Microsoft

Microsoft Project Catapult
Mt Granite Shell
Architectural Specifications

Date: July 15, 2017
Version: V1.2

Revision History

Revision/Date	Notes	Updated by
V1.0 06/30/2016	Initial release version	Dan Zhang
V1.1 12/21/2016	Added Catapult Software API section	Dan Zhang
V1.2 7/15/2016	Revamped document. Added details on SerialLite operation	Dan Zhang

1 Introduction

1.1 Overview

Mt Granite is a Field Programmable Gate Array (FPGA) card designed for use within Microsoft data centers. This document is the architectural specification of the **Mt Granite Academic Shell**, a hardware abstraction layer for the PCIe, DRAM, and serial FPGA links.

1.2 The Mt Granite Card

Mt Granite contains an Altera Stratix V 5SGSMD5H2F35 and two channels of DDR3, each providing 4GB for a total of 8GB. The card connects to the server via eight lanes of Gen3 PCI Express (PCIe), but also has four pairs of SerialLite III lanes, routed through two standard mini-SAS connectors.

In the standard data center configuration, custom cables attach to the two mini-SAS connectors of each server blade to connect them to other Mt. Granite cards in adjacent server blades to form a two-dimensional torus. Each of the two processors in the standard Catapult server can interact with the FPGA over the eight lanes of PCIe.

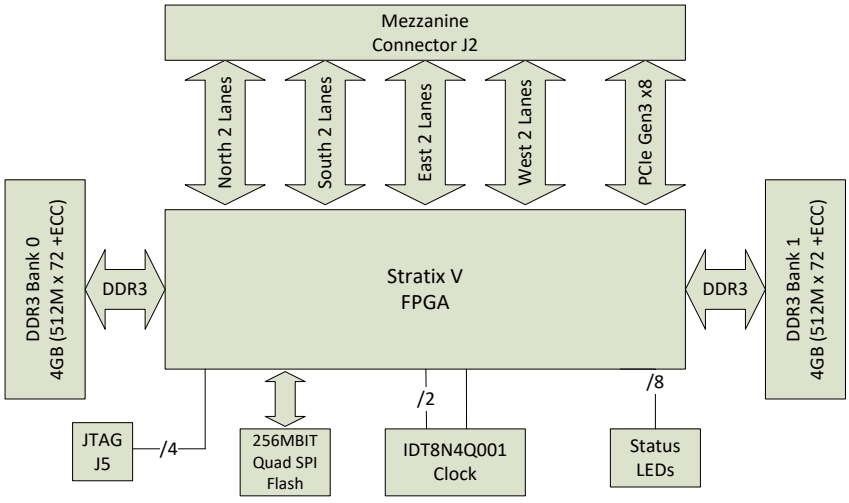

Figure 1: Block diagram of the Mt Granite FPGA board.

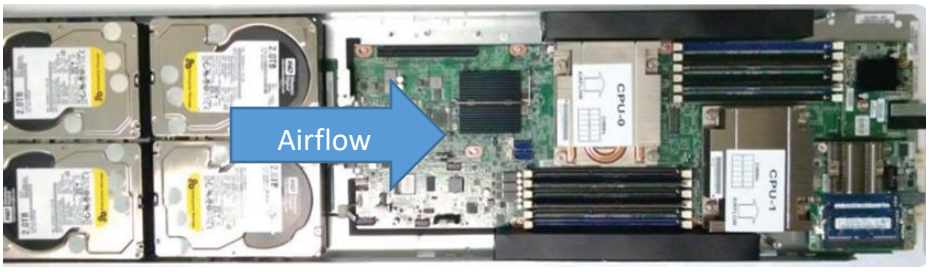

Figure 2: A diagram of the 1U, half-width server that hosts the FPGA board. The air flows from left to right, leaving the FPGA in the exhaust of both CPUs.

2 Overview

We conceptually divide an FPGA image into the **Shell** and the **Role** as shown in Figure 3. The Role is the application specific to a product or scenario, such as the Bing ranker. The Role is what most Catapult users will develop.

The Shell is a simple hardware abstraction layer for the components accessible from the FPGA. It is intended to insulate application writers from the low-level details of the platform, board, and components through standardized and easy-to-use interfaces. The Shell is the top-level module that directly interfaces to the FPGA's pins, and instantiates a set of IP blocks that provide the abstractions to the Role.

Changes to the Shell are expected to be infrequent; we expect no more than one or two rollouts per year. The Shell and the Role are expected to be compiled together statically. Thus, changes to either requires a full re-compilation of the bitstream.

Figure gives a high-level block diagram of the Shell and Role. The following components are in the Shell: (1) four point-to-point SerialLite III transceivers that connect to mini-SAS connectors, (2) a PCIe Gen3 x8 IP block connected to the host server, (3) two DDR3 memory controllers connected to two 4GB channels (for a total of 8GB) of on-board DRAM, and (4) various peripherals including flash reading/writing/configuration, clock generation, and temperature sensing.

