BlazePPS (Blaze Packet Processing System)
CSEE W4840 Project Design

Valeh Valiollahpour Amiri (vv2252)
Christopher Campbell (cc3769)
Yuanpei Zhang (yz2727)
Sheng Qian (sq2168)

March 26, 2015

I) Hardware Design

Overall Design

The high-level block diagram for the overall design is presented below in two parts - the first part (fig. 1) shows the FPGA design and the second part (fig. 2) shows the HPS design. These designs will be referenced throughout this document.

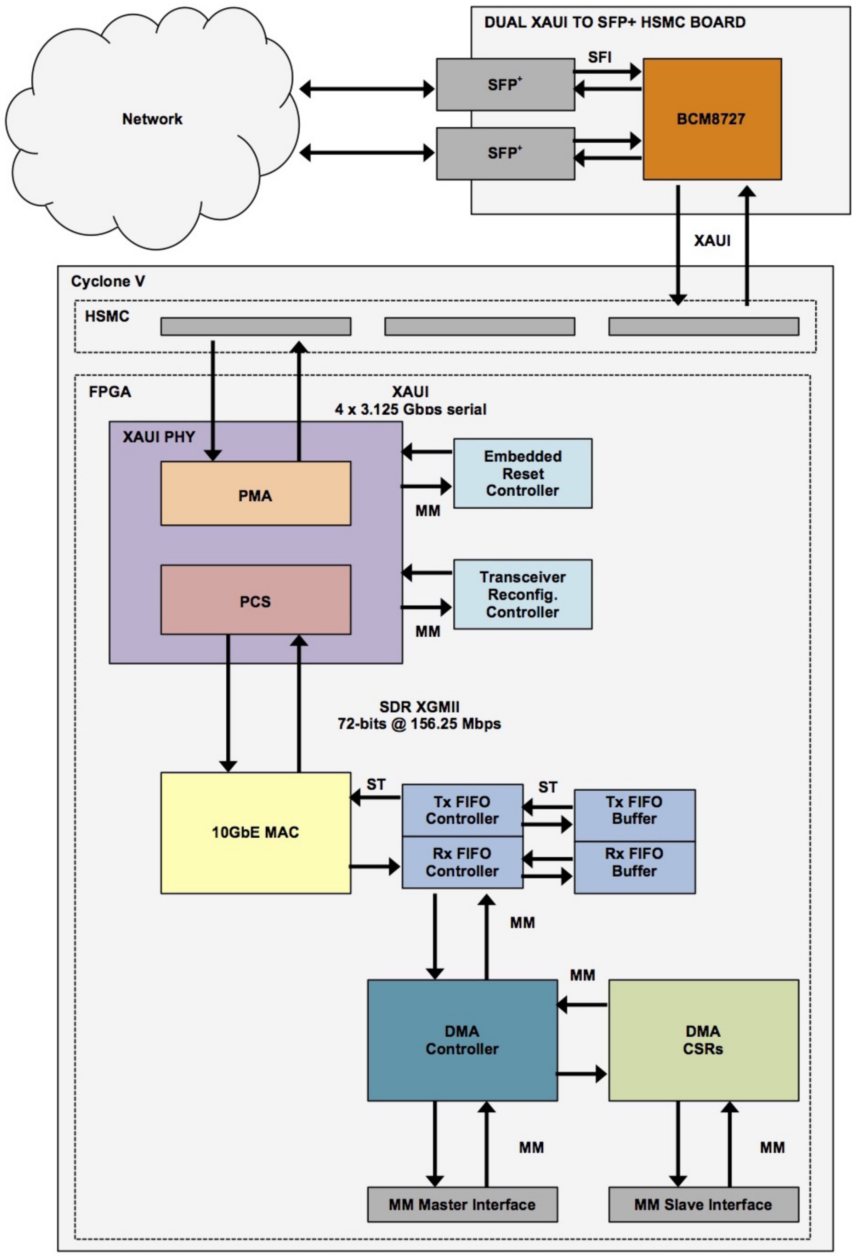

fig. 1 - High-level block diagram (FPGA)

