierarchy.

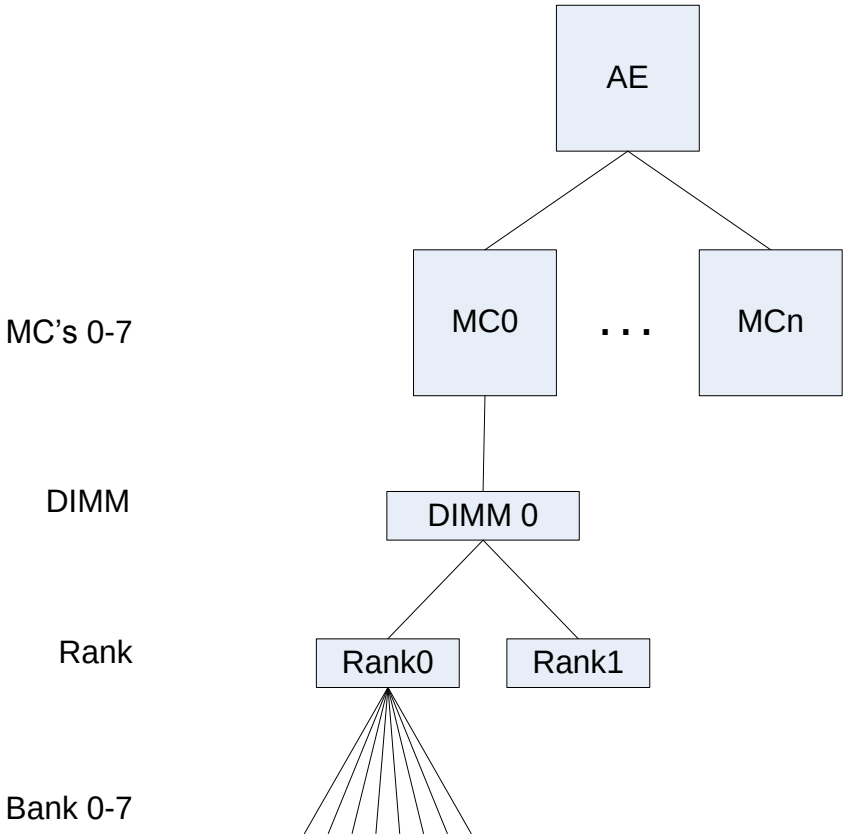

Figure 21 – Wolverine Memory Hierarchy

8.3.8.1.4 Binary Interleave

Binary Interleave is the recommended option for most applications. The 64 banks are accessed with the following virtual address assignments:

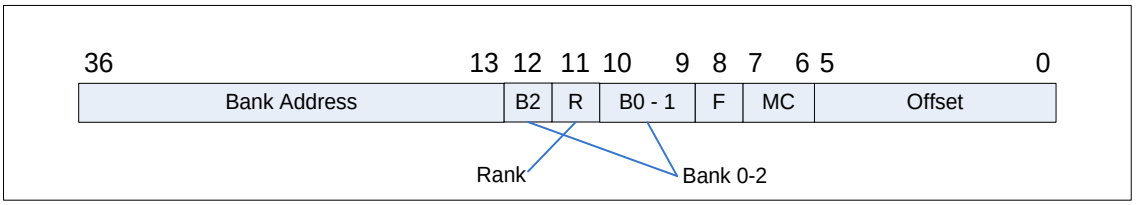

Figure 22 – Wolverine Binary Interleave

8.3.8.2 Optional Crossbar MC Interface

The optional crossbar is used to connect each memory interface port of custom personality to every MC interface. The crossbar allows the personality to maintain an abstracted view of memory, since the address decode and request/response routing is handled by the crossbar. The crossbar supports binary interleave.

The crossbar is enabled with the MC_XBAR variable.

Variable Name	*Description*
MC_XBAR	0 – Disable the optional crossbar 1 – Instantiate the optional crossbar (default)

Table 25 – Crossbar Variable

8.3.8.3 31/31 Interleave (HC Platform)

The 31/31 interleave scheme was defined to meet the following requirements:

1. Provide the highest possible bandwidth for all memory access strides, with a particular focus on power of two strides.
2. Keep each memory line (64-bytes) on a single memory controller.
3. Maintain the interleave pattern across virtual memory page crossings. This helps large strides where only a few accesses are to each page.
4. All virtual addresses must map to unique physical addresses.

The scheme uses a two level hierarchical interleave approach. The 1024 banks are divided into 32 groups of 32 banks each. The first interleave level selects one of 31 groups of banks. The second interleave level selects one of 31 banks within a group. Note that of the 32 groups of banks, one is not used. Similarly, one bank within each group of banks is not used. A prime number (31) of banks and groups of banks is used to maximize the sustainable memory bandwidth for as many different strides as possible, at the expense of wasting 6% of memory, and decreasing the peak memory bandwidth by 6%.

An optional memory crossbar with an optional 31/31 interface is available.

The interleave mode is a boot option. See the Convey System Administration Guide for instructions on configuring the system interleave.

8.3.8.3.1 31/31 Interleave Crossbar Option

Applications which access memory in power of two strides may achieve higher performance using 31/31 interleave. For these applications the 31/31 option can be instantiated in the crossbar.

The 31/31 option may also be instantiated when the system will be used for multiple applications supporting different interleave options.

The interleave option on the crossbar is instantiated with the MC_XBAR_INTLV variable as shown in Table 26. For the system to run 31/31 interleave, the boot option must also be set to 31/31 interleave.