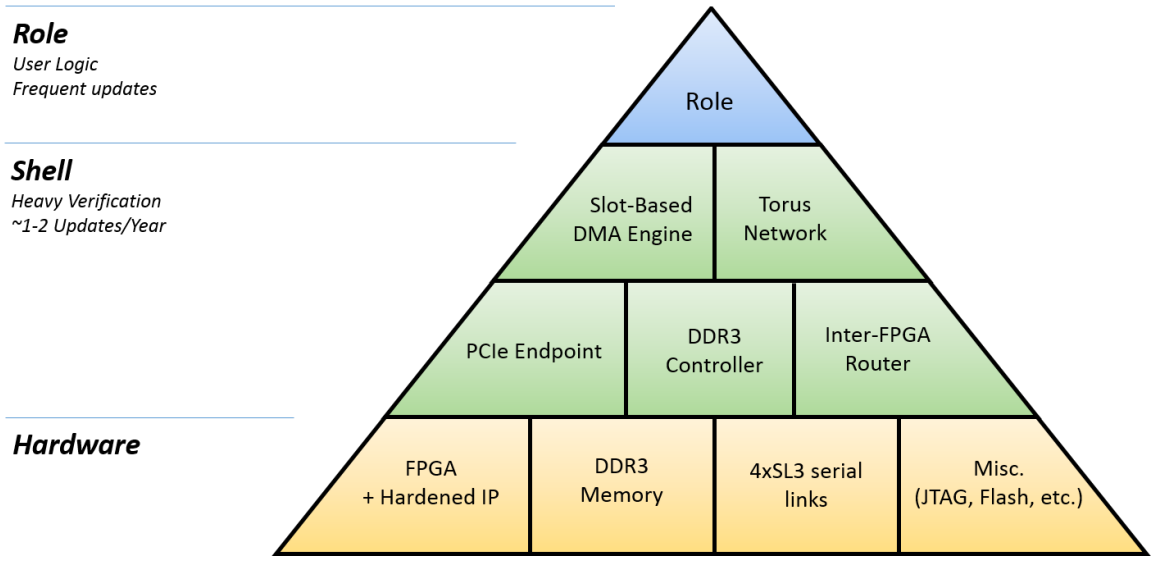

Figure 3: Mt Granite Role, Shell, and Hardware Overview

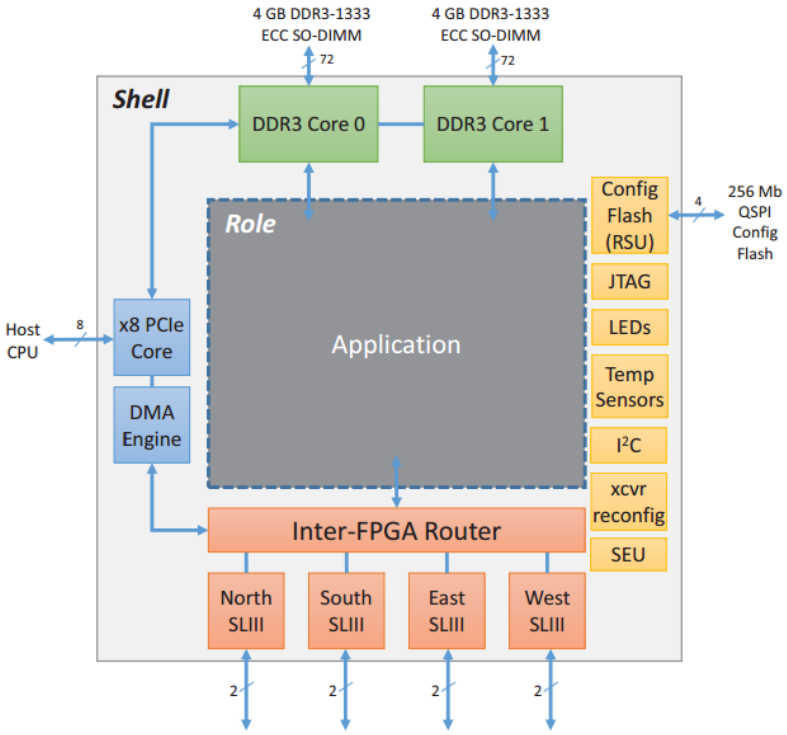

Figure 4: High-level Block Diagram of the Mt Granite Shell

2.1 SerialLite III

Each Mt Granite board has four bi-directional, point-to-point SerialLite III (SL3) lanes, enabling Mt Granite boards to directly communicate with other Mt Granite boards in a two-dimensional 6x8 torus network. Each logical SL3 lane is physically implemented with two pairs 6Gbps links in each direction, for a maximum bandwidth of 2*6Gbps = 12 Gbps (1.5GB/s) for each lane in each direction.

The Shell abstracts the low-level Altera SL3 MegaCore function, presenting a simple send/receive FIFO-based interface for each lane. However, **flow control is not implemented**: lanes receiving packets cannot backpressure. It's up to the user to create a proper backpressure scheme, or architecturally guarantee that the receiver will always have room for incoming packets. Each direction is split into two virtual channels through time-division multiplexing: the main data channel and an out-of-band channel (OOB), in which the OOB channel is allocated 1/16th of the bandwidth and is intended for implementing a credit-based flow control scheme.

2.2 PCI Express

PCI Express (PCIe) is the only means to communicate between the host system and its locally attached Mt Granite FPGA card. The Altera Stratix V D5 FPGA on Mt Granite includes a built-in Gen3x8 PCIe hard IP (HIP) block that enable the FPGA to send and receive PCIe Transaction Level Packets (TLP) to and from the CPUs' PCIe root complex.

To simplify communication between the host system and the FPGA, Microsoft has developed a **Slot-based DMA IP Block** layered on top of a single HIP. The slot-based architecture allows software running on the host to send and receive data to the FPGA at latencies in the order of microseconds, while exposing associated synchronization mechanisms.

In addition to DMA, the host software communicates with the FPGA using a set of pre-defined PCIe configuration registers. The PCIe register space is divided into three categories: (1) internal PCIe registers, (2) shell registers, and (3) soft registers. **Internal PCIe registers** are intended for use by the low-level FPGA driver stack and are required for FPGA-to-Host DMA transfers and synchronization. The **shell registers** expose a wide variety of status information specific to the shell only, including interface configuration (e.g. PCIe bypass, SL3 initialization), counters and statistics (e.g. number of packets received and sent), and health monitoring (e.g. number of bit errors detected in DRAM, FPGA junction temperature). The **soft registers** are intended to be specified by the Role and therefore not defined in the shell specification.

2.3 Memory Interface

The Mt Granite FPGA card incorporates two memory channels to the on-board 2x4GB of DDR3 memory, each channel being 512M x 64b + ECC, 533MHz bus. The SDK includes two DDR3 memory controllers generated via Altera's IP wizard, one with ECC enabled and one with ECC disabled. Users can select which to instantiate by setting the correct parameter in the Project.tcl file. The DDR3 memory controller exposes an Altera Avalon bus interface operating at 200 MHz. The Shell abstracts away the Avalon bus and presents a Microsoft-defined interface called the User Memory Interface (UMI) to the Role. The purpose of using UMI is to insulate the Role from bus-specific protocols, e.g. burst size limits, while providing a convenient and high-performance abstraction that can be portable across future architectures with possibly different memory bus protocols or standards. The Shell internally implements a lightweight UMI-to-Avalon bridge that translates UMI transactions into Avalon.

2.4 Peripherals

There are several other small peripherals in the FPGA that provide resources for health monitoring and other basic services. These modules are mostly read-only status outputs and provide very limited configurability within the Shell.

2.4.1 ClockGen

This module provides clocking and reset signals for the rest of the FPGA using a Phase Locked Loop (PLL). The PLL takes the PCI Express reference clock from the host system, and interconnect reference clocks and generates several derivative clocks. Soft-Shell and Role developers can derive further clocks from these outputs if needed.

2.4.2 I^2C Slave

The shell includes an I2C slave interface which exposes a set of registers. The I2C slave sclk port supports speeds up to 100kHz. Table 1**Error! Reference source not found.** lists the exposed values and control

bits on the I2C Slave interface. These registers are **single byte** values. FPGA health and temperatures are exposed through the Shell register interface described in Section 5.

Address	Description
0x00	Zero Reg. Will always return 0x00
0x01	Current FPGA die temperature in Celsius
0x03	FPGA Health Status Bit 0: PCIe HIP Up Bit 1-7: Reserved
0x04	I2C Version Register.
0x05-0xFF	Reserved

Table 1: I2C Slave Registers

2.4.3 LEDs

Mt Granite provides 8 light-emitting diodes (LEDs) for debugging during initial board bring-up. During normal data-center operation, the LEDs are connected to a pattern generator and simply provide a visual indication that the FPGA is programmed and receiving a valid clock signal.

2.4.4 Single Event Upset (SEU) Mitigation Control

Altera's Stratix V FPGA used on Mt Granite includes configuration ROM scrubbing logic to help mitigate the effects from random particle strikes that might otherwise slowly corrupt the FPGA's programming.

The configuration of SEU scrubbing is done during compilation of the FPGA image. The shell uses an Altera provided block to tap into the internal scrubbing logic and provide a count of how many errors have occurred and whether they were correctable or not. These counts are exposed via registers to the host system so it can take corrective action if necessary.

Please see the Altera *SEU Mitigation for Stratix V Devices SV51011* documentation for more details.

2.4.5 Temperature Sensor

Stratix V includes a built-in temperature sensor. This module tracks the current chip temperature and exposes it via a status register as documented in the interface described in Section 5. This status register also reports the Min and Max temperatures since power-on and whether the FPGA is in the Critical or Shutdown temperature ranges.

Operating Zone	Min Temperature	Max Temperature
Normal		95 C
Critical	95 C	102 C
Shutdown	102 C	

Table 1: Temperature Sensor Ranges

The temperature module provides a `temperature_shutdown` signal to alert the ClockGen block that the chip is overheating. Currently this results in the FPGA being held in reset until the temperature is restored to normal operating range.