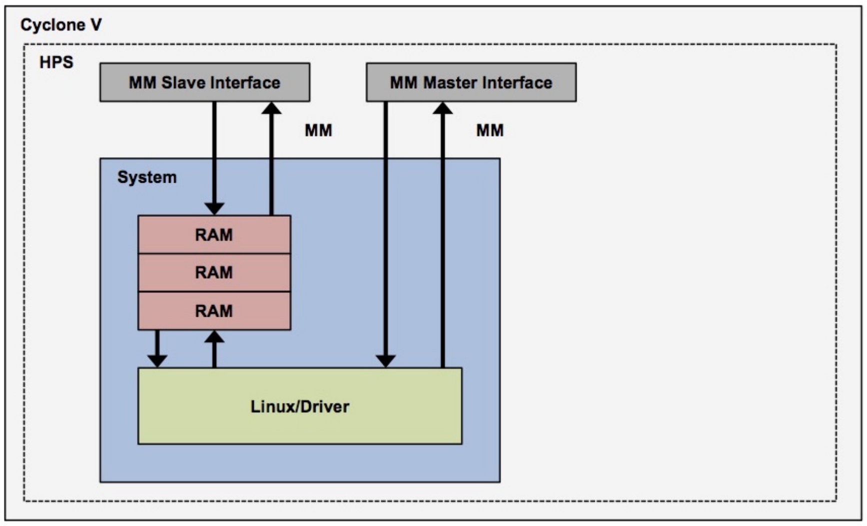

fig. 2 - High-level block diagram (HPS)

Dual XAUI TO SFP+ HSMC Board

The DUAL XAUI TO SFP+ HSMC board provides a SFP+ to XAUI platform that allows a 10GbE system to be implemented on the Cyclone 5 (fig. 1 and 3). It interfaces with the Cyclone V via the High Speed Mezzanine Card (HSMC) interface. It has two SFP+ transceivers that support 10GbE. The SFP+ transceivers convert incoming 10Gbps signals from the network to SFI serial signals. Only one SFP+ transceiver will need to be utilized for this project. The SFP+ transceiver sends the SFI signal to the BCM8727, which is a 10GbE SFI-to-XAUI transceiver. The BCM8727 converts the incoming SFI signal to a XAUI signal. XAUI stands for 10 Gigabit Attachment Unit Interface, it is a standard for extending the 10 Gigabit Media Independent Interface between the MAC layer and PHY layer of 10GbE. The XAUI interface is attached to the HSMC side of the board. The XAUI receiver utilizes 4 serial lanes for transmitting the signals from the BCM8727 to the device through the HSMC connector. Each serial lane carries 64 bits at 3.125 Gbps. For sending data from the device, to the BCM8727 transceiver, to the SFP+ transceiver, and onto the network, all of the aforementioned signal conversions are simply carried out in reverse.

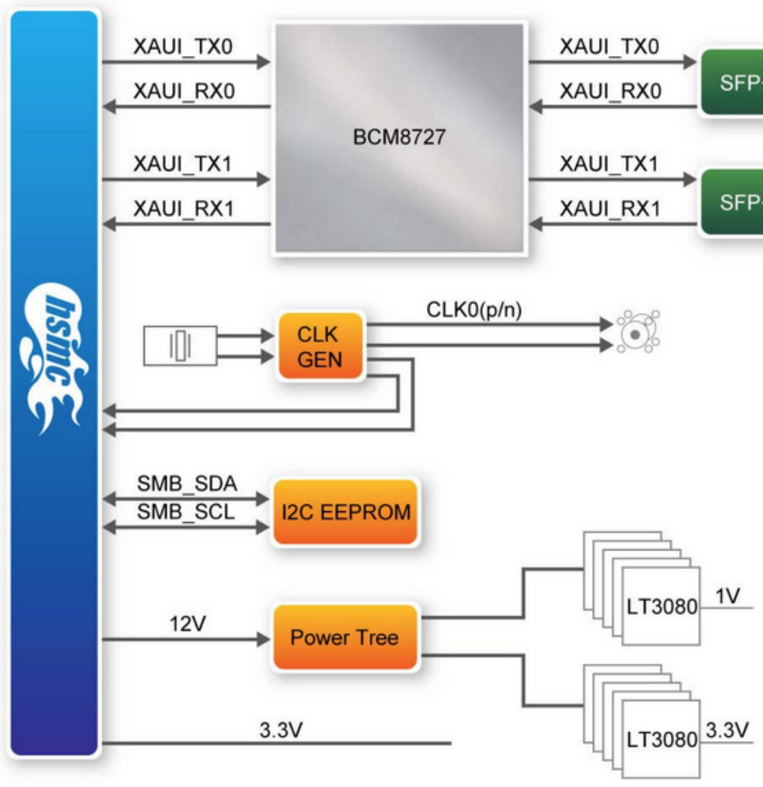

fig. 3 - DUAL XAUI TO SFP HSMC board

It has two full duplex 10G SFP+ channels

GHRD

The Altera Golden Hardware Reference Design (GHRD), which is part of the Golden System Reference Design (GSRD) (see Software Design section) will serve as the base hardware design. The GHRD will be utilized because it provides basic components that our design will take advantage of (fig. 4). More specifically, the GHRD includes:

- ARM Cortex™-A9 MPCore HPS
- Two user push-button inputs
- Four user DIP switch inputs
- Four user I/O for LED outputs
- 64KB of on-chip memory
- JTAG to Avalon master bridges
- Interrupt capturer for use with System Console
- System ID

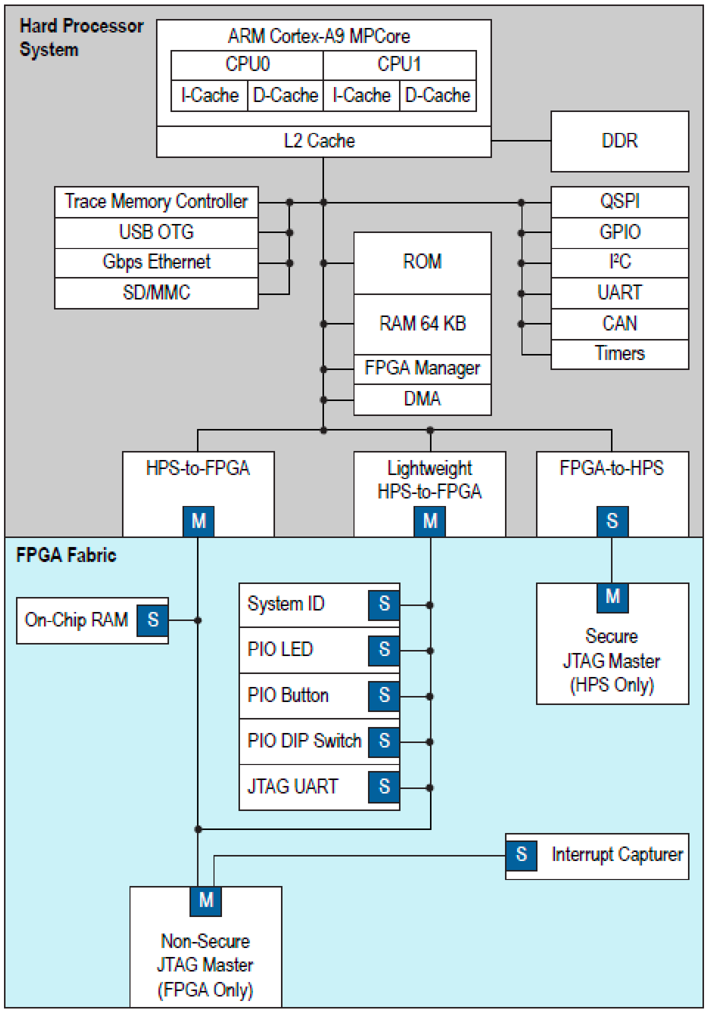

fig. 4 - GHRD

XAUI PHY

The Altera XAUI PHY IP Core will be utilized for the PHY layer (fig. 1 and 5). It implements the IEEE 802.3 Clause 48 specification to extend the operational distance of the XGMII interface. More specifically, it extends the physical separation possible between the 10GbE MAC and the Ethernet standard PHY component to one meter. On the side interfacing with the HSMC interface, the XAUI PHY takes the XAUI signals as input and also produces them as output. It implements either a hard or soft Physical Medium Attachment (PMA), which is chiefly responsible for synchronization and de-serialization/serialization. The PMA outputs to the Physical Coding Sublayer (PCS) and also receives input from the PCS going in the opposite direction. The PCS is chiefly responsible for performing scrambling/descrambling that aids in clock recovery. The PCS output utilizes the SDR XGMII (10 Gigabit Media Independent Interface). This interface is 72 bits wide (it carries 8 bytes plus 1 bit of control information for each byte) and the data comes out at the recovery clock of 156.25 Mbps. Its output is sent to the 10 GbE MAC (fig. 6). The PCS also receives 72 bits at 156.25 Mbps from the 10 GbE MAC going in the opposite direction (fig. 6). An embedded reset controller block and transceiver reconfiguration block have to be implemented along with the Altera PHY IP Core (fig. 1). These blocks are also provided by Altera.