Variable Name	*Description*
MC_XBAR_INTLV	0 – Disable the optional 31/31 option on the crossbar (default). 1 – Instantiate the optional 31/31 option on the crossbar

Table 26 – 31/31 Interleave Crossbar Variable

8.3.8.3.2 31/31 Interleave Simulation

The CNY_PDK_SIM_INTERLEAVE=<3131|binary> environment variable selects the interleave mode for simulation.

8.3.8.4 Optional AE-AE Interface

The AE-to-AE interface allows data to be transferred directly from one AE to another. The AE to AE interface is an optional interface. This section contains information on the AE – AE interface for the HC platform. The optional AE – AE Interface for Wolverine will be available in a future release.

Two types of AE – AE interfaces are available.

- A set of counter flowing rings
- Point to point connections for “next door AEs” (only available on the HC platforms)

The AE – AE Interface is enabled by setting the AE_AE_IF variable to 1. The AE_AE_IF variable enables all AE-AE links (ring and point to point).

Variable Name	*Description*
AE_AE_IF	0 – Disable the optional AE – AE Interface (default). 1 – Instantiate the optional AE – AE Interface

Table 27 – AE to AE Interface Variable

If the AE – AE interface is not used, AE_AE_IF is set to 0 and the PDK designer does not implement any logic for the AE – AE interface. If a subset of the available AE_AE links are implemented, the AE_AE_IF variable is set to 1 and the outputs of the unused interfaces are tied to 0. The default for the AE_AE_IF variable is 0.

Note 1: The AE-to-AE interface is not connected in the PDK sample personality.

Note 2: The AE to AE interface utilizes the same FPGA resources as the asynchronous clocking option for the custom personality on HC-1 and HC-2 only. See section 8.3.5.2.1 of this document for further information.

8.3.8.4.1 Physical connections

The AE FPGAs are physically connected with 2 counter flowing rings consisting of 8 signals. The HC platforms also support point to point next door links consisting of 19

signals to the previous AE, as shown in the figure below. A single set of next door links are supported on the HC-1 / HC-2, while two sets of next door links are supported on the HC-1ex / HC-2ex.

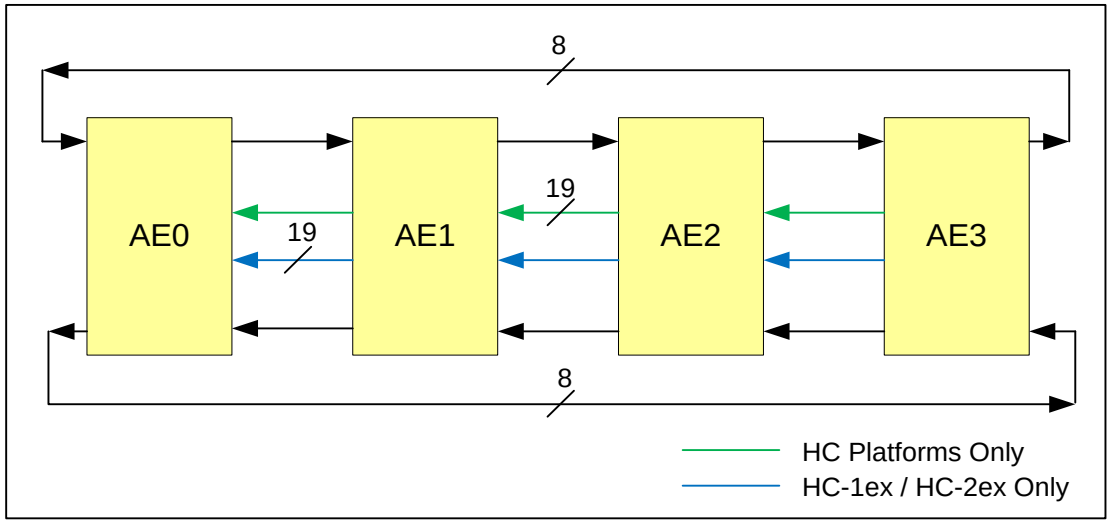

Figure 23 – AE to AE Interface Physical Connections on HC

8.3.8.4.2 AE – AE Ring Interface

The AE – AE Interface provides transparent transport for two counter flowing rings supporting data rates up to 600Mbps each. The PDK designer determines the protocol. CRC checking on the data and flow control are provided by the AE – AE Interface.

The diagram below shows the signal interface between the AE – AE Interface and the custom AE personality, for the counter flowing rings.