2.4.6 Unique Chip ID

Each Stratix V FPGA has a unique 64-bit ID hard-coded at manufacturing time. The shell reads this unique ID and exposes it several ways:

1. Through status registers 62 & 63 via the interface described in Section 5.
2. Via the JTAG Probe interface with id *CHID.*

3 Shell Interfaces

The following section describes the contract between the top-level Shell and any instantiated Role. In the short term, the current Shell interfaces are expected to sufficiently to cover most scenarios that may interest academics. In the long term, as the Academic program evolves, this contract will likely grow to accommodate new scenarios or new FPGA boards. As new Shell designs are released, Microsoft will attempt to maintain backwards compatibility with applications targeting older Shells, but we provide no guarantee that existing interfaces will always be preserved.

The remainder of this section discusses each of the Shell interfaces in greater detail.

3.1 PCIe Interface

The Host-to-FPGA communication layer is supported by a PCIe slot-based DMA architecture. Each ***slot*** consists of an input buffer and an output buffer, where "input" and "output" are relative to the FPGA, i.e. the input buffer moves data from software to the FPGA and the output buffer moves data from the FPGA to software. Slot buffers are allocated in contiguous physical system memory. An input slot is associated with a doorbell memory location that, when software writes to it, indicates that the input slot has data for the FPGA. Likewise, the output slot is marked as containing valid output data after the FPGA raises an interrupt.

Figure 5 gives a simplified illustration of this architecture.

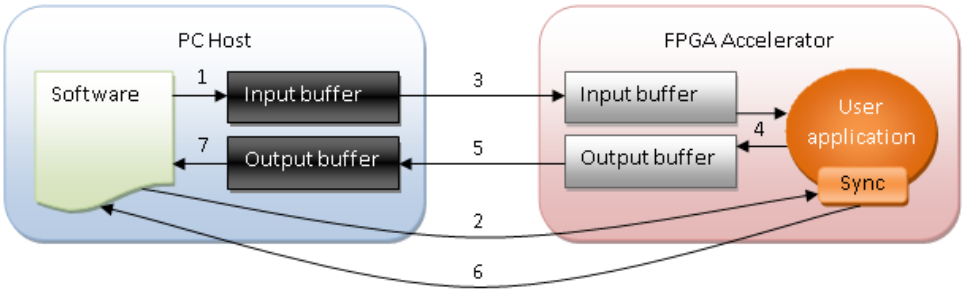

Figure 5: Simplified view of the PCIe Slot-Based DMA Architecture.

As shown in Figure 5, when a host wishes to send data to the FPGA (and process its response), the following steps are implemented:

1. The CPU places data to be processed into a kernel-pinned input buffer located in main memory.
2. The CPU rings the FPGA doorbell to inform it to retrieve and begin processing the data. The CPU thread is then either put to sleep waiting on a notification event from the FPGA, or continues execution.
3. The Shell transfers data from the CPU's memory and places it into the FPGA input buffer.
4. The Role processes data from the input buffer and writes the results to the output buffer.
5. The FPGA copies the contents of the output buffer into the CPU's memory in a kernel-pinned output buffer.
6. The Shell raises an interrupt, which signals that the output buffer is ready to be consumed.
7. The CPU thread wakes up and consumes the processed data from FPGA. The CPU discards the output buffer, which allows the FPGA to reuse it for the next transaction.

Figure 6 gives a more detailed view of the current slot-based implementation. Each buffer is 64KB in size, and there are 64 buffers available to the user. The FPGA itself has two internal input buffers and two output data buffers (4*64=256KB total) to enable overlapping of computation and communication. In addition, a pair of result buffers (one per output buffer) store metadata about a specific transaction, e.g. number of words returned. There are also additional control buffers and status registers for internal hand-shaking and synchronization between the FPGA and the CPU.

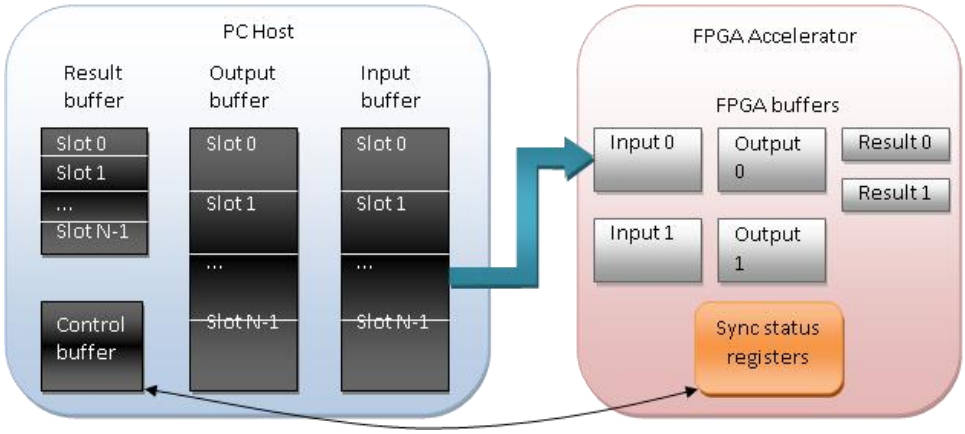

Figure 6: The PCIe Slot DMA Architecture exposes up to 64 buffers (aka slots), where each slot is 64KB in size. Control buffers and sync status registers are used for internal hand-shaking and synchronization.

3.1.1 FPGA-side Interface

In hardware, the CPU-transmitted message is delivered to the soft shell using the PCIe Host-to-FPGA interface shown in Table 3 and Figure 7. The interface is FIFO-based. Messages range from 32B to 64KB in size, with each message packet containing 16B of data. Messages arrive atomically: a message arriving on a given slot completes before starting another message.

When `pcie_full_out` is de-asserted by the Role, the Shell asserts `pcie_wren_in` to indicate a valid incoming packet. A packet comprises the 16B of data (`pcie_data_in`), the slot number used by software

to perform the DMA (pcie_slot_in), and whether this packet is the last data word of the message (pcie_last_in).

Note that in the current hardware implementation, all messages must be a multiple of 16B. We reserve an extra signal pcie_padbytes_in to support non-16B aligned messages in future releases, but this should currently be always set to zero.

Signal name	Dir	Width	Description
pcie_wren_in	IN	1	Asserted when the Shell is delivering a valid word of a PCIe message to the soft Shell. Only asserted when pcie_full_out is 0.
pcie_full_out	OUT	1	Asserted by the soft shell when it cannot accept a valid data word from the shell.
pcie_data_in	IN	128	A 16B word of the PCIe message being delivered from the shell.
pcie_slot_in	IN	16	The slot number used to deliver the message (set by software). Only bits 5:0 are currently valid, bits 15:6 are reserved and set to 0.
pcie_padbytes_in	IN	4	A reserved port to support non-16B-aligned accesses in the future. Should always be set to 0 and ignored by the user.
pcie_last_in	IN	1	Asserted on the last data word of the message.

Table 3: PCIe Host-to-FPGA interface.

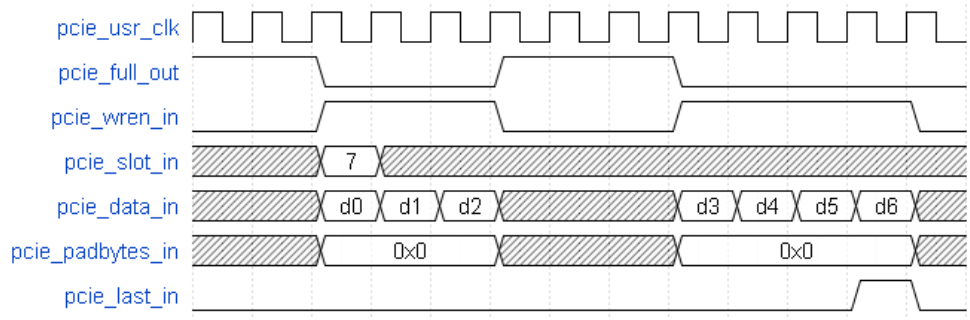

Figure 7: Example of receiving a message from the PCIe Slot-based DMA engine.

