Scalable-Configurable AXI Switch (SCAS)

User Guide

V1.0 December 2013

Nanyang Technological University, Singapore

The ARCH Research Group, Nanyang Technological University, Singapore
`http://arch.sce.ntu.edu.sg/`
contact: vipin2@ntu.edu.sg

Contents

1 Introduction **1**

2 Installing SCAS **2**

3 Hardware **5**

3.1 PCIe manager 6

3.1.1 PCIe Endpoint block 6

3.1.2 PCIe transaction layer 6

3.1.3 Global Register Set 8

3.1.4 System Monitor 10

3.1.5 User Clock Generator 10

3.1.6 Multiboot Controller 10

3.1.7 PCIe-DRAM DMA/PIO controller (PDMA) 10

3.1.8 PCIe Stream Generator (PSG) 11

3.1.9 PSG Stream Arbitrators (PSA) 11

3.2 DRAM manager 11

3.2.1 DDR Controller 11

3.2.2 Four port memory controller (FPMC) 11

3.2.3 DRAM Stream Generator (DSG) 12

3.2.4 DSG stream arbitrator (DSA) 12

3.3 Ethernet manager 12

3.4 User Interface 13

4 Integrating Accelerators **15**

5 Software **17**

5.1 Software use cases 19

5.1.1 FPGA PIO access 19

5.1.2 DRAM PIO access 20

5.1.3 Host to DRAM DMA 20

5.1.4 DRAM to host DMA 21

5.1.5 Host to USER PCIe stream interface DMA 22

5.1.6 USER PCIe stream interface to host DMA 23

5.1.7 USER DRAM stream interface to DRAM DMA 23

5.1.8 DRAM to USER DRAM stream interface DMA 23

5.1.9 Host to USER DRAM stream interface DMA with buffering in DRAM 24

6 Customisation **25**

List of Figures

2.1 Directory Structure 2

3.1 Switch Architecture. 5
3.2 PCIe Manager Architecture. 7
3.3 DRAM Manager Architecture. 12

List of Tables

2.1 SCAS source hierarchy 3

3.1 Supported communication paths 6
3.2 Global Register Set 8
3.3 User Interface Signal Description 13

4.1 SCAS specifications 15

5.1 APIs 17

Chapter 1

Introduction

Reconfigurable computing (RC) aims to fill the gap between hardware and software to achieve much higher performance than software, while maintaining a higher level of flexibility than hardware. In order to achieve this performance benefit while supporting wide varieties of applications, reconfigurable systems are usually formed with the combination of reconfigurable logic and general purpose processors (GPPs). Presently the most widely used reconfigurable logic in RC is the Field Programmable Gate Arrays (FPGAs) due to their high logic capacity, flexible routing architecture and wider tool support. The reconfigurable logic implements custom hardware accelerators, which along with the software executed on the GPP provide much higher system performance compared to complete software implementations.

Although RC has been widely adopted in custom computing systems, it has not found acceptance into every day computing and scientific research. Most of the custom RC applications are developed as standalone systems with very limited portability and reusability since there is no unified software and hardware communication interface for them. Developers have to design communication and reconfiguration infrastructure even before testing the functionality of the target application. This means there is lower productivity and longer design and implementation time.

Scalable-Configurable AXI Switch (SCAS) is an attempt to encourage the adoption of FPGA based accelerators on commercial computers. This platform enables developers to quickly integrate hardware accelerators to a reusable communication as well as reconfigurable infrastructure capable of very high performance throughput. It supports PCIe, DRAM and Ethernet interfaces along with a configurable number of AXI-stream based communication channels to the user logic. In addition to this, the user logic is also provided an address/data (PIO) interface for both PCIe and DRAM. It allows run-time configuration of the communication pathways along with support for run-time reconfiguration of the user logic. The hardware as well as software infrastructure is made completely open-source so that developers can adapt the platform to cater their specific requirements. The present version is fully portable across Xilinx Virtex-6 FPGA based ML605 and Virtex-7 FPGA based VC707. To enable this portability, some performance benefits of VC707 are sacrificed. A high-performance platform will be released in the future exploiting the full capabilities of VC707.

Chapter 2

Installing SCAS

The SCAS source files along with installation scripts can be downloaded from our public GIT repository `https://github.com/vipinkmenon/fpgadriver`. The source file contains both the hardware as well as software components for SCAS. The hardware components include all the source files for creating the FPGA configuration in verilog format, the Xilinx IP cores in synthesised (netlist) format, the constraints file for directing the FPGA place and route tools and some scripts for enabling command line execution of Xilinx development tools. The software components mainly include the source file for the FPGA driver, the user library and a number of example applications. The hardware development can be done on both Windows as well as Linux operating systems, but the SCAS FPGA driver is only supported on Ubuntu platform.

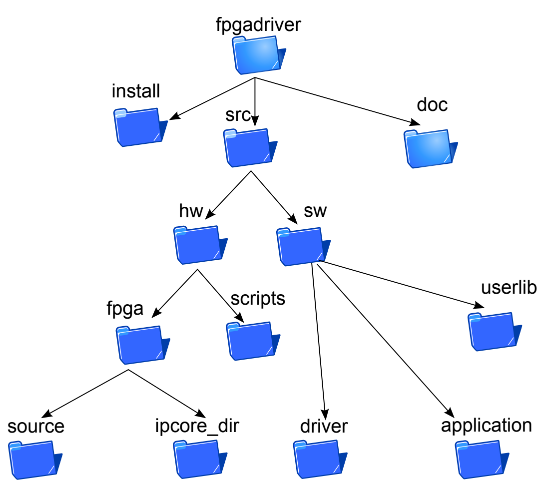

Figure 2.1: Directory Structure

The hierarchy of SCAS components are shown in Fig. 2.1 and described in Table 2.1. The user library contains all the APIs required by the user application software to communicate with the hardware. A single installation script is provided, which will install the Linux driver, user libraries and the ”hardware” components. To install the required file, switch the working directory into *fpgadriver/install* and run the following commands.

```
chmod 777 install.sh
sudo ./install
```

Running script builds the driver and the user library. It then installs the driver and copies the user library (libfpga.so) to the shared user library location (/usr/lib/). The installation also copies the hardware source files to a global location (/usr/include/fpga), so that the user application can access these components from any location similar to accessing the software shared library. Once the installation is finished, the user software application can include the *fpga.h* user library header file to use the driver APIs and can use *-lfpga* as the library path during compiling.