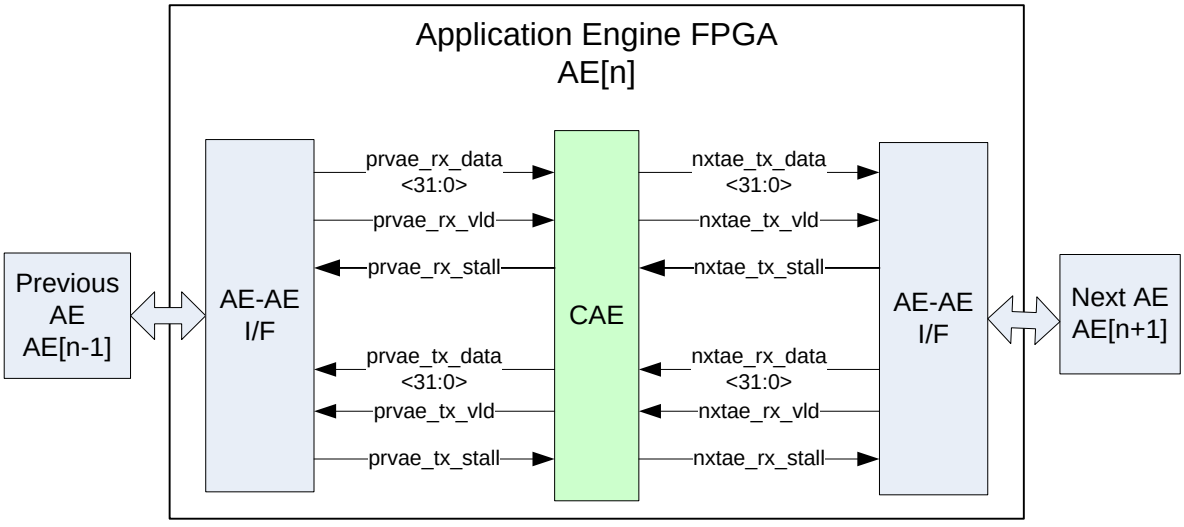

Figure 24 – AE to AE Loop Interface Diagram

Convey's AE-AE interface defines these signals as follows:

Signal Name	*Type*	*Description*
nxtae_rx_data<31:0>	input	Next AE receive data
nxtae_rx_vld	input	Next AE receive data valid
nxtae_rx_stall	output	Next AE receive stall
nxtae_tx_data<31:0>	output	Next AE transmit data
nxtae_tx_vld	output	Next AE transmit data valid
nxtae_tx_stall	input	Next AE transmit stall
prvae_rx_data<31:0>	input	Previous AE receive data
prvae_rx_vld	input	Previous AE receive data valid
preae_rx_stall	output	Previous AE receive stall
preae_tx_data<31:0>	output	Previous AE transmit data
preae_tx_vld	output	Previous AE transmit data valid
preae_tx_stall	input	Previous AE transmit stall

Table 28 – AE to AE Interface Ring Signal Definitions

8.3.8.4.3 Next Door Point to Point Interface

The AE – AE Interface provides transparent transport for point to point connections between “next door AEs” supporting data rates up to 247.5 Mbps each. The HC-1 / HC-2 supports a single set of point to point links and the HC-1ex / HC-2ex supports two sets of point to point links. The PDK designer determines the protocol. Parity generation and checking on the data and flow control are provided by the AE – AE Interface.

The diagram below shows the signal interface between the AE – AE Interface and the custom AE personality, for the counter flowing rings.

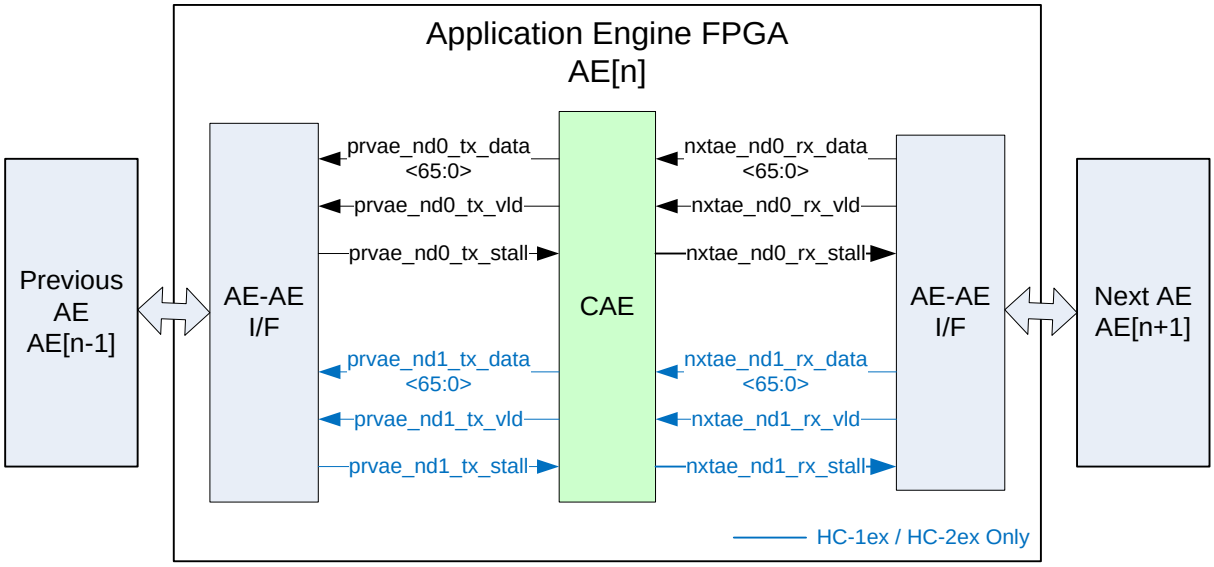

Figure 25 – HC Platform AE to Next Door AE Point to Point Interface Diagram

Signal Name	*Type*	*Description*
prvae_nd0_tx_data<65:0>	output	Previous AE next door transmit data
prvae_nd0_tx_vld	output	Previous AE next door transmit data valid
prvae_nd0_tx_stall	input	Previous AE next door transmit stall
nxtae_nd0_rx_data<65:0>	input	Next AE next door receive data
nxtae_nd0_rx_vld	input	Next AE next door receive data valid
nxtae_nd0_rx_stall	output	Next AE next door receive stall