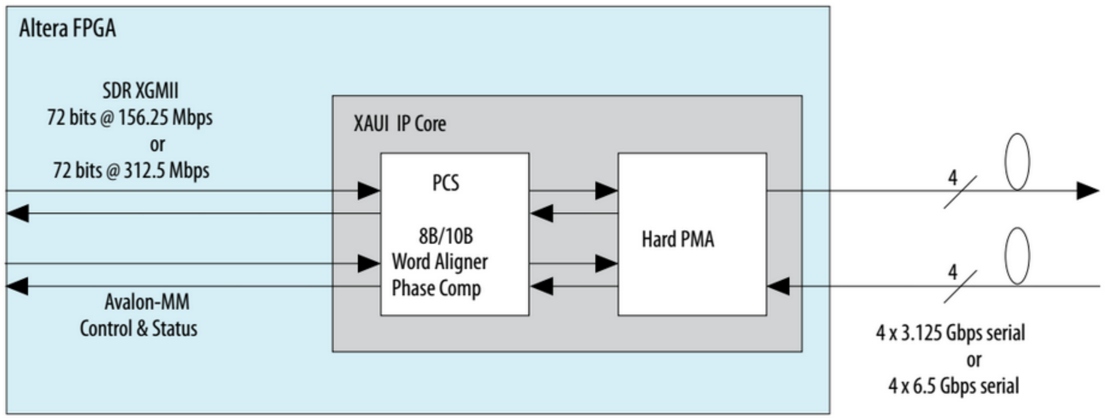

fig. 5 - XAUI PHY IP Core

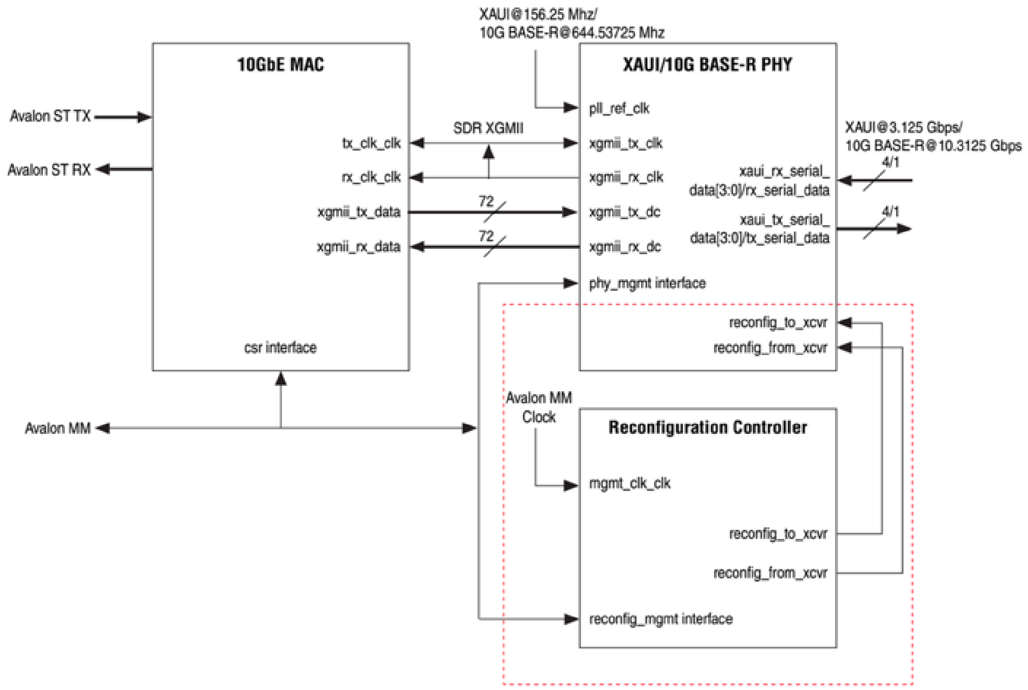

fig. 6 - XAUI PHY interfacing with the 10GbE MAC

10GbE MAC

The Altera 10GbE MAC IP core will be used to implement the 10 GbE MAC layer (fig. 1 and 7). The MAC IP core provides a direct interface to 64-bit SDR XGMII running at 156.25 MHz, through which it can receive data from and transmit data to PHY layer. On the receive path, the received data, which is in 802.3 Ethernet frame structure, is decoded within the MAC IP core and the payload is sent out to MAC layer network. On the transmission path, the payload is encoded into the Ethernet frame structure before sending to the PHY layer.