Table 2.1: SCAS source hierarchy

Name	Description
`fpgadriver`	The ROOT directory
`src`	hardware as well as software source directory
`install`	installation scripts
`doc`	SCAS documentation directory
`hw`	Contains source files and scripts corresponding to hardware implementation
`sw`	Contains the source code of driver, user library and example applications
`fpga`	Subdirectory corresponding to FPGA implementation source files
`scripts`	Scripts for command-line execution of Xilinx implementation tools. Also contains Xilinx GUI based implementation projects
`source`	Verilog source files corresponding to hardware module implementation
`ipcore_dir`	Pre-synthesised(netlists), verilog source files and regeneration scripts corresponding to Xilinx IP cores
`driver`	Source files corresponding to fpga driver
`application`	Example software applications
`userlib`	User software shared library source

Use the example bitstream provided (v6_top.bit or v7_top.bit depending upon the target FPGA board) in the fpga/bitstream directory to program the FPGA for verification. If you find difficulty in installing the Xilinx USB-JTAG cable for Ubuntu, follow the instructions at `http://www.george-smart.co.uk/wiki/Xilinx_JTAG_Linux#Newer_UDEV_.28Ubuntu_9.10.29`. Reboot the host system since this is the first installation and the host needs to detect the board. Now make sure that hardware is properly detected. If you can visually inspect the board after rebooting, there should be 3 LEDs glowing and 1 LED constantly blinking.

- LED0 : Blinking. Indicating Proper PCIe clock
- LED1 : Internal PLL lock for DRAM
- LED2 : PCIe link status
- LED3 : DRAM link status

It has been observed that ML605 board intermittently fails to detect DRAM. In this case, power cycle the FPGA board, reprogram the FPGA and reboot. To make sure that the host system has properly detected the FPGA board, execute the following command in the terminal

```
sudo lspci -vvv -d 10EE:*
```

lspci command lists the PCIe devices in the system and 0x10EE is the vendor ID for Xilinx. This command will give detailed information about the configuration space of the FPGA PCIe endpoint. The capability register values should indicate the device is Gen.2 capable and the link width is x4. The control register indicates what is the configuration set by the host for the FPGA device. For low-end host systems, the host may configure the device as only Gen.1 capable or in the worst-case as Gen.1 capable with only x1 link width. This can severely affect the system performance. If you have spare PCIe slots, try to plug-in the board to a different slot if this scenario occurs.

Make sure that the lspci command lists ”fpga” as the driver for the endpoint device. If the host fails to detect the driver, use the ”dmesg” command to detect possible errors.

```
dmesg | grep fpga
```

If the hardware and driver installations are successful, go the application directory. Run the *fpga_pio* example application by executing

```
./fpga_pio
```

This should return the FPGA hardware version and the system health parameters. The health parameter values are valid only on ML605 board since the VC707 board lacks the on-board sensors to measure these values.

Detailed description of integrating user applications and writing the application software are given in Chapters 4 and 5.

Chapter 3
Hardware

SCAS hardware provides programmable communication paths between different interfaces while maximising the throughput. From an abstract point of view, SCAS can be seen as a form of crossbar switch (although not entirely true) interfaced with standard PCIe, DRAM, and Ethernet interfaces and a scalable AXI4-Stream based user logic interface. The PCIe interface supports up to x4 PCIe Gen.2 standard, the DRAM interface supports up to 1GB DDR3-800 SODIMM modules and the Ethernet interface supports 1Gbit (Gigabit) raw Ethernet communication bandwidth.

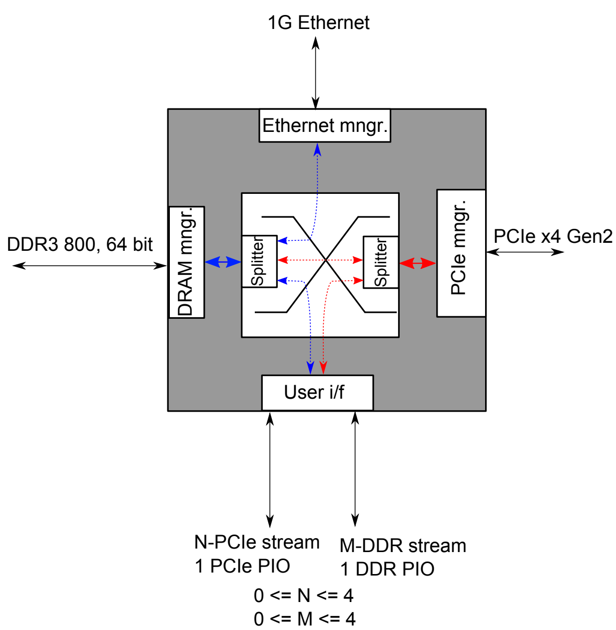

Figure 3.1: Switch Architecture.

The user interface supports 64-bit wide AXI4-Stream interfaces from both PCIe and DRAM (User PCIe Stream interface and User DRAM Stream interface). The number of AXI4-Stream channels can be configured between 0 and 4 for both PCIe and DRAM. The interface clock frequency to user logic for these stream interfaces are run-time reconfigurable and it defaults to 250MHz. An abstract view of the switch architecture is given in Fig. 3.1.

The valid communication pathways for SCAS are shown in table 3.1. The host system where the SCAS FPGA card is plugged in can directly communicate to the on-board DRAM and the user interface. Presently communication through Ethernet is supported only from/to the DRAM. Concurrent communication operations are possible and are managed within the FPGA logic and the driver software. Generally user application does not have to bother about communication pathways interacting and causing data corruption or bottlenecks. Some of the possible contention scenarios are described in Chapter 4. Detailed description of hardware modules are provided in the following sections. For developing applications or using SCAS, users do not have to understand these low level details. These are provided to encourage developers who would like to customise SCAS for their specific use cases.

Table 3.1: Supported communication paths

Interface	PCIe	DRAM	Ethernet	User
PCIe	✓	✓	✗	✓
DRAM	✓	✗	✓	✓
Ethernet	✗	✓	✓	✗
User	✓	✓	✗	✗

3.1 PCIe manager

The PCIe manager module manages all the data communication between the host system and the FPGA through PCIe interface. The major sub-modules within this block are shown in Fig. 3.2.

3.1.1 PCIe Endpoint block

This module uses the Xilinx PCIe Endpoint hard block configured in PCIe Gen.2 configuration with x4 link width. This is the highest configuration supported in the Virtex-6 FPGAs although Virtex-7 devices supports PCIe Gen.2 in x8 configuration. Theoretically PCIe x4 Gen.2 configuration can give a maximum throughput of 2GBytes/sec per direction in full-duplex mode. The present configuration settings enable high-portability of user applications irrespective of the target FPGA. The endpoint block implements the lower layers of PCIe protocol such as the data link layer and the physical layer. The physical layer uses Xilinx's GTP tranceivers. The maximum payload size for the PCIe core is set to 256bytes and the core uses 4 internal BRAM based buffers to store the PCIe packets. The backend of the Endpoint is integrated with the fabric using AXI4-Stream interface to generate and consume PCIe packets from the upper layers.