Table 29 – HC Platform AE to Next Door Interface Signal Definitions

Signal Name	*Type*	*Description*
prvae_nd1_tx_data<65:0>	output	Previous AE next door transmit data
prvae_nd1_tx_vld	output	Previous AE next door transmit data valid
prvae_nd1_tx_stall	input	Previous AE next door transmit stall
nxtae_nd1_rx_data<65:0>	input	Next AE next door receive data
nxtae_nd1_rx_vld	input	Next AE next door receive data valid
nxtae_nd1_rx_stall	output	Next AE next door receive stall

Table 30– Additional AE-to-Next Door AE Interface Signal Definitions (HC-1ex / HC-2ex)

8.3.8.4.4 AE – AE Transmit

To send a transaction to another AE, assert *_tx_vld while driving *_tx_data. The *_tx_stall signal is provided as a backpressure mechanism to stall transactions from the custom personality. When stall is asserted, the personality must stop sending data within two cycles to avoid overflowing buffers in the AE-AE interface.

8.3.8.4.5 AE – AE Receive

When *_rx_vld is asserted, a receive transaction is valid and *_rx_data should be latched by the CAE. The custom personality must accept the transaction. The CAE can stall data from another AE by asserting *_rx_stall. The custom personality must be able to accept five additional transactions after asserting *_rx_stall.

8.4 FPGA Tool Flow

The PDK tool flow allows the user to simulate the custom personality and to produce personality FPGA images using the Xilinx tools.

8.4.1 PDK Revisions

Convey-supplied components of the PDK, such as interface RTL, simulation libraries and user constraints, are provided in /opt/convey/pdk2/<*rev*>/<*platform*>, where *rev* is a dated revision and *platform* is the Convey platform. The user can point to a different revision of the PDK using the CNY_PDK_REV variable.

8.4.2 PDK Project

A PDK project contains all personality-specific components, RTL code (Verilog or VHDL) for the personality, the software model of the AE, and any user constraints to be used during the Xilinx build. A typical project will have the following directory structure and components.

```
<project_name>/
      Makefile.include – top-level include file for project settings
```

```
app/ - host application
    cpapp.s – HC only, coprocessor routine
    gdbregisters – HC only
    Makefile
    run
    UserApp.c – host source code
phys/ - physical implementation of FPGA
    Makefile – runs Xilinx tools to synthesize, place and route
    *.ucf files – Xilinx ISE user constraints files (HC optional)
    *.xdc files – Xilinx Vivado constraint files (WX optional)
sim/ – hardware and software simulation
    Makefile – builds software sim .exe, runs hardware sim
    *.cpp - AE software simulations source files
    sc.config – optional config file for simulation
verilog/ - Verilog source files for the personality
    *.v
```

8.4.3 PDK Variables

Several variables are used to configure the PDK.

Variable	*Description*
CNY_PDK	Points to PDK installation, typically /opt/convey/pdk2
CNY_PDK_REV	Point to the dated revision of the PDK or “latest”

The installation location and revision are set using the following command

```
source <CNY_PDK>/<CNY_PDK_REV>/settings.(c)sh
```

Variable	*Description*
CNY_PDK_PLATFORM	Selects the target platform (hc-1, hc-1ex, hc-2, hc-2ex, WX-690, WX-2000)
CNY_PDK_HDLSIM	Synopsys or Mentor

The platform and HDL variables can be set in the Makefiles or in the environment.

8.4.4 Simulation

Convey provides an HDL simulation environment to enable FPGA developers to simulate custom personalities with the rest of the coprocessor system. Bus functional models of

each of the Convey interfaces are provided so that the user can focus on the custom personality.

The Convey Coprocessor architecture simulator contains a VPI (Verilog Procedural Interface) interface to an HDL simulator. This allows the actual user application, running on the architecture simulator, to provide the stimulus for the hardware simulation of the FPGA.

The hardware simulation environment provided by Convey is useful in debugging system-level interfaces. Convey recommends that developers first verify custom logic at the module level before integrating into the AE.

During hardware simulation, the device under test (DUT) is the entire FPGA, consisting of the user-developed personality as well as the Convey-supplied hardware interfaces. For the HC platforms this includes a single AE and for the WX platforms all AEs are included.

8.4.4.1 Running Simulation

Convey provides a script (**`runsim`**) located in the **`sim`** directory to easily simulate the hardware with the host application (**`UserApp`**) found in the **`app`** directory using default the default simulation options.

The user can also run *runpdksim* directly or modify the *runsim* script if other options are desired. A complete list of options is available using *runpdksim* with the –help option.

```
runpdksim -help
```

8.4.5 Xilinx Tool Flow

As part of the PDK, Convey provides an environment to ease the process of implementing the AE with Xilinx tools. The Xilinx development process and tool flow is documented in detail in the Xilinx ISE Software Manuals, which can be downloaded from www.xilinx.com.

8.4.5.1 Xilinx Project File

The Xilinx tools use a project file <design>.prj to identify the source files used in the design. Many of these are provided by Convey in the PDK. Others are developed by the user and exist in the user's project tree.

Convey's makefile automatically generates a project file if one doesn't already exist in the project phys directory. The script first looks in the default project verilog folder <project>/verilog and its subdirectories for files with a '.v' extension. It also looks for user code in the <project>/coregen directory. Using make file variables the user can specify alternate locations for source files.

If new source files are created after the project is created, the user should run

```
make clean
```

in the project phys directory, so the project file will be regenerated to include the new source files.