The 10GbE MAC IP core consists of three blocks: MAC receiver (MAC Rx), MAC transmitter (MAC Tx), and Avalon-MM bridge. The MAC Rx and MAC Tx handle data flow between the XAUI PHY and MAC layer networks, which are the Tx FIFO and the Rx FIFO. The Avalon-MM bridge provides a single interface to all Avalon-MM interfaces within the MAC, which allows a host to access 32-bit configuration and status register, and statistics counters.

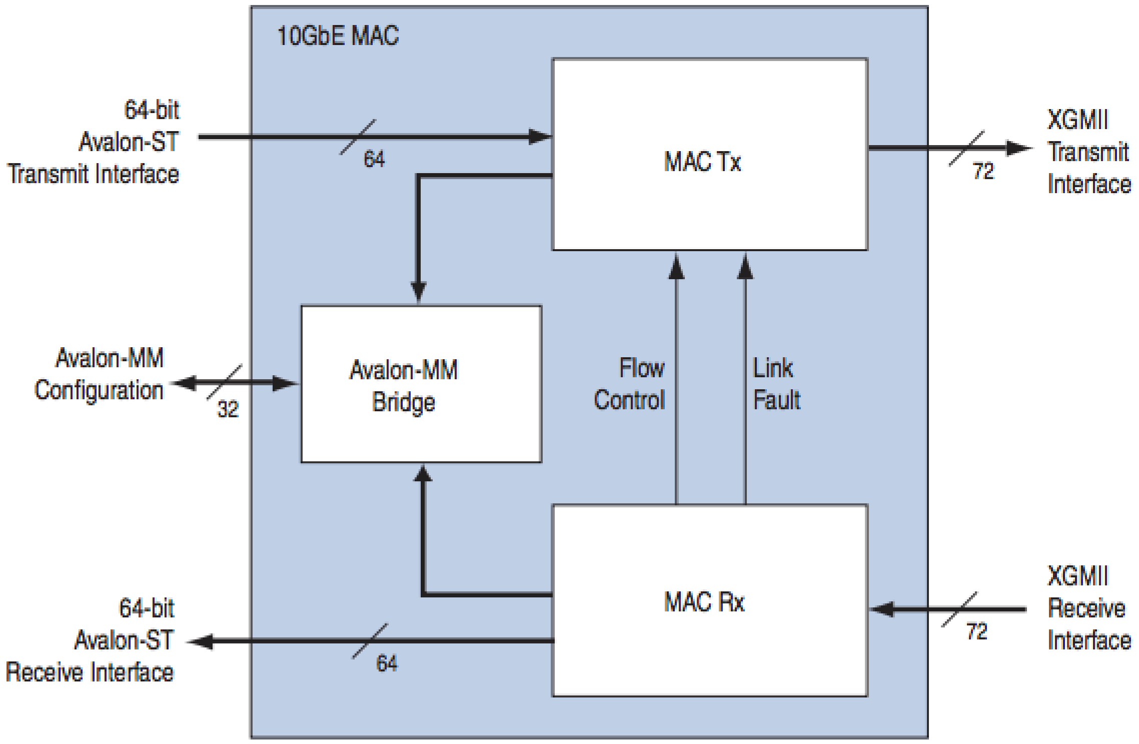

fig. 7 - 10Gb IP Core Block Diagram

On the MAC layer network side, the 10GbE MAC connects FIFO controllers through Avalon-ST, which is a synchronous point-to-point, unidirectional interface that connects a source of data stream to a sink of data stream. The packet data transfer feature is enabled for the purpose of transferring packets, and three additional signals are needed: *startofpacket*, *endofpacket*, and *empty*. During data transfer, *startofpacket* marks the active cycle containing the start of a packet, and it is only interpreted when *valid* is asserted. Similarly, *endofpacket* marks the active cycle containing the end of a packet. The *empty* signal indicates the number of symbols that are empty in the *endofpacket* cycle.