3.1.2 PCIe transaction layer

The Endpoint block is directly interfaced with the receive and transmit engines, which act as the transaction layer for the PCIe protocol. Transaction layer generates and consumes packets called transaction layer packets (TLPs), which are the unit of communication in PCIe. The interface to the transaction layer is 64bits wide and runs at 250MHz clock frequency provided by the Endpoint. The receive engine (Rx engine) decodes the received TLPs and route them to the appropriate sub modules. The received TLPs may correspond

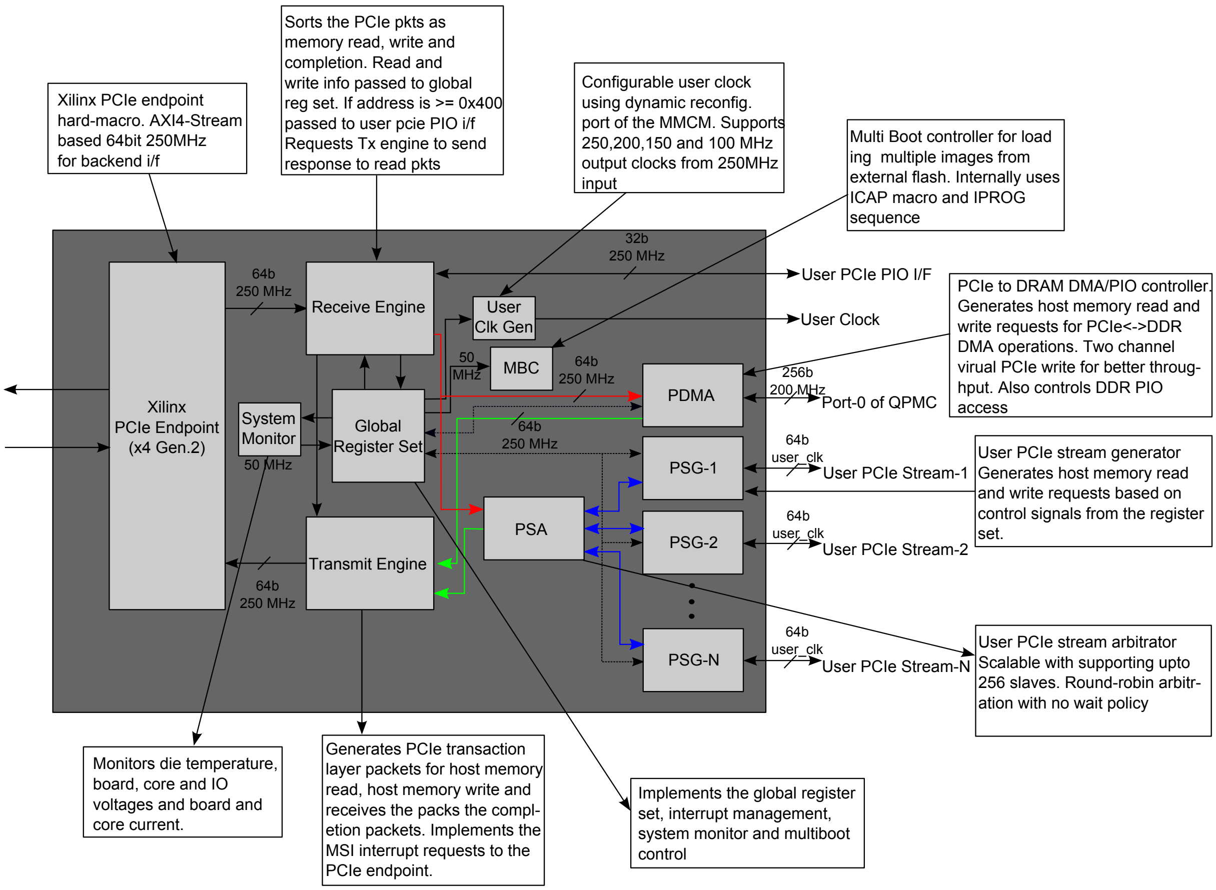

Figure 3.2: PCIe Manager Architecture.

to a memory read requests, memory write requests or a completion packets resulting from a memory read request issued by the FPGA. If the memory write request is for an address location less than 0x400, it is routed to the global register set else it is directed to the user PCIe PIO interface. Similarly for memory read requests, the request is forwarded to the global register or the user PCIe PIO interface depending upon the address and also triggers the transmit engine (Tx engine) to send the completion packet corresponding to this request with the data from the requested location.

From the completion packets, Rx engine extracts and packs the data and streams it out along with the *tag number* in the received packet. Tag number is a field in the PCIe packet inserted by a read operation initiator. The device which responds to a read request should keep this same tag number in the completion packet, which enables the initiator to determine the specific request which generated this packet. This is particularly useful when there are multiple outbound read requests.

The Tx engine generates TLPs corresponding to memory read, memory write and completion. Memory read TLPs are generated during DMA operations to fetch data from host memory to the FPGA. Memory write TLPs are generated while transmitting data from FPGA to the host memory and completion TLPs are generated in response to PIO read requests from the host. The Tx engine also manages PCIe interrupts to the host (known as message signaled interrupt (MSI)) based on the request from the interrupt manager in the global register set.

3.1.3 Global Register Set

Table 3.2 lists the global register space of SCAS. The PCIe endpoint device is configured to request for 4MBytes in the host system memory address space. The first 0x3FC is reserved for the global register space and any access from the host above 0x3FC is routed to the user PCIe PIO interface. If more memory space is required by the user application, the Xilinx PCIe endpoint IP has to be regenerated. The details of regenerating the IP cores are described in Chapter 6. Users are also free to add additional registers in the global set if required.

Table 3.2: Global Register Set

Address	Name	Description
00h	VER	Hardware Version number
04h	SCR	Scratchpad Register for read write testing
08h	CTRL	Control register for switch configuration
0Ch	RES	Reserved
10h	STA	Status register corresponding to control register settings
14h	RES	Reserved
18h	UCTR	User control register for reset and clock control
20h	PIOA	Address for DRAM PIO operations
24h	PIOD	Data corresponding to DRAM PIO operations
28h	PC_DDR_DMA_SYS	Host memory starting address for Host to DRAM DMA
2Ch	PC_DDR_DMA_FPGA	DRAM starting address for Host to DRAM DMA
30h	PC_DDR_DMA_LEN	Host to DRAM DMA length
34h	DDR_PC_DMA_SYS	Host memory starting address for DRAM to Host DMA
38h	DDR_PC_DMA_FPGA	DRAM starting address for DRAM to Host DMA
3Ch	DDR_PC_DMA_LEN	DRAM to Host DMA length
40h	ETH_SEND_DATA_SIZE	Ethernet send data size
44h	ETH_RCV_DATA_SIZE	Ethernet receive data size
48h	ETH_DDR_SRC_ADDR	DRAM starting address for Ethernet send data
4Ch	ETH_DDR_DST_ADDR	DRAM starting address for storing Ethernet receive data
50h	RECONFIG_ADDR	External flash starting address for FPGA reconfiguration
54h-5Ch	RES	Reserved
60h	PC_USER1_DMA_SYS	Host memory starting address for Host to USER PCIE Stream-1 DMA
64h	PC_USER1_DMA_LEN	Host to USER PCIE Stream-1 DMA length
68h	USER1_PC_DMA_SYS	Host memory starting address for USER PCIE Stream-1 to Host DMA
6Ch	USER1_PC_DMA_LEN	USER PCIE Stream-1 to Host DMA length
70h	USER1_DDR_STR_ADDR	DRAM starting address for USER DRAM Stream-1 to DRAM DMA
74h	USER1_DDR_STR_LEN	USER DRAM Stream-1 to DRAM DMA length
78h	DDR_USER1_STR_ADDR	DRAM starting address for DRAM to USER DRAM Stream-1 DMA