Table 4 and Figure 8 illustrate how the Role sends a message to the host. The Role de-asserts pcie_empty_out when it has a valid packet to send. The Shell asserts pcie_rden_in when it has accepted the valid packet. When sending a message, the user sets pcie_slot_out to the desired output slot buffer on the host and asserts pcie_last_out on the last packet of the message. Messages must be between 32B and 64KB in size, and a message sent to a given slot must complete before starting another message. The signal pcie_padbytes_out is reserved to support non-16B aligned accesses in the future and should always be set to 0 by the user.

Signal name	Dir	Width	Description
pcie_empty_out	OUT	1	De-asserted by the soft shell when it has a valid word of data to send.
pcie_rden_in	IN	1	Asserted by the soft shell when pcie_empty_out is de-asserted and when the shell is ready to accept a word of data.
pcie_data_out	OUT	128	A 16B word of the PCIe message being delivered to the shell.
pcie_slot_out	OUT	4	The slot number used to deliver the message (set by the hardware). Only bits 5:0 are currently valid, bits 15:6 are reserved and set to 0.
pcie_padbytes_out	OUT	16	A reserved port to support non-16B aligned accesses in the future. Should always be set to 0 by the user.
pcie_last_out	OUT	1	Asserted on the last data word of the message.

Table 4: PCIe FPGA-to-Host Interface.

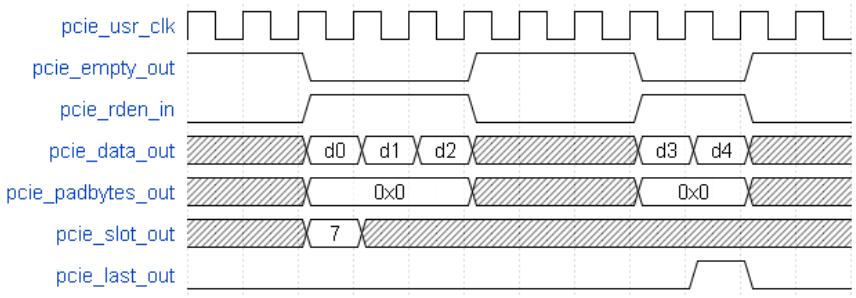

Figure 8: Sending a message from the PCIe Slot-based DMA engine.

3.2 PCIe Soft Register Interface

The shell exposes a "soft" register (softreg) interface that allows simple register-based communication between the software and the Role. This interface is typically used during run-time initialization or to query the FPGA for status or statistics. The two FPGACoreLib functions shown below allows software to write and read 64-bit values to/from a 32-bit address space.

```
FPGA_STATUS FPGA_ReadSoftRegister(FPGA_HANDLE fpgaHandle, DWORD address, DWORD64 *readValue);
FPGA_STATUS FPGA_WriteSoftRegister(FPGA_HANDLE fpgaHandle, DWORD address, DWORD64 writeValue);
```

Figure 9 illustrates the hardware transaction that occurs when `FPGA_WriteSoftRegister` is called. The Shell asserts `softreg_write_in` to denote a valid softreg write request to the Role. The request comprises a 64-bit payload (`softreg_wrdata_in`) and 32-bit address (`softreg_addr_in`). The Role logic can either process the request or choose to ignore it, but no indication of either is returned to software. There is no backpressure mechanism available to prevent a softreg write from occurring.

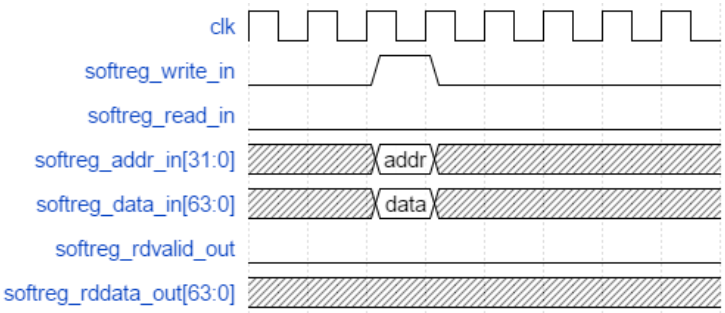

Figure 9: Waveform illustrating hardware transaction after software issues an FPGA soft register write.

Figure 10 similarly illustrates how the FPGA responds to a read request when `FPGA_ReadSoftRegister` is called. The Shell asserts `softreg_read_in` to denote a valid softreg read request and provides a 32-bit address (`softreg_addr_in`). The Role must reply with a 64-bit data response within **1000** clock cycles or a timed out PCIe response will be issued, resulting in undefined behavior. The Role sends a softreg read response by asserting `softreg_rdvalid_out` along with the 64-bit data payload (`softreg_rddata_out`).

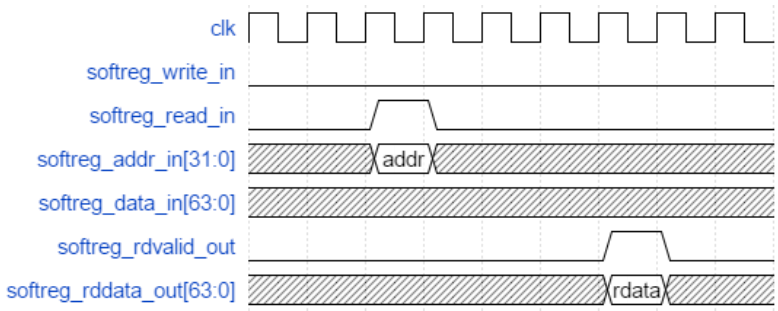

Figure 10: Waveform illustrating how hardware responds to a read request after software issues an FPGA soft register read.

3.3 User Memory Interface (UMI)

The User Memory Interface (UMI) protocol exposes a simple split-transaction DRAM interface to the Role. Figure 11 illustrates the major components of the FPGA DRAM subsystem for a single channel; the blocks are duplicated for the second channel. On the far right of the figure, 4GB of DDR3 ECC memory is attached to Altera's DDR3 memory controller. The UMI-to-Avalon adapter translates the Avalon interface into the Microsoft-defined User Memory Interface (UMI) protocol.

Using UMI insulates users from bus-specific protocols, e.g. burst size limits, while providing a convenient and high-performance abstraction that can be portable across future architectures with possibly different memory bus protocols or standards. For example, the UMI interface allows the user to issue memory requests up to the size of the DRAM capacity in a single transaction. The UMI-to-Avalon Adapter then automatically translates a logical request into smaller burst-optimized requests to the Avalon-based memory controller.