Tx/Rx Subsystem

The Tx/RX subsystem will be implemented from scratch. The Tx/RX subsystem consists of Tx/Rx controllers that regulate the Ethernet frames between the MAC controller and the Tx and Rx FIFOs as well as between the Tx and Rx FIFOs and the DMA subsystem. The Tx/Rx controllers use the 64-bit Avalon-ST interface to communicate with the MAC and Tx/RX buffers. The ST interface is used because data transmission needs to be high bandwidth and low latency in this situation. Additionally, because the data needs to be transferred as packets, the packet data feature of the ST interface is utilized (i.e. signals for start of packet, end of packet, and empty are implemented). The Rx controller is a sink for the 10GbE MAC and a source for the Rx buffer. The Tx controller is a source for the 10GbE MAC and a sink for the Tx buffer. The 64-bit Avalon-MM interface is used between the Tx/Rx controllers and the DMA controller. It is used to implement read and write interfaces for master and slave components. Here, the DMA controller is the Avalon-MM master and the Tx/Rx controllers are slaves.

DMA Subsystem

The Direct Memory Access (DMA) subsystem will be implemented from scratch. At a high level the DMA controller will shuttle data between the FIFOs and system memory (fig. 1). It will implement an Avalon MM master interface to put data in the Tx FIFO and take data out of the Rx FIFO. It will also implement an Avalon MM master interface to put data in system memory and take data out of system memory (i.e. the data plane). It will communicate with system memory via the FPGA-to-HPS Bridge, where the DMA is the Avalon MM master and the system memory is the Avalon MM slave. Additionally, the DMA will implement an Avalon MM slave interface to check control registers and set status registers (CSR - Control Status Registers) (i.e. the control plane). The CSR block will implement an Avalon MM slave interface and will be written to and read by the DMA as well as the device driver. The device driver will communicate with the CSR through the lightweight HPS-to-FPGA Bridge, where the Avalon MM master will be a device written to by the driver and the Avalon MM slave is the CSR block. The point of the CSR block is to allow the DMA to provide status information to the driver and the driver to provide control information to the DMA. The Avalon MM interfaces associated with the data plane will be 64-bits wide, but it is not certain how wide the MM interfaces associated with the control plane will be because the details of the DMA subsystem have yet to be fully worked out. The DMA controller implemented inside the Altera EMAC IP core as well as third-party DMA IP cores will be used as a reference point in the design of our DMA.

II) Software Design

Embedded Linux

We will build the architecture of our system, starting from the GSRD (Golden System Reference Design) provided with the SocKit Evaluation board. This provides us with a set of hardware and software tools and a basic configuration of the system which we can use to customize our own architecture.

From a software viewpoint, we will be using the SD card image provided by the GSRD v13.1 to boot Linux on the board[1]. The SD card image contains the precompiled Linux binaries package (version 13.1) which we will use to boot Linux on the Altera Cyclone V SoC Development board. Below is the partitioning if the SD card that we will be using:

[1] An alternative method would be to boot the board using the 1Gb QSPI flash on the board, using the SD card is a more user-friendly method chosen by the GSRD and that we will be using.

Location	File Name	Description
Partition 1	socfpga.dtb	Device Tree Blob file
	soc_system.rbf	FPGA configuration file
	u-boot.scr	U-boot script for configuring FPGA
	zImage	Compressed Linux kernel image file
Partition 2	various	Linux root filesystem
Partition 3	n/a (recognized by the MBR)	Preloader image
	n/a (recognized by the MBR)	U-boot image

fig. 8 - SD card partitioning[2]

Most of these files will most likely be modified during the course of the project. As an example, we will be modifying the Device Tree to add our new components, this will change the Device Tree Blob file (socfpga.dtb). Similarly, the modifications brought in hardware to add/modify modules will change the FPGA configuration file (soc_system.rbf). In these cases, instead of recreating the SD-card image, we will simply replace these individual files by mounting the card, copying the new version of the modified file(s), and unmounting it.

The Linux version provided in the GSRD v13.1 precompiled binaries package is Linux 3.9.0. Therefore, we will have to compile our driver such that it can run in this environment. We will see the challenges that this might lead to and how to handle them.

Cross-compilation of the driver

One of the concerns in the process of writing and running our driver on the board, is to compile it for the Linux 3.9.0 kernel which is the version of Linux that the board will boot on from the SD card. Since the kernel is too big to fit on the SD card itself, we will not be able to compile the driver on the SD card, instead we will compile it

[2] The MBR (Master Boot Record) contains descriptions of the partitions on the card, including their type, location and length

on another Linux platform and then load the compiled driver on the SD card. This platform will most likely run another version of Linux. In this case, we will have to perform cross-compilation, meaning that we will compile the driver with the source files of a kernel whose version is different (version 3.9.0 in our case) from the actual Linux kernel that runs on the platform *where* we compile the driver.