7Ch	`DDR_USER1_STR_LEN`	DRAM to USER DRAM Stream-1 DMA length
80h	`PC_USER2_DMA_SYS`	Host memory starting address for Host to USER PCIE Stream-2 DMA
84h	`PC_USER2_DMA_LEN`	Host to USER PCIE Stream-2 DMA length
88h	`USER2_PC_DMA_SYS`	Host memory starting address for USER PCIE Stream-2 to Host DMA
8Ch	`USER2_PC_DMA_LEN`	USER PCIE Stream-2 to Host DMA length
90h	`USER2_DDR_STR_ADDR`	DRAM starting address for USER DRAM Stream-2 to DRAM DMA
94h	`USER2_DDR_STR_LEN`	USER DRAM Stream-2 to DRAM DMA length
98h	`DDR_USER2_STR_ADDR`	DRAM starting address for DRAM to USER DRAM Stream-2 DMA
9Ch	`DDR_USER2_STR_LEN`	DRAM to USER DRAM Stream-2 DMA length
A0h	`PC_USER3_DMA_SYS`	Host memory starting address for Host to USER PCIE Stream-3 DMA
A4h	`PC_USER3_DMA_LEN`	Host to USER PCIE Stream-3 DMA length
A8h	`USER3_PC_DMA_SYS`	Host memory starting address for USER PCIE Stream-3 to Host DMA
ACh	`USER3_PC_DMA_LEN`	USER PCIE Stream-3 to Host DMA length
B0h	`USER3_DDR_STR_ADDR`	DRAM starting address for USER DRAM Stream-3 to DRAM DMA
B4h	`USER3_DDR_STR_LEN`	USER DRAM Stream-3 to DRAM DMA length
B8h	`DDR_USER3_STR_ADDR`	DRAM starting address for DRAM to USER DRAM Stream-3 DMA
BCh	`DDR_USER3_STR_LEN`	DRAM to USER DRAM Stream-3 DMA length
C0h	`PC_USER4_DMA_SYS`	Host memory starting address for Host to USER PCIE Stream-4 DMA
C4h	`PC_USER4_DMA_LEN`	Host to USER PCIE Stream-4 DMA length
C8h	`USER4_PC_DMA_SYS`	Host memory starting address for USER PCIE Stream-4 to Host DMA
CCh	`USER4_PC_DMA_LEN`	USER PCIE Stream-4 to Host DMA length
D0h	`USER4_DDR_STR_ADDR`	DRAM starting address for USER DRAM Stream-4 to DRAM DMA
D4h	`USER4_DDR_STR_LEN`	USER DRAM Stream-4 to DRAM DMA length
D8h	`DDR_USER4_STR_ADDR`	DRAM starting address for DRAM to USER DRAM Stream-4 DMA
DCh	`DDR_USER4_STR_LEN`	DRAM to USER DRAM Stream-4 DMA length
E0-130	`RES`	Reserved
134h	`ETH_RX_STATISTIC`	Ethernet receive statistics
138h	`ETH_TX_STATISTIC`	Ethernet transmit statistics
13Ch-1FC	`RES`	Reserved
200h	`SMT`	System monitor temperature
204h	`SMA`	System monitor Vccint
208h	`SMV`	System monitor VccAux
20Ch	`SMP`	System monitor Iccint
270h	`SBV`	Board 12V supply current

274h	`SAC`	Board 12V Voltage
278-3FC	`RES`	Reserved

This module also implements the interrupt manager. The interrupt manager makes sure that multiple back to back interrupts are not sent to the host in case of concurrent data transfer operations since this may sometimes crash the host or cause interrupt misses. The manager queues the interrupt requests and clears the interrupts which are acknowledged by the host. The host acknowledges interrupts by write clearing the status register (0x10) when it detects an interrupt.

3.1.4 System Monitor

System monitor is used to monitor the system health parameters such as temperature and power consumption. It uses the Xilinx's System monitor hard macro along with on-board sensors. These parameters can be accessed from the host system using an API (*fpga_read_sys_param()*). The readings obtained from the System monitor are converted to actual values by the driver software using Xilinx specific transfer functions. This monitoring functionality is available only on ML605 board since there are no on-board sensors on VC707 board.

3.1.5 User Clock Generator

This module generates the clock frequency used for all user stream interfaces. There may be situations where the user logic cannot work at the system clock frequency. This module takes the 250MHz clock from the PCIe core as the input and generates the output frequency depending upon an API request (*user_set_clk()*). Presently the supported output frequencies are 250,200,150 and 100MHz. Internally this module uses the dynamic reconfiguration port (DRP) of an mixed-mode clock manager (MMCM) to derive the required clock frequency. Using dedicated MMCM for user clock generation makes sure that other clock signals are not disrupted during clock reconfiguration.

3.1.6 Multiboot Controller

The multiboot controller (MBC) helps to reconfigure the FPGA by loading a new bitstream from the external storage memory (Platform Flash or BPI). Internally the MBC instantiates Xilinx's ICAP primitive and uses the *IPROG* command sequence to trigger a reconfiguration operation. The application software uses the *fpga_reboot()* API to trigger a reconfiguration operation by specifying the starting address of bitstream in the memory. The ML605 board is equipped with both Platform Flash as well as the BPI flash and VC707 board has only a BPI flash. On VC707, while storing the bitstream in the BPI flash, it should be noted that the starting address-bitstream size combination should not cross 32MBytes since the BPI is partitioned into four region using the FPGA version selection pins.

3.1.7 PCIe-DRAM DMA/PIO controller (PDMA)

This module is responsible controlling data movement between the PCIe controller and the external DRAM memory. This can access DRAM in both PIO (address/data) and DMA modes. This module is configured by the registers in the global register space. This module implements two virtual channels for host-DRAM write operations to achieve better performance. In some applications PCIe write (host to FPGA) performance is lower due to some restrictions of the PCIe protocol. As per PCIe standard, the maximum data that can be requested by an endpoint from the host in a single request is 4KB (or maximum read request size set by the host in the PCIe endpoint configuration space). For large data transfers, this makes the endpoint to make several requests. When an endpoint point makes multiple outstanding requests, the protocol provides the host the flexibility to return the completion packets in any order, irrespective of the requested order. This is called out-of-order completion. If the endpoint cannot manage this scenario, it is forced to make a new request only after receiving the data for the previous request. This can severely affect system performance. To overcome this issue, SCAS make use of the *tag* number in PCIe packet. Each time a memory request is made to the host, the request is tagged with a specific number. As per the protocol, the completion packet returned for each request should maintain the tag number unaltered. This enables reassembling data in correct order by PDMA. Experiments showed that implementing only two such virtual channels will provide sufficiently high throughput performance. Asymmetric FIFOs are used to convert 64bit wide PCIe interface data stream to 256bit wide DRAM interface data stream.