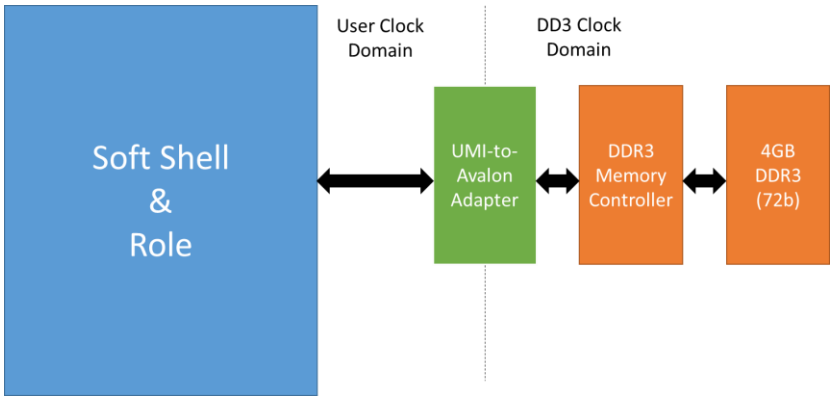

Figure 11: UMI Interface between Altera's Avalon-based DDR3 Memory Controller and the Soft Shell.

Table 5 lists every signal used in the UMI protocol. All UMI transactions are fully pipelined, i.e., multiple

requests can be issued in flight while multiple responses are pending.

Signal name	Dir	Width	Description
umi_raise_out	OUT	1	Asserted when requesting a new read or write transaction.
umi_write_out	OUT	1	Asserted when a write (not read) transaction is desired. Held steady when umi_raise_out is asserted.
umi_addr_out	OUT	64	Starting byte address of request. Value must be aligned to 64B. Held steady when umi_raise_out is asserted.
umi_size_out	OUT	64	Size of memory request in bytes. Must be a multiple of 64B. Held steady when umi_raise_out is asserted.
umi_grant_in	IN	1	Asserted when a request is accepted.
umi_rddata_in	IN	512	A 64B chunk of data read from DRAM.
umi_rdrdy_in	IN	1	Informs the user when a 64B chunk of data is available.
umi_rden_out	OUT	1	Asserted by the user when accepting a 64B word of data. It is only safe to strobe this when umi_rdrdy_in is asserted. For a single transaction, the user is expected to strobe this value N times, where N = umi_size_out / 64.
umi_wrdata_out	OUT	512	A 64B chunk of data to write to DRAM.
umi_wrrdy_in	IN	1	Asserted when there is sufficient space to write 64B of data.
umi_wren_out	OUT	1	Asserted by the user when writing 64B of data to DRAM. It is only legal to strobe this when umi_wrrdy_in is asserted. For a single transaction, the user is expected to strobe this value N times, where N = umi_size_out / 64.

Table 5: User Memory Interface (UMI)

Figure 12 gives an example of an UMI read transaction. The role asserts umi_raise_out until umi_grant_in is asserted. The umi_grant_in signal could be asserted on the same clock cycle as umi_raise_out or may take multiple cycles to be asserted. When umi_raise_out is asserted, the auxiliary signals umi_write_out, umi_addr_out, umi_size_out must be held valid. The umi_raise_out signal can remain asserted 1 cycle after umi_grant_in is asserted to immediately send another request, i.e. pipelining and back-to-back issuing of requests is permitted. The bottom six bits of umi_addr_out are ignored, forcing it to be a 64B aligned address. The umi_size_out signal must be a minimum of 64B and a multiple of 64B.

On a read response, the UMI controller asserts umi_rdrdy_in when the 64B data packet (umi_rddata_in) is ready to be sampled. The user asserts umi_rden_out when it is ready to accept the data, which can be on the same cycle as umi_rdrdy_in is asserted. On multi-cycle read transactions, the user must continuously assert umi_rden_out for each 64B read response packet. The user cannot indefinitely delay asserting umi_rden_out as doing so could cause the interface to deadlock.

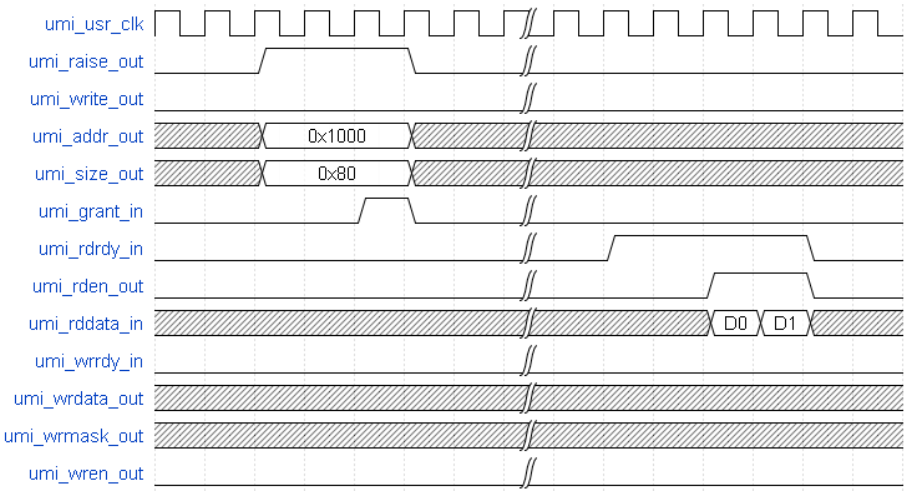

Figure 12: Example of an UMI read request to address 0x1000 for 128 bytes. All addresses and sizes must be aligned to 64.

Figure 13 shows how an UMI write request is handled. Writes are handled over a split interface, with the command interface providing the write address and size, and the data interface providing the 64B data packets. To send a write command, the Role asserts `umi_raise_out`, along with `umi_write_out`, `umi_addr_out`, and `umi_size_out`. To send the write data, when `umi_wrrdy_in` is valid, the Role should present a valid data word on `umi_wrdata_out` and assert `umi_wren_out`. The Role can write data prior to receiving `umi_grant_in` as long as `umi_wrrdy_in` is asserted. The currently-unimplemented `umi_wrmask_out` signal will eventually allow the user to perform partial writes into DRAM.

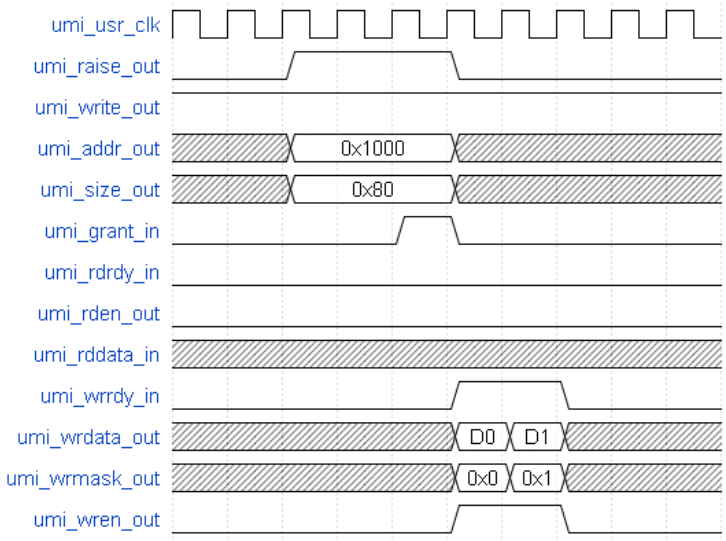

Figure 13: Example of an UMI write request to address 0x1000 for 128 bytes. All addresses and sizes must be aligned to 64.

3.4 SerialLite III

The SerialLite III (SL3) interface presents a simple FIFO-based send and receive interface for each of the four SL3 lanes. Tables 6 and 7 list the SL3 TX (transmit) and RX (receive) lane interfaces, respectively. The four lanes are named NORTH, SOUTH, EAST, and WEST. Each signal is declared as a four-entry array, representing the four bi-directional lanes. Each lane direction is associated with its own clock signal. Interfacing with SL3 will therefore require crossing clock boundaries between the Role clock and proper SL3 lane clock. Note that in Table 7, the RX interfaces do not have associated backpressure signals. It's up to the user to manually implement a flow control mechanism. Each lane direction is split into two virtual channels: the main RX/TX channel, and the RX/TX OOB channel. The OOB channels are low bandwidth; one could use this to implement a credit-based flow control mechanism.