For this purpose, we will first re-compile the kernel code after modifying the Makefile so that the the source files it points to are those of the Linux 3.9.0, that we can obtain from online resources. After this step, we can compile the driver on the given platform by making sure that the executable will be compatible with a Linux 3.9.0 environment.

An alternative way to deal with the cross-compilation is to use Buildroot. This is a configurable tool that generates a set of Makefiles and patches to automatically build a bootable Linux environment for a given embedded platform. Based on the configuration parameters, it can generate a multitude of useful resources including a cross-compilation toolchain, a Linux Kernel image, a bootloader for the specified Linux environment, etc. In our case, we can use Buildroot to automate the cross-compilation process of our driver for the Linux 3.9.0.

Driver

Interaction between the Host CPU and the DMA controller

The Linux driver is in charge of configuring the DMA so that the data will be transferred to and from memory by the DMA controller with a minimal intervention from the Host system. There are two types of data that transit between the Host Linux environment and the hardware module residing on the FPGA: the first type of data is control data meant to configure the DMA, and the second is the actual useful data that is read from/written to memory directly by the DMA and with minimal intervention from the Host. Furthermore, the controller can interrupt the Host system in case of completion of packet reception/transmission (normal interrupts), or on potential error conditions (abnormal interrupts).

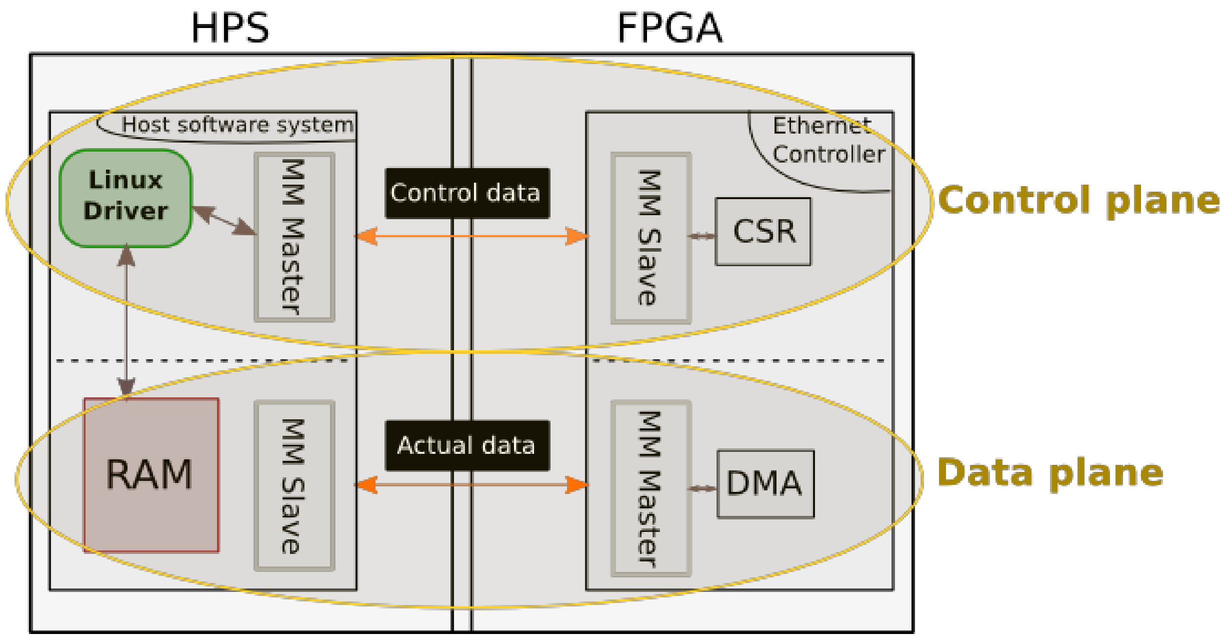

fig. 9 - Control data and actual data transfer between the Host CPU system and the FPGA

On the fig. 9, the CSR stands for the Control Status Registers. As we explained in the Hardware Design section, this is a set of control registers used by the Host system to control the status of the DMA. It is through data written in the specific (control) registers of the CSR that the DMA is configured by the Host. In particular this is how the DMA controller is made aware of the address where to read/write from/to the Host memory, whether or not operate in burst mode, when to start a given transaction, etc. Similarly, the Host is notified by the change of status on the DMA side by reading in some specific (status) registers of the CSR.