3.1.8 PCIe Stream Generator (PSG)

This module generates AXI-streams to the user PCIe-stream interface. It is responsible for making data requests to the host system for transferring data from host to user stream interfaces and vice versa. PSG instantiates asynchronous FIFOs between the user interface and PCIe data managers, which enables running the user PCIe stream interface at a different clock frequency than the PCIe core frequency. The number of PSGs instantiated in SCAS depends upon the number of user PCIe stream interfaces specified by the user. The FIFOs within PSGs are 8KB in size and a PSG never initiates a PCIe read or write operation unless there is sufficient buffer to store data. This avoids creating deadlocks in the PCIe core. For example if PSG requests for data without checking the space in the local FIFO and the user application stops receiving data from PSG, the PCIe core is stalled until the user application starts accepting data.

3.1.9 PSG Stream Arbitrators (PSA)

This module arbitrates among different PSGs. It employs round-robin-arbitration among the streams to access the PCIe core controller. It is highly scalable with current implementation supporting up to 256 PSGs.

3.2 DRAM manager

DRAM manger controls communication between the PCIe manager, user DRAM interface and the Ethernet manager with the external DRAM memory.

3.2.1 DDR Controller

This module instantiates the Xilinx's DDR3 soft memory controller. It is generated using Xilinx's memory interface generator (MIG) available with the IP Core generator. The controller is configured to control a 1GB DDRM SODIMM module running at 400MHz I/O clock frequency. Its backend is a custom interface defined by Xilinx which is 256bits wide and runs at 200MHz.

3.2.2 Four port memory controller (FPMC)

This is the most important component of DRAM manager. This module multiplexes DRAM access from different sources. It has four identical ports which are interfaced to PCIe-DRAM DMA/PIO controller (PDMA), User DRAM PIO interface, User DRAM stream arbitrator

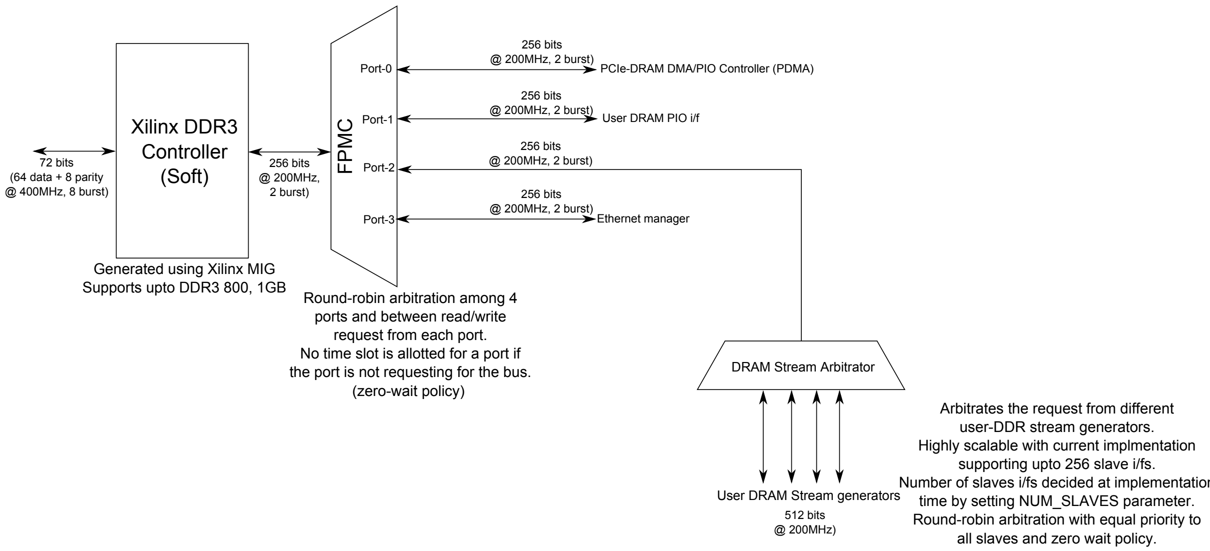

Figure 3.3: DRAM Manager Architecture.

(DSA) and the Ethernet manager. Requests from these sources are served in round-robin-arbitration scheme. Since the DDR controller has a single request port for access request, read-write request from each port is also served based on round robin. For better system performance, FPMC allows back-to-back DRAM read requests and internally stores them in a tracking buffer. When data is received from the DDR controller, it is routed to the appropriate port based on this tracking information.

3.2.3 DRAM Stream Generator (DSG)

DRAM stream generators control the DRAM-User logic AXI stream interfaces. They are quite similar to PSGs except the fact that their user side interface is 64bits wide and DRAM interface side is 256bits wide. Currently Xilinx does not support asymmetric (different read and write width) AXI Fifos. Hence DSGs use two FIFOs with intermediate logic to convert interface widths. The number of DSGs instantiated in the design depends upon the number of User DRAM stream interfaces specified by the user. The FIFOs also enable the user DRAM stream interface to work at a different frequency than the DRAM core frequency.

3.2.4 DSG stream arbitrator (DSA)

DSG arbitrates requests from DSGs in a round-robin fashion. DSG also implements a tracking buffer similar to the FPMC, which enables accepting back-to-back read requests from different DSGs.

3.3 Ethernet manager

The Ethernet manager control data movement between the DRAM manager and the Ethernet interface. For Virtex-6, the Ethernet control used is a hard Tri-mode Ethernet MAC, where as Virtex-7 uses a soft Ethernet core. The Virtex-7 Ethernet core requires special license from Xilinx for core as well as bitstream generation. Ethernet manager has sub modules which perform Ethernet specified packing and unpacking of data and FIFOs and associated control logic to manage interface width mismatch.

3.4 User Interface

The user interface includes PCIe stream interfaces, DRAM stream interfaces, PCIe PIO interface, DRAM PIO interface and the interrupt interface. The user interface signals are described in Table 3.3.