Using SL3 requires first configuring the relevant SL3 shell registers. For neighbor discovery, each connected machine should have a unique user-defined 8-bit nodeID. In addition, all nodes are blocked from receiving traffic by default. To set the nodeID and to enable receiving traffic for each lane individually, software should set Control Register 2 to the proper value (see Section 5).

Signal name	Dir	Width	Description
`sl_tx_clk_in[3:0]`	IN	1	Transmission clocks. All signals below are synchronous to these clocks.
`sl_tx_out[3:0].valid`	OUT	1	If asserted, outgoing data is valid.
`sl_tx_out[3:0].data`	OUT	128	Outgoing data. Only valid if `sl_tx_out.valid` is true.
`sl_tx_out[3:0].last`	OUT	1	Asserted when this data word is the last in the outgoing message.
`sl_tx_stall_in[3:0]`	IN	1	De-asserted by the shell when able to accept a 128b word of data.
`sl_tx_oob_out[3:0].valid`	OUT	1	If asserted, outgoing OOB data is valid.
`sl_tx_oob_out[3:0].data`	OUT	15	Outgoing OOB data. Only valid if `sl_tx_oob_out.valid` is true.
`sl_tx_oob_rden_in[3:0]`	IN	1	Asserted when Shell accepts outgoing OOB data word.

Table 6: SerialLite III Transmit (outgoing) Interface

Signal name	Dir	Width	Description
`sl_rx_clk_in[3:0]`	IN	1	Reception clocks. All signals below are synchronous to these clocks.
`sl_rx_in[3:0].valid`	IN	1	If asserted, incoming data is valid.
`sl_rx_in[3:0].data`	IN	128	Incoming data. Only valid if `sl_rx_in.valid` is true.
`sl_rx_in[3:0].last`	IN	1	Asserted when this data word is the last in the incoming message.
`sl_rx_oob_in[3:0].valid`	OUT	1	If asserted, incoming OOB data is valid.
`sl_rx_oob_in[3:0].data`	OUT	15	Incoming OOB data. Only valid if `sl_rx_oob_in.valid` is true.

Table 7: SerialLite III Receive (incoming) Interface

4 Catapult Software API

This section provides a high-level summary of the user-facing software API implemented in the FPGA Communication Library. All functions return an FPGA_STATUS (data type is `unsigned int`) which indicates an error unless the value is FPGA_STATUS_SUCCESS.

4.1.1 Initialization and Cleanup

To access FPGA functions, first open a handle. This handle is then passed to all functions that communicate with the FPGA. Table 8 describes the API calls to open and close an FPGA handle.

API Name	Description

`FPGA_STATUS FPGA_CreateHandle(`
`  _Out_ FPGA_HANDLE *fpgaHandle,`
`  _In_ DWORD endpointNumber,`
`  _In_ DWORD flags,`
`  const char *pchVerDefnsFile,`
`  const char *pchVerManifestFile);`
	The FPGA_CreateHandle function instantiates and returns a handle to an FPGA device. endpointNumber should be set to 0. An additional 'flags' argument permits additional options that are reserved for future use and currently should be set to 0.
	
	The two last arguments point to files that are used to perform version checking of the FPGA bitstream at run-time.
	pchShellVerDefnsFile: Defines which version registers to verify. Leave as NULL to use default "FPGAVersionDefinitions.ini" (included in the library package).
	
	pchVerManifestFile: Defines compatible values for version registers. Leave as NULL to use default "FPGADefaultVersionManifest.ini" (included in the library package).

`FPGA_STATUS FPGA_CloseHandle(` `  _In_ FPGA_HANDLE fpgaHandle);`	Release and closes an FPGA handle.

Table 8: Software API for opening and for closing a handle to the FPGA

These functions are intended to be multi-process safe. Within a multithreaded process, FPGA_CreateHandle() should only be called once, and the handle can be shared among multiple threads. Similarly, FPGA_CloseHandle() should be called once, after all threads are done accessing the FPGA.

Before the FPGA can be used for normal operation, the following happens behind-the-scenes:

- BAR0 of the FPGA is mapped into the CPU's memory space (currently in FPGA driver).
- Pre-pinned physically contiguous memory is allocated (currently in root port filter driver) and the physical addresses are sent to the FPGA (currently within FPGA_CreateHandle()).
- An ISR is registered to the FPGA (currently in FPGA driver).

4.1.2 FPGA Capability Functions

The following functions in Table 9 can be used to query the capabilities of the FPGA. When FPGA_CreateHandle() is called, some reads over PCIe are performed to retrieve the values of the FPGA board capabilities. The functions below return the cached value from software. These values will always return the same value for Mt Granite boards and are used to maintain compatibility with future Catapult FPGA boards.

API Name	Description
FPGA_STATUS FPGA_GetNumberEndpoints(_In_ FPGA_HANDLE fpgaHandle, _Out_ DWORD *numEndpoints);	Retrieves the total number of available PCIe endpoints on an FPGA.
FPGA_STATUS FPGA_GetNumberBuffers(_In_ FPGA_HANDLE fpgaHandle, _Out_ DWORD *numBuffers);	Retrieves the total number of CPU slots (number of buffers sets).
FPGA_STATUS FPGA_GetBufferSize(_In_ FPGA_HANDLE fpgaHandle, _Out_ DWORD *bufferBytes);	Retrieves the total number of bytes available to use, per FPGA PCIe buffer.
FPGA_STATUS FPGA_GetNumberShellRegisters(_In_ FPGA_HANDLE fpgaHandle, _Out_ DWORD *numShellRegs);	Retrieves the total number of Shell registers.

Table 9: Software API for querying the capabilities of the FPGA

4.1.3 DMA Preparation Functions

The DMA engines require pre-pinned physically contiguous memory, which can only be allocated in kernel space (driver). The functions in Table 10 map the DMA buffers into user space so that software applications can use them. Note that PDWORD is a pointer to a DWORD.

The result buffer is a secondary output buffer meant for metadata, such as the amount of data that the FPGA wrote into the output buffer of the same slot number. If no other metadata is needed, an application does not need to obtain a pointer to any result buffer since this information is returned in the DMA operation API. The result buffer pointers are still available in case some applications want to add additional metadata, e.g. checksums.

API Name	Description
FPGA_STATUS FPGA_GetInputBufferPointer(_In_ FPGA_HANDLE fpgaHandle, _In_ DWORD slotNumber, _Out_ PDWORD *inputBufferPtr);	Given the handle and input slot number, retrieves a user-mapped pointer to the input buffer for the slot. The available buffer capacity is finite and should be queried using FPGA_GetBufferSize(). The total number of buffers available can be queried using FPGA_GetNumberBuffers().
FPGA_STATUS FPGA_GetOutputBufferPointer(_In_ FPGA_HANDLE fpgaHandle, _In_ DWORD slotNumber, _Out_ PDWORD *outputBufferPtr);	Same as the above, but for the output buffer.
FPGA_STATUS FPGA_GetResultBufferPointer(_In_ FPGA_HANDLE fpgaHandle, _In_ DWORD slotNumber, _Out_ PDWORD *resultBufferPtr);	Same as above, but for the result buffer. The size of each result buffer is hardcoded to 128 bytes. Currently the first DWORD of the result buffer stores the number of bytes of data that the FPGA placed into the output buffer of the same slot number.