Memory data structures and descriptors

The way we structure the memory to hold the packet data written by the DMA is the following: data is read and written to and from buffers which are each “described” by a data structure that we call Descriptor in the following. There are two lists of Descriptors: one for the Reception buffers and one for the Transmission buffers. The list structure we chose for holding the Descriptors is a ring structure described in the following image:

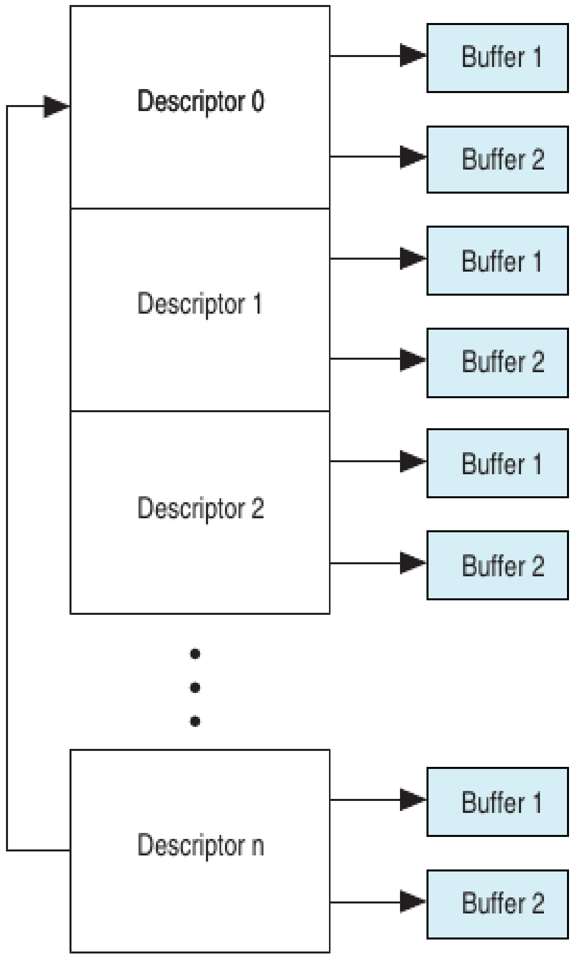

fig. 10 - Descriptor ring structure for Reception or Transmission data buffers

As we can observe on fig. 10, each Descriptor holds information about two buffers. This information includes, but is not limited to, the base addresses of the buffers in the Host physical memory system. Other types of information held by the Descriptors are, for example: information about a potential error in the IP header, or indication about the ownership status of the Descriptor which specifies whether it is the DMA or the Host who is the owner of the Descriptor at a given moment. The Descriptor list resides in the Host's physical memory and its base address is passed as control data to the Control Status Registers on the DMA side. In particular, this is how the DMA can find the addresses of the Reception/Transmission buffers to read/write packet data to/from.

Each of the buffers can contain up to one frame of data written by the DMA. These buffers contain the actual payload, their status and address information is contained in the associated Descriptor. This organization allows us to use buffers which can be allocated at arbitrary locations in the memory and which do not necessarily require contiguous memory.

Note that all data structures are addressed with their *physical* addresses. This eliminated the overhead of virtual to physical address translation and facilitates memory management.

III) Milestones

Milestone 1

Hardware: Program the FPGA with the GHRD. Add XAUI PHY IP core. Add 10GbE MAC IP core. Perform some basic tests.

Software: Create cross-compilation environment and toolchain targeted for embedded Linux. Test driver cross-compilation and installation on embedded Linux by successfully cross-compiling a basic driver, modifying the Device Tree of the system, installing the kernel module, and testing it on the board.

Milestone 2

Hardware: Implement and test Tx/Rx controllers and FIFOs. Implement and test basic DMA.

Software: Determine the data structures (descriptor + data) that will be used to relay data between the HPS, system memory, and Tx/Rx FIFOs. Implement basic driver that writes and reads from the data structures.

Milestone 3

Hardware: Complete and test the design.

Software: Implement and test the full driver.

IV) References

-) RocketBoards
(http://www.rocketboards.org/foswiki/Main/WebHome)
-) Altera Cyclone V Technical Refernce Manuals
(http://wl.altera.com/literature/lit-cyclone-v.jsp)
-) 10Gb/s packet Processing on Hybrid SoC/FPGA Platform - Embedded systems project final report
(http://www.cs.columbia.edu/~sedwards/classes/2014/4840/reports/10Gbs.pdf)