Table 3.3: User Interface Signal Description

Signal Name	Direction	Description
Clocks and Reset		
`i_pcie_clk`	input	250Mhz clock from PCIe controller. Should be used for PCIe PIO operations
`i_ddr_clk`	input	200Mhz clock from DRAM controller. Should be used for DRAM PIO operations
`i_user_clk`	input	User clock for PCIe and DRAM stream interfaces. Configurable at run-time and default value is 250Mhz.
`i_rst`	input	User logic reset signal. Polarity can be controller from the software
PCIe PIO interface		
`i_user_addr[19:0]`	input	PIO address for both read and write
`i_user_data[31:0]`	input	PIO 32-bit input write data
`i_user_wr_req`	input	PIO write request
`i_user_rd_req`	input	PIO read request
`o_user_data[31:0]`	output	PIO read data from user logic
`o_user_rd_ack`	output	PIO read data valid aligned with read data
DDR PIO interface		
`o_ddr_wr_data_valid`	output	Indicating user logic wants to write some data to DRAM. Xilinx DRAM controller requires 64-bytes of data in each write operation
`o_ddr_wr_data[255:0]`	output	Data to DRAM with [63:0] representing least significant data
`o_ddr_wr_data_be_n[255:0]`	output	Active low byte enables for write data
`o_ddr_rd`	output	Read request to DRAM. Each read request will return 64bytes data as two completions
`o_ddr_rd_addr[31:0]`	output	DRAM Read address.Should be 64bits aligned
`o_ddr_wr_addr[31:0]`	output	DRAM write address.Should be 64bits aligned
`i_ddr_rd_data[255:0]`	input	DRAM read data
`i_ddr_rd_data_valid`	input	DRAM read data valid signal
`i_ddr_wr_ack`	input	Acknowledge signal for DRAM write request
`i_ddr_rd_ack`	input	Acknowledge signal for DRAM read request. User logic can issue back to back read requests. The DRAM data is received along with data valid in the requested order after DRAM access latency.
USER PCIe stream interface		
`i_pcie_str1_data_valid`	input	Input stream data valid signal(AXI TVALID)
`o_pcie_str1_ack`	output	Acknowledge signal for data valid (AXI TREADY)
`i_pcie_str1_data[63:0]`	input	Input stream data
`o_pcie_str1_data_valid`	output	Output stream data valid
`i_pcie_str1_ack`	input	Output data valid acknowledge

`o_pcie_str1_data[63:0]`	output	Output stream data
USER DRAM stream interface		
`i_dram_str1_data_valid`	input	Input stream data valid signal(AXI TVALID)
`o_dram_str1_ack`	output	Acknowledge signal for data valid (AXI TREADY)
`i_dram_str1_data[63:0]`	input	Input stream data
`o_dram_str1_data_valid`	output	Output stream data valid
`i_dram_str1_ack`	input	Output data valid acknowledge
`o_dram_str1_data[63:0]`	output	Output stream data
USER interrupt interface		
`o_intr_req`	output	User interrupt request to host through PCIe. Should be kept asserted until acknowledge signal is asserted
`i_intr_ack`	input	Acknowledge signal for interrupt request signal. Asserted for a single clock cycle

All the streaming interfaces (PCIe and DRAM) interface works at *i_user_clk* frequency domain, whose default value is 250MHz. The PCIe PIO interface as well as the interrupt interface works at the PCIe core frequency (250MHz). The DRAM PIO interface works at DRAM core frequency, which is 200MHz.

Chapter 4
Integrating Accelerators

The user logic (or the accelerator) should have an interface which has a subset of signals described in Table 3.3. The user logic can have any number of sub modules, but the top most module name should be user_logic_top since it is the instantiation name used in SCAS. If following command line implementation, three files are required for generating the final bitstream.

1. fpga_spec.h : The SCAS specification function
2. V6_scas.tcl/V7_scas.tcl : Implementation TCL file
3. make_fpga.sh : Script to run the TCL file

The specifications in fpga_spec.h are given in Table 4.1.

Table 4.1: SCAS specifications

Spec.	Description	Default	Valid values
`FPGA_ADDR_MAX`	Upper limit for FPGA global register space	0x400	0x400 to 0x400000
`NUM_PCIE_STRM`	Number of User PCIe stream interfaces	4	0 to 4
`NUM_DDR_STRM`	Number of User DRAM stream interfaces	4	0 to 4
`ENET_ENABLE`	Whether Ethernet interface is required	1	0 or 1
`RECONFIG_ENABLE`	Whether multiboot option is required	1	0 or 1
`RCM_ENABLE`	Whether reconfigurable user clock option is required	1	0 or 1

When enabling or disabling the Ethernet interface, a few things has to be noted. For V6, the netlist for the Ethernet controller is installed in the global user library. If the interface is disabled in the fpga_spec.h file, the user has to set ENET_ENABLE to 0 in the tcl file also. This is to exclude the specific constraints used for Ethernet controller during implementation. Otherwise this will cause errors in the ISE translate phase. For V7, the Ethernet core has to be generated by the user with the name v7_emac_controller in SGMII standard. This is since generating this core requires a special license from Xilinx and hence cannot be publically distributed. The netlist for this core can be then stored in the global location or can be stored locally in the current working directory.

If the user logic is using sub modules, the names of these files have to be added in the tcl file using *xfile add* attribute along with the presently listed files. Additional constraints files can be also added in a similar manner. Now execute the script make_fpga with v6 or v7 as the argument depending upon the target board. The script fetches all the required source, netlist and constraints files from the global location and also uses the local user logic files. The final output will be top_v6.bit or top_v7.bit bitstream depending upon the target board. Since the intermediate files share the same name, do not run the script concurrently for different boards from the same working directory.

Chapter 5
Software

The SCAS software comes with a software driver and a user library. The driver implements all the low level device access files while the user library provides the APIs for high level data communication and system management. When SCAS is installed, the user library is compiled as a shared library and is stored at the Linux shared user library directory. This enables using these library functions from anywhere in the system by including the library header file (fpga.h) in the application software. The APIs provided by the user library are listed in Table 5.1.

Table 5.1: APIs

API name	Arguments	Description
High level APIs		
`load_bitstream`	(bitfile, dest_id)	Reprogram FPGA through JTAG with *bitfile*, specifying target device (FPGA_V6, FPGA_V7, FLASH_V6 or FLASH_V7)
`fpga_reboot`	(boot address)	Reprogram by loading a bitstream from the external flash stored at the boot address using ICAP
`fpga_read_sys_param`	NA	Read system monitor values such as temperature, voltage, current, power
`fpga_transfer_data`	(src, dest, buffer, len, addr, block)	The main data moving function. Transfers the specified number of bytes (len) from the source *(HOST, DRAM, USERPCIE1..USERPCIE4, USERDRAM1 ..USERDRAM4, ETHERNET)* to the destination. If the HOST is involved in the transfer, *buffer* is the user buffer to receive/transmit data. *block* specifies whether the operation has to be blocking/non-blocking