8.4.5.2 User Constraints

Convey provides all constraints necessary for Convey interfaces. These constraints are automatically used when the physical Makefile is run. Constraints for the custom personality should be added by the user in the project phys directory

8.4.5.2.1 Wolverine

Each *.xdc file should be added to the Makefile using the USER_SYNTH_XDC and USER_PLACE_XDC variables

For constraints used for sysnthesis

```
USER_SYNTH_XDC = <file1>.xdc
```

For constraints used for sysnthesis

```
USER_PLACE_XDC = <file2>.xdc
```

8.4.5.2.2 HC

Each *.ucf file should be added to the Makefile using the UCF_FILES variable:

```
UCF_FILES += <file1>.ucf
UCF_FILES += <file2>.ucf
```

Note that the += must be used to add each file to the list of Convey supplied files, rather than over-writing them.

8.4.5.3 HC-1 / HC-2 FPGA Resources

The Application Engines in the HC-1 and HC-2 platforms are implemented in Xilinx Virtex 5 LX330 FPGAs. The required Convey hardware interfaces—the dispatch interface, CSR interfaces and MC interfaces—use about 10% of the available logic resources and about 25% of the block rams.

The dispatch interface is located in the center of the chip. The MC interfaces are on the left and right sides of the FPGA, and the MC CSR interfaces are in the corners of the part. The figure below shows the FPGA floor plan.

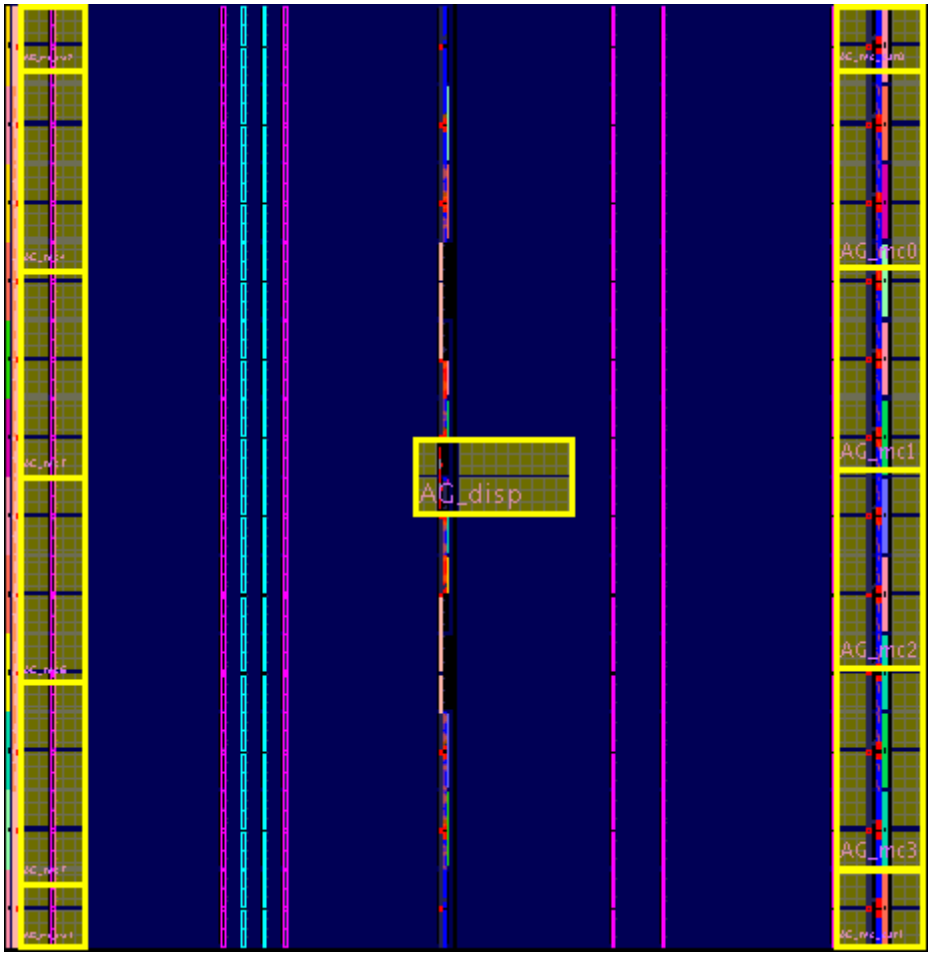

Figure 26 – AE FPGA Floor Plan for HC-1 / HC-2

8.4.5.4 HC-1ex / HC-2ex FPGA Resources

The Application Engines in the HC-1ex and HC-2ex platforms are implemented in Xilinx Virtex 6 LX760 FPGAs. The required Convey hardware interfaces—the dispatch interface, CSR interfaces and MC interfaces—use about 6% of the available logic resources and about 12% of the block rams.

The dispatch interface is located in the center of the chip. The MC interfaces are in vertical columns just to the left and right of the center of the FPGA, along with the MC CSR interfaces. The figure below shows the FPGA floor plan.