Table 10: Software API for retrieving DMA buffers

4.1.4 DMA Operation Functions

Once pointers to the DMA buffers are retrieved, call the functions in Table 11 to send and receive data to and from the FPGA. The driver supports both polling and interrupts; polling results in higher performance but significantly increased CPU utilization. FPGA_SendInputBuffer() is non-blocking: the call returns immediately, so if you wish to send more data to the same slot, you must first wait for the FPGA to finish consuming the data by calling FPGA_GetInputBufferFull() described in the next section.

API Name	Description
FPGA_STATUS FPGA_SendInputBuffer(_In_ FPGA_HANDLE fpgaHandle, _In_ DWORD slotNumber, _In_ DWORD sizeBytes, _In_ BOOL useInterrupt);	This function initiates a send operation that transfers the contents of a specified input slot buffer to the FPGA. This function is non-blocking and returns immediately to the user. The user must specify whether an interrupt should be generated when a message is written to the corresponding output buffer by the FPGA.
FPGA_STATUS FPGA_WaitOutputBuffer(_In_ FPGA_HANDLE fpgaHandle, _In_ DWORD slotNumber, _Out_ DWORD *pBytesReceived, _In_ BOOL useInterrupt, _In_ double timeout = 1.0e-5);	This function blocks the calling thread until a message is received from the FPGA for a given output slot buffer. The user must specify whether an interrupt is expected or polling should be used. If an interrupt is expected, a previous interrupt-enabled call to FPGA_SendInputBuffer must have been made to the same corresponding input buffer number. The timeout parameter, when overridden from the default, will cause the calling thread to unblock after a specified duration (in seconds). To disable timeout, set to 0.
FPGA_STATUS FPGA_DiscardOutputBuffer(_In_ FPGA_HANDLE fpgaHandle, _In_ DWORD slotNumber);	This function allows a user to discard an output buffer after its contents have already been consumed and are no longer needed. The user should call this after waiting on an output buffer using FPGA_WaitOutputBuffer().

Table 11: Software API for DMA operation

4.1.5 DMA Status Functions

One can determine input and output slot buffer status using the functions in Table 12. These provide read-only access to the relevant parts of the CPU control buffer. Input buffer data should only be modified when the input buffer is empty (i.e. not full). Output buffer data should only be read when the output buffer is done.

API Name	Description
FPGA_STATUS FPGA_GetInputBufferFull(_In_ FPGA_HANDLE fpgaHandle, _In_ DWORD slotNumber, _Out_ BOOL *isFull);	This function indicates whether an input buffer is full and unavailable for the host to write.
FPGA_STATUS FPGA_GetOutputBufferDone(_In_ FPGA_HANDLE fpgaHandle, _In_ DWORD slotNumber, _Out_ BOOL *isDone);	This function indicates whether the FPGA wrote to an output buffer, and is available for the host to read.

Table 12: Software API for query the status of DMA

4.1.6 Shell Register Access

The software API for reading and writing to Shell registers is summarized in Table 13. Section 5 describes the functionality and accessibility (e.g. read-only vs. read and write) of each register number. Each access to a Shell register is atomic and therefore thread-safe. However, a series of accesses is not atomic.

API Name	Description

`FPGA_STATUS FPGA_ReadShellRegister(`
`  _In_ FPGA_HANDLE fpgaHandle,`
`  _In_ DWORD registerNumber,`
`  _Out_ DWORD *readValue);`
	This function allows users to read 32-bit Shell registers.

`FPGA_STATUS FPGA_WriteShellRegister(`
`  _In_ FPGA_HANDLE fpgaHandle,`
`  _In_ DWORD registerNumber,`
`  _In_ DWORD writeValue);`
	This function allows users to write 32-bit Shell registers.

Table 13: Software API for accessing Shell registers

Table 14 lists the software for API for reading and writing to soft registers in the user Role. Like the Shell registers, accesses are atomic and therefore thread-safe.

API Name	Description

`FPGA_STATUS FPGA_ReadSoftRegister(`
`  _In_ FPGA_HANDLE fpgaHandle,`
`  _In_ DWORD registerNumber,`
`  _Out_ DWORD64 *readValue);`
	This function allows users to read 64-bit Soft Shell registers.

`FPGA_STATUS FPGA_WriteSoftRegister(`
`  _In_ FPGA_HANDLE fpgaHandle,`
`  _In_ DWORD registerNumber,`
`  _In_ DWORD64 writeValue);`
	This function allows users to write 64-bit Soft Shell registers.

Table 14: Software API for accessing soft registers

5 PCIe Memory Mapped Registers

As discussed in Section 3.1, many software-visible PCIe memory-mapped registers are accessible through PCIe Base Address Register (BAR) offsets. The register space is divided into three categories: shell registers, internal PCIe registers for slot-based DMAs, and user-defined soft registers. Table 15 gives a list of shell registers. Please note, these register definitions are subject to change.

As discussed in Section 4, the shell and soft registers are accessible through the FPGA Communication Library.

Application address	Register Name	Access	Description
0	Control register	R/W	Control register for shell, initialized to all 0's. Can be used to force resets or to clear status registers Bit 3 = clear temperature records Bit 6 = enable PCIe role interface. If disabled, incoming PCIe data is bypassed sent through loopback. Bit 14 = clear SL3 ECC error count Bit 30 = force manual app reset Bit 31 = force manual core reset All other bit fields are currently reserved and should be set to 0.
	RESERVED		
5	Control register 2	R/W	Control register 2 for shell, initialized to all 0's. Used for SL3 configuration. Bits [7:0] = node ID Bit 16 = Enable RX NORTH link Bit 17 = Enable RX SOUTH link Bit 18 = Enable RX EAST link Bit 19 = Enable RX WEST link
	RESERVED		
11	Soft register mutex	R/W	Reserved register to preserve legacy role register interface.
12	Soft register address	W	Reserved register to preserve legacy role register interface.
13	Soft register data 0 [31:0]	R/W	Reserved register to preserve legacy role register interface.
14	Soft register data 0 [63:32]	R/W	Reserved register to preserve legacy role register interface.
15	Soft register initiate read	W	Reserved register to preserve legacy role register interface.
16	Soft register data 1 [31:0]	R/W	Reserved register to preserve legacy role register interface.
17	Soft register data 1 [63:32]	R/W	Reserved register to preserve legacy role register interface.
	RESERVED		
32	DDR status register 1	R	32-bit register indicating DDR channel 1 health. Bit 0 = local calibration succeeded Bit 1 = local calibration failed Bit 2 = local calibration in progress Bit 3 = ECC errors detected Bit 29:4 = unused (set to 0) Bit 30 = DDR soft reset status (1 = in reset) Bit 31 = unused (set to 0)
33	DDR error count register	R	32-bit register count of single-bit correctable and double-bit non-correctable errors for DDR channel 1. Bit 15:0 = single-bit error count Bit 23:16 = double-bit error count Bit 31:24 = uncorrectable error count (detected not corrected)
34	PCIe DMA Health	R	PCIe slot DMA health monitoring.