`fpga_wait_interrupt`	(channel)	Synchronization function for data transfers. Channel specifies the specific DMA channel for which synchronization is needed *(hostdram, dramhost, hostuser1, hostuser2, hostuser3, hostuser4, user1host, user2host, user3host, user4host, dramuser1, dramuser2, dramuser3, dramuser4, user1dram, user2dram, user3dram, user4dram, enet,user)*
`fpga_reg_wr`	(addr,data)	Write single 32-bit register in global register set or user logic
`fpga_reg_rd`	(addr)	Reading single 32-bit register in global register set or user logic
`fpga_ddr_pio_wr`	(addr, data)	Indirect write to a single 32-bit DRAM memory location
`fpga_ddr_pio_rd`	(addr)	Indirect read from a single 32-bit DRAM memory location
`user_soft_reset`	(polarity)	Issues a soft reset to the user logic with the specified polarity and deasserts
`user_set_clk`	(frequency)	Set the clock frequency to the user logic. (250, 200, 150 and 100 MHz)
`fpga_malloc`	(size)	Allocate specified size of block in FPGA DRAM
`fpga_free`	(start_address)	Free the allocated block in the FPGA DRAM
Low level APIs		
`fpga_init`	NA	Virtual device initialisation function
`fpga_close`	NA	Virtual device close function
`fpga_send_data`	(dest, buffer, len, addr)	Function to send data from the HOST to the FPGA. *dest* can be DRAM, USERPCIE1..USERPCIE4. If the destination is DRAM, address specifies the DRAM address, else used as the blocking/non-blocking argument
`fpga_recv_data`	(src, buffer, len, addr)	Function to receive data from the FPGA to the HOST. *src* can be DRAM, USERPCIE1..USERPCIE4. If the source is DRAM, address specifies the DRAM address, else unused
`fpga_send_ddr_user_data`	(dst, addr, buffer, len)	Function to send data from host to user DRAM stream interface via double buffering in the DRAM. *dst* can be USERDRAM1..USERDRAM4.

`ddr_user_send_data`	(dest, len, addr, block)	Function to send data from DRAM to user DRAM stream interface. *dst* can be USERDRAM1..USERDRAM4. *block* specifies whether the operation is blocking/non-blocking.
`user_ddr_send_data`	(src, len, addr, block)	Function to send data from user DRAM stream interface to DRAM. *src* can be USERDRAM1..USERDRAM4. *block* specifies whether the operation is blocking/non-blocking.
`enet_send_data`	(len, addr, block)	Function to send specified length of data from DRAM through Ethernet.
`enet_recv_data`	(len, addr, block)	Function to receive and store specified length of data from Ethernet to DRAM.

From the API description, it could be seen that several APIs have of blocking and non-blocking attribute. A blocking function is not returned until the operation is finished. A non-blocking function returns soon after the operation is initialised but in this case the user has to synchronise the operation later by calling a synchronisation function (fpga_wait_interrupt()). When the host is involved in data transfer, the blocking option is valid only when data is sent from host to the FPGA and not in the reverse direction. For non-blocking data transfers from host, the maximum data chunk size for a single operation is 4MB. Users must be careful while using blocking operations directly to the user logic as well as Ethernet interface. If the host initiates a large data transfer directly to user PCIe stream interface in blocking fashion, and if the user interface is quite slow in accepting it, there can be significant performance loss. Same applies to Ethernet data reception since it will be difficult to predict when the Ethernet data will be coming.

A potential dead lock when using block operation to user logic is when the user logic directly streams the processed data to the host. If the user initiates a blocking write with a large data (more that what can be buffered within user logic and stream controllers), all the buffers gets filled up and the user application can not initiate a read operation since the write operation never exits without transferring the complete data. For large data transfer operations, it is always encouraged to store the input data in DRAM and process it by reading from there and store the processed data back in the DRAM and later read back to the host.

5.1 Software use cases

This section describes the different use cases of communication between the host system and the fpga using different APIs.

5.1.1 FPGA PIO access

The following example shows how to access the FPGA global registers using the PCIe PIO operations and get the system health parameters such as die temperature, FPGA core and I/O supply voltages, FPGA and board power consumption etc using the *fpga_read_sys_param()* API. The *fpga_reg_rd()* and *fpga_reg_wr()* functions can be used to access the registers implemented in the user logic also. It should be noted that all the registers in the user logic should have address equal to or above 0x400 since address space

0x00 to 0x3FC are used by the FPGA global register space. The upper address limit for PIO access is 0x400000 limited by the present PCIe BAR0 configuration setting. The complete address map for the fpga global registers is available in the driver header file (fpga.h).

```
#include <stdio.h>
#include "fpga.h"

int main()
{
    sys_stat stat;                            //Structure returned by
    int rtn;                                  //fpga_read_sys_param function
    rtn = fpga_reg_rd(VER_REG);               //Read version register
    printf("Version : %0x\n",rtn);
    printf("Write scratch pad with 0x5\n"); //Write and read back scratchpad register
    rtn = fpga_reg_wr(SCR_REG,0x5);
    printf("Read scratch pad: ");
    rtn = fpga_reg_rd(SCR_REG);
    printf("%0x\n",rtn);
    printf("Reading system parameters\n");
    stat = fpga_read_sys_param();             //returns the sys_stat structure,
    printf("Temperature %f C\n",stat.temp); //which contains all the voltage,current,
    printf("VCCint %f V\n",stat.v_int);      //temperature and power information
    printf("Vaux %f V\n",stat.v_aux);
    printf("V12s %f V\n",stat.v_board);
    printf("Iint %f A\n",stat.i_int);
    printf("Iboard %f A\n",stat.i_board);
    printf("FPGA Power %f Watt\n",stat.p_int);
    printf("Board Power %f Watt\n",stat.p_board);
    return 0;
}
```

5.1.2 DRAM PIO access

The following example shows how to access the on-board DRAM memory location 0x1000 using PIO operations. Both ML605 and VC707 are shipped with 1-GByte DDR3 memory.

```
int main()
{
    int rtn;
    rtn = fpga_ddr_pio_wr(0x1000,0xA5A5A5A5);  //DRAM PIO write to 0x1000 with 0xA5A5A5A5
    rtn = fpga_ddr_pio_rd(0x1000);             //DRAM PIO read from 0x1000
    return 0;
}
```

5.1.3 Host to DRAM DMA

The *fpga_transfer_data()* API is used in the following example to transfer data from the host memory to the FPGA board DRAM. In the API, the source is designated as the DMA source and DRAM as the destination. The *fpga_malloc* API is used to get a free memory block in the DRAM. The incremental data to be sent is pre-stored in an array which will be copied to the kernel memory space by the driver. DMA between the host and the DRAM is always a blocking operation and hence the *block* parameter has no effect in the API call.

```
#define DATA_POINTS (1024*1024)          //Size of current DMA write in bytes
unsigned int senddata[DATA_POINTS/4];     //Buffer to hold the send data

int main()
{
    int rtn,i;
    unsigned int DRAM_ADDR;
    unsigned int arg = 0;
    unsigned int block = 0;

    DRAM_ADDR = fpga_malloc(DATA_POINTS); //Get the DRAM start address

    //Incremental Data for testing
    for(i = 0; i < DATA_POINTS/4; i++){
        senddata[i] = arg;
        arg++;
    }
    //Transfer the data
    rtn = fpga_transfer_data(HOST,DRAM,(unsigned char *)senddata,DATA_POINTS,DRAM_ADDR,block);
    rtn = fpga_free(DRAM_ADDR);
    return 0;
}
```