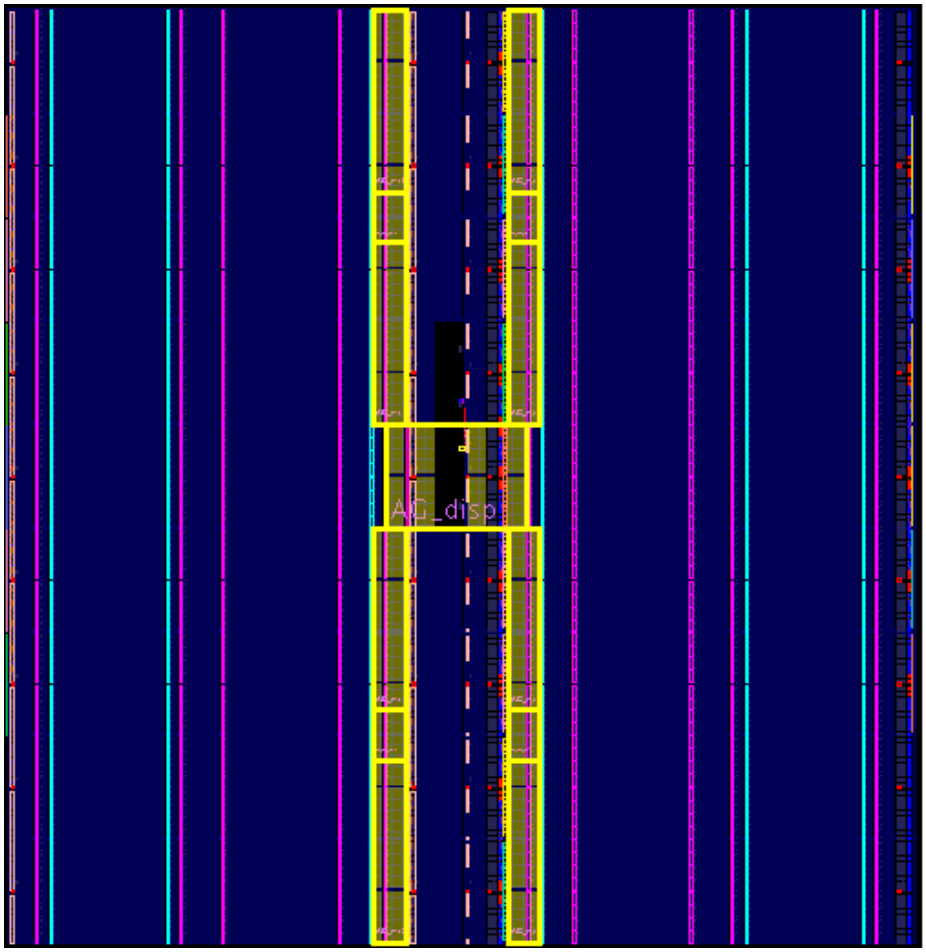

Figure 27 – AE FPGA Floor Plan for HC-1ex / HC-2ex

8.4.6 Installing the FPGA Image

When a physical build is run, an FPGA bitfile (ae_fpga.bit) is created in the project phys directory. To package the bitfile to be installed on the Convey system, run 'make release' in the phys directory. This creates a release directory at the same level as the project. Inside the release directory is a dated directory with a ae_fpga.tgz file. This file should be copied to the appropriate personality directory in /opt/convey/personalities on the Convey server. The FPGA image must be called "ae_fpga.tgz.

Anytime a new .tgz file is placed in the personality directory, the MP cache must be flushed to force a reload of the FPGA. The following command flushes the MP cache:

```
/opt/convey/sbin/mpcache -f
```

8.4.7 Debugging with ChipScope

8.4.7.1 HC

The PDK supports remote debugging with Xilinx ChipScope 11.2 or greater.

8.4.7.1.1 Inserting the Core

To insert a ChipScope core, run the ChipScope Core Inserter (inserter.sh or inserter for ISE 12+) from the <project>/phys directory with a routed netlist. Use "cae_fpga.ngc" for the input design netlist and "cae_fpga.ngo" for the output design netlist. When the core is inserted, typing "make" in the phys directory will reimplement the design from the ngdbuild step. The makefile will automatically insert a ChipScope core if the file "cae_fpga.cdc" exists..

8.4.7.1.2 Load the FPGA

When running ChipScope it is necessary to load the FPGA prior to running the analyzer. The following commands flush the cache, add the new image and load the image to be tested.

```
/opt/convey/sbin/mpcache --flush
/opt/convey/sbin/mpcache --add -S <personality>
/opt/convey/sbin/mpcache --load -S <personality>
```

8.4.7.1.3 Running the Analyzer

Once the FPGA with ChipScope is installed on the Convey system, the ChipScope analyzer (analyzer.sh) can be run on the development system and can remotely connect to the Convey system. Follow the steps below to run the analyzer remotely:

1. On the Convey host server, start the remote ChipScope server:

   ```
   /opt/convey/sbin/mpChipScope start
   ```

2. On the development system, run the ChipScope client (ChipScope version 11.2 or greater is required for remote connectivity).

   ```
   analyzer.sh
   click on the "JTAG Chain" menu and select "Open Plug-in"
   In the Plug-in Parameters box, enter
           'xilinx_xvc host=[host IP]:2542
   disableversioncheck=true'
   ```

3. A pop-up window displays 15 available FPGA devices. Click "OK" and wait for the analyzer to start.

4. Import the CDC file for one or more of the AE FPGAs (Devices 2, 3, 9 and 10):

   AE0 → DEV 10

   AE1 → DEV 9

   AE2 → DEV 2

   AE3 → DEV 3

9 Sample Custom Personality

This chapter describes the sample personality that can be used as a reference for new personality designs.

9.1 Overview

As part of the PDK package, Convey has developed a sample custom personality that can be used as a reference design. The sample personality was designed to be very simple while using all of the required hardware interfaces inside the FPGA. In addition the sample custom personality uses the read order cache and has the option to use the crossbar and the asynchronous interface.