			Bit 31 = presence detect (old designs may not have this feature) Bits 30:8 = reserved for future use Bit 7 = hang detected in FIFO shim from role to PCIe Bit 6 = hang detected within DMA FPGA to CPU Bit 5 = hang detected in FIFO shim from PCIe to role Bit 4 = hang detected within DMA PC to FPGA Bit 3 = FIFO shim underflow in FIFO going to PCIe Bit 2 = FIFO shim overflow in FIFO going to PCIe Bit 1 = FIFO shim underflow in FIFO coming from PCIe Bit 0 = FIFO shim overflow in FIFO coming from PCIe
	RESERVED		
40	Shell read data 2	R/W	Reserved register to preserve legacy role register interface.
41	Shell read data 2	R/W	Reserved register to preserve legacy role register interface.
57	Board ID	R	Bit 31:0 = unique FPGA board identification B0 = Mt. Granite Board
58	Shell Release Versions	R	Bit 31:16 = major release version Bit 15:0 = minor release version
	RESERVED		
61	Soft shell version number	R	32-bit version number of the soft shell. User defined.
62	Chip ID [31:0]	R	Lower 32 bits of a unique serial chip ID burned into every Altera FPGA.
63	Chip ID [63:32]	R	Upper 32 bits of a unique serial chip ID burned into every Altera FPGA.
64	Shell identifier	R	32-bit identifier for a given shell release.
65	Role version number	R	32-bit version number of the role. User defined.
66	Cycle counter [31:0]	R	Lower 32 bits of running counter operating on the soft shell `clk`.
67	Cycle counter [63:32]	R	Upper 32 bits of running counter operating on the soft shell `clk`.
68	Shell status register	R	32-bit shell status register. Bit 0 = shell is initialized and ready Bit 1 = reserved and set to 1 Bit 2 = shell PLL is locked Bit 3 = DRAM PLL is locked All other bits reserved and cleared to 0.
69	PCIe link status register	R	32-bit PCIe status register. Bit 3:0 = PCIe active lanes (1, 2, 4, 8) Bit 7:4 = PCIe link speed (1, 2, 3) Bit 31:16 = reserved (set to 0)
70	Role status register	R	32-bit role status register. User defined.
71	Temperature status register	R	32-bit temperature information. Bit 0 = temperature warning (exceeded 95 degrees C) Bit 1 = temperature emergency shutdown (exceeded 102C) Bit 7:2 = unused (zero) Bit 15:8 = current temperature in Celsius Bit 23:16 = minimum temperature since power on Bit 31:24 = maximum temperature since power on
72	Capabilities register	R	32-bit register showing available capabilities for this shell. Bit 0 = reserved (set to 0) Bit 1 = DDR core enabled Bit 2 = reserved (set to 0) Bit 3 = reserved (set to 0) Bit 4 = PCIe enabled Bit 31:5 = reserved (set to 0)
73	DDR status register 0	R	32-bit register indicating DDR channel 0 health. Bit 0 = local calibration succeeded

			Bit 1 = local calibration failed Bit 2 = local calibration in progress Bit 3 = ECC errors detected Bit 29:4 = unused (set to 0) Bit 30 = DDR soft reset status (1 = in reset) Bit 31 = unused (set to 0)
74	DDR ECC counts 0	R	32-bit register count of single-bit correctable and double-bit non-correctable errors for DDR channel 0. Bit 15:0 = single-bit error count Bit 23:16 = double-bit error count Bit 31:24 = uncorrectable error count (detected not corrected)
75	PCIe Engine Identifier	R	Bit 15:0 = PCIe Engine Type 0x5107 = Slot-based DMA engine
76	PCIe version	R	31:16 = major revision 15:0 = minor revision
	RESERVED		
80	SL3 link status (NORTH)	R	32-bit register indicating SL3 NORTH link status Bit 0 = all lanes up and ready Bit 1 = TX errors detected Bit 2 = RX errors detected Bit 3 = wire swap enabled Bit 7:4 = lane up status by lane, [4] = lane 0 Bit 11:8 = error status by lane, [8] = lane 0 Bit 15:12 = errors present in shell, [12] = lane 0 Bit 23:16 = number of link locks Bit 31:24 = number of link errors
81	SL3 error count (NORTH)	R	32-bit register count of single-bit correctable and double-bit non-correctable errors. Bit 15:0 = Single bit error count Bit 31:16 = double-bit error count
	RESERVED		
84	SL3 link status (SOUTH)	R	32-bit register indicating SL3 SOUTH link status Bit 0 = all lanes up and ready Bit 1 = TX errors detected Bit 2 = RX errors detected Bit 3 = wire swap enabled Bit 7:4 = lane up status by lane, [4] = lane 0 Bit 11:8 = error status by lane, [8] = lane 0 Bit 15:12 = errors present in shell, [12] = lane 0 Bit 23:16 = number of link locks Bit 31:24 = number of link errors
85	SL3 error count (SOUTH)	R	32-bit register count of single-bit correctable and double-bit non-correctable errors. Bit 15:0 = Single bit error count Bit 31:16 = double-bit error count
	RESERVED		
88	SL3 link status (EAST)	R	32-bit register indicating SL3 EAST link status Bit 0 = all lanes up and ready Bit 1 = TX errors detected Bit 2 = RX errors detected Bit 3 = wire swap enabled Bit 7:4 = lane up status by lane, [4] = lane 0 Bit 11:8 = error status by lane, [8] = lane 0 Bit 15:12 = errors present in shell, [12] = lane 0 Bit 23:16 = number of link locks Bit 31:24 = number of link errors

89	SL3 error count (EAST)	R	32-bit register count of single-bit correctable and double-bit non-correctable errors. Bit 15:0 = Single bit error count Bit 31:16 = double-bit error count
	RESERVED		
92	SL3 link status (WEST)	R	32-bit register indicating SL3 WEST link status Bit 0 = all lanes up and ready Bit 1 = TX errors detected Bit 2 = RX errors detected Bit 3 = wire swap enabled Bit 7:4 = lane up status by lane, [4] = lane 0 Bit 11:8 = error status by lane, [8] = lane 0 Bit 15:12 = errors present in shell, [12] = lane 0 Bit 23:16 = number of link locks Bit 31:24 = number of link errors
93	SL3 error count (WEST)	R	32-bit register count of single-bit correctable and double-bit non-correctable errors. Bit 15:0 = Single bit error count Bit 31:16 = double-bit error count
	RESERVED		
96	SL3 overall link status	R	32-bit register containing SL3 error bits Bit 0 = NORTH lane up Bit 1 = SOUTH lane up Bit 2 = EAST lane up Bit 3 = WEST lane up Bit 7:4 = unused (set to 0) Bit 8 = NORTH hard error Bit 9 = SOUTH hard error Bit 10 = EAST hard error Bit 11 = WEST hard error Bit 31:12 = unused (set to 0)
97	SL3 neighbor IDs	R	32-bit register containing detected 8-bit neighbor node IDs Bit 7:0 = NORTH node ID Bit 15:8 = SOUTH node ID Bit 23:16 = EAST node ID Bit 31:24 = WEST node ID
99	Soft shell identifier	R	32-bit identifier for soft shell (reserved). Passthrough = 32'h8AA5
100	Soft shell status register	R	Custom-defined for a given soft shell (reserved).
101	Role ID	R	32-bit Role ID.
104	Configuration CRC error count	R	32-bit count of configuration memory errors. Bit 15:0 = number of CRC errors detected in configuration memory during background scrubbing.
105	I2C Version register	R	8-bit I2C major version.
	RESERVED		

Table 15: PCIe Shell Registers

6 Shell Versioning and Distribution

Shells have version numbers of the following format: *year.month.major.minor*.

For example: 2014.12.3.7 (2014, December, v3.7.)

Major revision numbers are bumped on breaking changes to either the soft shell / role, or the PCIe software stack.

Minor revision numbers are bumped on bug fixes, performance optimizations, and other minor tweaks.

Shells will be distributed in the form of a pre-compiled encrypted netlist that cannot be edited by the user.