5.1.4 DRAM to host DMA

This example shows how data can be transferred from DRAM back to the host. It should be noted that the *block* parameter has to effect in this case. The function will return only after reading the complete data from DRAM.

```
#define DATA_POINTS (1024*1024)    //Size of current DMA read

unsigned int gDATA[DATA_POINTS];   //Buffer to hold the receive data

int main()
{
    int rtn,i;
    long usecs;
    unsigned int DRAM_ADDR = 0;
    unsigned int block = 0;
    //Transfer the data
    rtn = fpga_transfer_data(DRAM,HOST,(unsigned char *)gData,DATA_POINTS,DRAM_ADDR,block);
    return 0;
}
```

5.1.5 Host to USER PCIe stream interface DMA

This example shows how data can be directly send from the host to user PCIe stream interface with blocking option.

```
#define DATA_POINTS (64*1024*1024)  //Size of current DMA write
unsigned int senddata[DATA_POINTS/4];  //Buffer to hold the send data

int main()
{
    int rtn,i;
    unsigned int arg = 0;
    unsigned int block = 1;

    //Incremental Data for testing
    for(i = 0; i < DATA_POINTS/4; i++){
        senddata[i] = arg;
        arg++;
    }
    //Transfer data in blocking mode
    rtn = fpga_transfer_data(HOST, USERPCIE1,(unsigned char *) senddata, DATA_POINTS ,0, block);
    return 0;
}
```

This example shows how using the non-blocking capability enables sending data back-to-back from host to two different user PCIe stream interfaces. Note how the synchronisation function is used at the end to sync. the operations.

```
#define DATA_POINTS (4*1024*1024)  //Size of current DMA write
unsigned int senddata[DATA_POINTS/4];  //Buffer to hold the send data

int main()
{
    int rtn,i;
    unsigned int arg = 0;
    unsigned int block = 0;

    //Incremental Data for testing
    for(i = 0; i < DATA_POINTS/4; i++){
        senddata[i] = arg;
        arg++;
    }
    //Transfer data to User PCIe stream1 in non-blocking mode
    rtn = fpga_transfer_data(HOST, USERPCIE1,(unsigned char *) senddata, DATA_POINTS ,0, block);
    //Transfer data to User PCIe stream2 in non-blocking mode
    rtn = fpga_transfer_data(HOST, USERPCIE2,(unsigned char *) senddata, DATA_POINTS ,0, block);
    //Synchonise User PCIe stream1 transfer
    fpga_wait_interrupt(hostuser1);
```

```
    //Synchonise User PCIe stream2 transfer
    fpga_wait_interrupt(hostuser2);
    return 0;
}
```

5.1.6 USER PCIe stream interface to host DMA

```
#define DATA_POINTS (1024*1024)  //Size of current DMA read
unsigned int gDATA[DATA_POINTS];  //Buffer to hold the send data

int main()
{
    int rtn,i;
    unsigned int block = 0;
    rtn = fpga_transfer_data(USERPCIE1,HOST,(unsigned char *) gDATA,DATA_POINTS,0,block);
    return 0;
}
```

5.1.7 USER DRAM stream interface to DRAM DMA

```
#define DATA_SIZE 1024*1024  //Total number of bytes
int main()
{
    int rtn,i;
    unsigned int block = 1;
    unsigned int DRAM_ADDR = 0x0;
    fpga_transfer_data(USERDRAM1,DRAM,DATA_SIZE,DRAM_ADDR,block);
    return 0;
}

#define DATA_SIZE 1024*1024  //Total number of bytes
int main()
{
    int rtn,i;
    unsigned int block = 0;
    unsigned int DRAM_ADDR = 0x0;
    fpga_transfer_data(USERDRAM1,DRAM,DATA_SIZE,DRAM_ADDR,block);
    fpga_transfer_data(USERDRAM2,DRAM,DATA_SIZE,DRAM_ADDR,block);
    fpga_transfer_data(USERDRAM3,DRAM,DATA_SIZE,DRAM_ADDR,block);
    fpga_transfer_data(USERDRAM4,DRAM,DATA_SIZE,DRAM_ADDR,block);
    //Synchonise transfers
    fpga_wait_interrupt(user1ddr);
    fpga_wait_interrupt(user2ddr);
    fpga_wait_interrupt(user3ddr);
    fpga_wait_interrupt(user4ddr);
    return 0;
}
```

5.1.8 DRAM to USER DRAM stream interface DMA

```
#define DATA_SIZE 1024*1024  //Total number of bytes
```

```
int main()
{
    int rtn,i;
    unsigned int block = 1;
    unsigned int DRAM_ADDR = 0x0;
    fpga_transfer_data(DRAM,USERDRAM1,DATA_SIZE,DRAM_ADDR,block);
    return 0;
}
```

```
#define DATA_SIZE 1024*1024  //Total number of bytes
int main()
{
    int rtn,i;
    unsigned int block = 0;
    unsigned int DRAM_ADDR = 0x0;
    fpga_transfer_data(DRAM,USERDRAM1,DATA_SIZE,DRAM_ADDR,block);
    fpga_transfer_data(DRAM,USERDRAM2,DATA_SIZE,DRAM_ADDR,block);
    fpga_transfer_data(DRAM,USERDRAM3,DATA_SIZE,DRAM_ADDR,block);
    fpga_transfer_data(DRAM,USERDRAM4,DATA_SIZE,DRAM_ADDR,block);
    //Synchonise transfers
    fpga_wait_interrupt(ddruser1);
    fpga_wait_interrupt(ddruser2);
    fpga_wait_interrupt(ddruser3);
    fpga_wait_interrupt(ddruser4);
    return 0;
}
```

5.1.9 Host to USER DRAM stream interface DMA with buffering in DRAM

This operation is supported only in blocking mode.

```
#define DATA_SIZE 1024*1024  //Total number of bytes
int main()
{
    int rtn,i;
    unsigned int block = 0;
    unsigned int DRAM_ADDR = 0x0;
    fpga_transfer_data(HOST,USERDRAM1,DATA_SIZE,DRAM_ADDR,block);
    return 0;
}
```

Chapter 6

Customisation

Users can customise SCAS for their specific application. Additional registers can be implemented in the global register space by following the coding style and comments in the reg_file.v verilog source file inside source/pcie_if directory. Another requirement will be to have more streaming interfaces to connect additional streaming peripherals. This can be easily done by instantiating additional PSG and DSG modules (defined in pcie_stream_generator.v and dram_stream_generator.v). These modules basically require information such as source and destination addresses, transfer length. This can be done by implementing additional registers in global register space or in the user logic. These modules use simple start/done control signals for operation. They should be then integrated to the PSA and DSA for accessing the PCIe and DRAM cores.