Convey's sample personality includes a memory-to-memory add instruction that adds values stored in one block of memory to the values stored in another block of memory. The array of results is stored back in memory. Details of the instructions and machine state registers are described below.

The sample personality contains 16 copies of a functional block, each of which adds a portion of the memory storing the operands. The functional blocks can directly connected to the memory controllers so that each block only performs operations on values in its attached memory or the optional crossbar may be used.

The sample personality can be used as a starting point for new PDK personalities. The sample personality can be found in each release of PDK in the following location:

```
/opt/convey/pdk2/<rev>/<platform>/examples
```

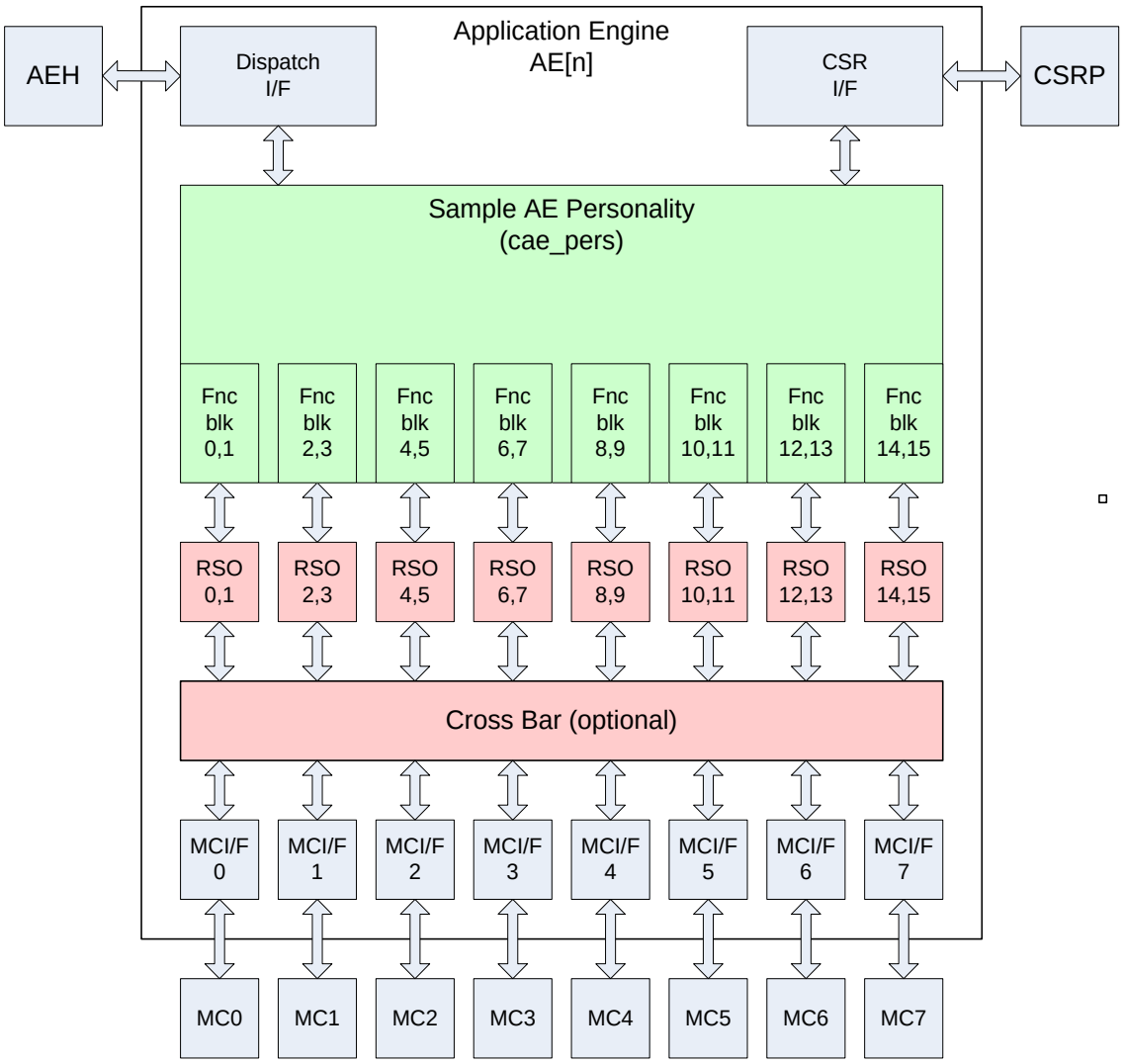

Figure 28 - Sample Personality Block Diagram

9.2 Sample Personality Machine State Extensions

The sample personality machine state extensions (AEC, AES and AEG registers) are described in section 0. AEG registers are defined as shown in the table below:

AEG Index	*Register Name*	*Description*
0	MA1	Memory Address 1
1	MA2	Memory Address 2
2	MA3	Memory Address 3

AEG Index	*Register Name*	*Description*
3	CNT	Count – number of add operations/elements in each memory array
30-33	SAE[3:0]	Application Engine Sums

Table 31 – Sample Personality Registers

9.2.1 MA1 Register

The MA1 register stores the base memory address of the first array of operands to be added using the memory-to-memory add instruction.

9.2.2 MA2 Register

The MA2 register stores the base memory address of the second array of operands to be added using the memory-to-memory add instruction.

9.2.3 MA3 Register

The MA3 register contains the base address of memory into which the results of the memory-to-memory add instructions are stored.

9.2.4 CNT Register

The CNT register stores the count of add operations to be performed by the memory-to-memory add instruction.

9.2.5 SAE[3:0] Register

The SAE[3:0] register array contains the sum of each AE. The sum from AE0 is stored in AEG[30], the sum of AE1 is stored in AEG[31], AE2 in AEG[32] and AE3 in AEG[33].

9.3 Exceptions

The sample personality adds a few exceptions to those defined in the PDK infrastructure. The table below describes those exceptions.

Exception Name	*AEE Bit*	*Description*
AEUAE	2	Unaligned address. The base address for one of the arrays was not aligned on MC0 (if the crossbar is not enabled)
AEROE	1	Result overflow.
AESOE	1	Sum overflow.

Exception Name	*AEE Bit*	*Description*
AEINE	1	Invalid memory interleave.

Table 32 – Sample Personality Registers

9.4 Sample Personality Instructions

This section presents the sample AE personality instruction set extensions. The custom defined instructions for the sample personality are defined below:

Instruction		*Operation Description*
CAEP00	Imm18	Memory-to-Memory Add, Masked

Table 33 – Sample Personality Instructions

9.4.1 CAEP00 – Memory-to-Memory Add

CAEP00 Immed18

9.4.1.1 Description

This custom instruction adds two arrays of 64-bit integers stored in memory and writes the array of results back to memory using signed quad word integer data format.

The instruction's m bit is used to determine whether all AEs enabled in the AEEM.IEEM mask field should participate (m=1), or the instruction directly specifies the single AE that participates in the operation (m=0). The AE specified in the directed instruction must also be enabled in the AEEM.IEEM register. The immediate field is unused.

9.4.1.2 Pseudo Code

```
for (i = 0; i < 4; i += 1) {
    if (AEEM.IEEM[i]) {
        if ((m == 1) || (i == ae)) {
            for (j = 0; j < AE[i].AEG[CNT]; j += 1) {
                effa1 = AE[i].MA1 + j*8;
                effa2 = AE[i].MA2 + j*8;
                effa3 = AE[i].MA3 + j*8;
                op1 = MemLoad( effa1, 64 );
                op2 = MemLoad( effa2, 64 );
                res = op1 + op2;
                MemStore( effa3, 64, res );
```

```
                    sum += res;
                }
            }
        }
        AE[ae].SAE[30+ae] = sum;
    }
```

9.4.1.3 Exceptions

Unaligned Address (AEUAE)

Result Overflow (AEROE)

Sum Overflow (AESOE)

Invalid Interleave (AEINE)

9.5 Running the Sample Application

9.5.1 Copy Sample AE and Sample Application

The vector add sample personality and application are installed with the PDK RPM in **`/opt/convey/pdk2/<rev>/<platform>/examples/cae_vadd64`**. The project should be copied to a working directory such as the user's home directory.

9.5.2 Build the Sample AE and Sample Application

Everything needed to run the sample application, including the application itself and the software model of the sample personality, is included.

To build the user application, type `**`make`**` in the **`app`** directory. This generates an executable called *UserApp.exe* that can be run in simulation or on the hardware.

9.5.3 Run the Application

Scripts in the **`app`** directory allow the user to run the application in simulation or on the hardware:

```
./run          # runs the simulation using the software model
```

Convey-specific environment variables are defined in the Convey HC PDK Programmers Guide or the Convey Wolverine PDK Programmers Guide.

A PDK Variables

The variables listed below are included in the project Makefile as needed.

Variable Name	*Platform*	*Description*
CNY_PDK	HC and Wolverine	Location of PDK
CNY_PDK_REV	HC and Wolverine	Version of PDK
CNY_PDK_PLATFORM	HC and Wolverine	Target Platform (hc-1, hc-1ex, hc-2, hc-2ex, WX-690, WX-2000)
CNY_PDK_SIG	HC and Wolverine	Personality number. 5 digit personality identifier. (Default is 65000)
CNY_PDK_NICK	HC and Wolverine	Personality name. May contain alpha, numeric and _ characters. Must start with and alpha character. The name is case sensitive Default is the personality number.
SUPPLEMENTAL_POWER	Wolverine	0 – Supplemental power is not needed (default) 1 – Supplemental power required
DISABLE_CP_MEM	Wolverine	0 – Memory controllers needed (default) 1 – Memory controllers not needed
AE_AE_IF	HC and Wolverine	0 – Disable the optional AE – AE Interface (default). 1 – Instantiate the optional AE – AE Interface
MC_READ_ORDER	HC and Wolverine	0 – Disable the optional read order cache (default). 1 – Instantiate the optional read order cache
MC_STRONG_ORDER	HC Only	0 – Disable the optional strong order cache (default). 1 – Instantiate the optional strong order cache

Variable Name	*Platform*	*Description*
MC_XBAR	HC and Wolverine	0 – Disable the optional crossbar. 1 – Instantiate the optional crossbar (Default)
MC_XBAR_INTLV	HC Only	0 – Disable the optional 31/31 option on the crossbar (default). 1 – Instantiate the optional 31/31 option on the crossbar
PERFMON	HC and Wolverine	0 – Disable optional performance monitor (default) 1 – Instantiate the optional performance monitor
CAE_CSR_ADR_MASK	HC and Wolverine	CSR Address Mask
CAE_CSR_SEL	HC and Wolverine	CSR Agent Address

B Customer Support Procedures

Email	support@conveycomputer.com
Web	Go to Customer Support at www.conveycomputer.com