The SccKit 1.4.0 User's Guide

Revision 0.92

(Part 9)

Please read the SCC Documentation Disclaimer on the next page.

Revision History for Document

0.9	Initial revision
0.91	Added tables for IODELAY and Power Management
0.92	Added range for eMAC general configuration registers; changed atomic increment table to state that a read returns the old value, not the updated value.

Table of Contents

List of Tables

List of Figures

9 Architecture

This section describes the details of the FPGA architecture. You will find this information useful if you are writing a custom operating system to run on the SCC cores or modifying the way SCC Linux interacts with the hardware.

9.1 General Information

9.1.1 Data Types and Interfaces

The Rock Creek 2D mesh uses flits, a 144-bit wide data type to transfer information between agents. Currently the agents are the MCPC and the four eMAC interfaces.

One SCC packet can consist of 1, 2 or 3 flits. For details refer to the *SCC External Architecture Specification*. For simplicity, the SIF defines a new, 384-bit wide packet data type which maps most flit fields 1:1 as described in the SIF router section, but also dropped some of the RC mesh specific fields for simplicity. The Mesh Interface Unit (MIU) translates flits into FPGA packets and vice versa.

Another data type found in the FPGA is PCIe transaction frames. The PCIe frame header information is not used outside the host interface and further down the line to RC. Only the data payload of the PCIe transfers is further used inside the SIF and interpreted as a FPGA packet. The PCIe and SIF do not share an address range and are logically separate.

Ethernet and SATA frames are also wrapped into FPGA packets for transportation through the FPGA.

9.1.2 Address translation

The SCC and the FPGA use 34-bit addresses which are mostly mapped 1:1, the exception being the upper 2 bits in the SCC addresses. These are used in the MIU to determine the destination ID inside the FPGA for request transactions received by the FPGA from SCC.

The upper 10 bits (33:24) of the Rock Creek system address are configurable via the Rock Creek LUTs. Hence, it is possible to select a different FPGA destination for every 16 MB block represented by one Rock Creek LUT entry for all 48 cores.

Address bits [33:31] indicate the destination.

Destination	Address
MCPC	Addr[33:31] = default
FPGA MIU (debug only)	Addr[33:31] = ‘h1
FPGA Register File	Addr[33:31] = ‘h2

Table 1: The upper three bits of the system address choose between the MCPC and the FPGA

9.1.3 Transaction ID handling

The response transactions for all requests from the FPGA to the SCC do not have the original address included. Transaction IDs are used to select the right destination within the FPGA. Each request agent (the MCPC and the four eMAC modules) have their own transaction ID range.

Transaction ID	Agent
0 … 63	MCPC
64 … 79	eMAC #0
80 … 95	eMAC #1
96 … 111	eMAC #2
112 … 127	eMAC #3
128 … 143	reserved
144 … 159	reserved
160 … 255	reserved

Table 2: The transaction ID ranges for the SCC agents

Each agent generates the transaction IDs within his own range and increments the IDs with each request it sends to SCC. When an agent receives the response, it checks the ID against its transfer counter. During normal operation, the value is ID(n) = ID(n-1) + 1. In case of a mismatch, an out-of-order response has been found. In this case an error status bit will be set in the in register FPGAStatus(0) (0x08020).

9.2 Mesh Interface Unit (MIU)

The mesh interface unit receives and generates SCC packets. Because the physical link is just half the size of the on-die mesh link, the received data must be assembled into the 144-bit SCC flit format using buffers which reside in the clock domain of the physical interface. The synchronization of the data to the slower system clock domain is done via clock-crossing FIFOs (CCFs). On the SCC side this block receives and generates SCC packets and does the credit handling as described in the *SCC External Architecture Specification*. On the other side the block generates and receives FPGA packets from the different sources of the router. The FPGA packet format is very similar to the SCC packet format.

9.3 Router

The router does simple blocking one-direction packet forwarding based on the destination field. Only one packet can pass the router at a time per output port. There is a round-robin arbiter associated with every output port managing the priorities. A router packet consists of 48 bytes and contains the following fields:

- data: 256-bit data (one SCC cache-line)
- byteenable: 8-bit byte enable defining valid bytes in non-cacheline packets

- transid: 8-bit transaction identification number
- srcid: 8-bit source id, indicating the FPGA packet sender
- destid: 8-bit destination id, indicating the FPGA packet destination
- addr: :34-bit physical target address
- cmd: 12-bit command type field
- rc_id: 8-bit SCC route id field (x,y tile coordinates)
- rc_subid: 3-bit SCC srcid/destid field
- reserved:-39-bit reserved for future use

Bytes	Contents	Packet Bits
47-44	reserved[38:7]	packet[383:352]
43-40	{reserved[6:0], rc_subid[2:0], rc_id[7:0], cmd[11:0], addr[33:32]}	packet[351:320]
39-36	addr[31:0]	packet[319:288]
35-32	{destid[7:0], srcid[7:0], transid[7:0], byteenable[7:0]}	packet[287:256]
31-28	data[255:224]	packet[255:224]
	…	…
7-4	data[63:32]	packet[063:032]
3-0	data[31:0]	packet[031:000]

Table 3: Format of the router packet

The following happens for a packet transfer:

1. An agent who wants to send a packet must assert its valid out signal.
2. The router expects the destid field. If the destination port is free and has a grant for the input, the packet is accepted into the router and forwarded to the destination port.
3. If multiple agents want to transfer packets to free ports at the same time, the Round-Robin engine will determine the priority.
4. Once the destination agent has consumed the packet, the router outport is idle again, ready to process the next packet.

9.4 PCIe Interface and DMA

The PCIe establishes the connection between the SCC platform and the MCPC. It is based on the PCIe endpoint IP and a DMA application note design from Xilinx. Packets that are sent from or to the MCPC get stored in the SCEMI buffers which are 64kB each. Flow control is implemented in a way that it generates back-pressure into the SCC or to the MCPC to prevent buffer overflow.

The data transfer between the FPGA and the MCPC can be done in two different modes. In PIO mode single dwords are transferred while in DMA mode larger chunks of packets can be transferred with data rates beyond 200MB/s.

Transfers to/from memory or register space are always controlled by the MCPC and needs to be done based on FPGA packets. There is no way to transfer "raw" accesses to the FPGA, as it can handle FPGA packets only. The packets need to be assembled on the MCPC before transferring them via PIO or DMA to the FPGA.

9.4.1 MCPC Linux Driver

The driver manages the data transfer with the FPGA HW via PCIe. It implements the flow control, configures the DMA transfers and handles the data exchange with the application SW..

The following section describes how the driver handles the initialization and read and write transactions. The register references are to Xilinx IP block registers (XLX_) or system FPGA registers (RCK_). The Xilinx DMA registers are seen from the HW perspective. This means that the read registers for the DMA engine read from MCPC memory and the write registers for DMA engine write to MCPC memory; but the system view of reading and writing data from the point of view of the application SW is vice versa.

DMA transfers as described here are done with TLP size of 4 (= 16 Bytes), thus for any number of FPGA packets (48 Bytes each) a whole number of TLP transfers is done. No fragmented packets are sent. However, this approach delivers suboptimal performance.

Better performance can be achieved by using a larger TLP size (only observed power of 2 numbers work reliably), but care must be taken to pass only complete, non-fragmented packets to the application SW. Good performance is achievable with 2 consecutive DMA transfers. The first DMA transfer is with the maximum TLP size possible on the PCIe bus (as defined by HW / BIOS), and the second DMA transfer is with TLP size that results in the highest whole number of TLPs for the remaining data in the FIFO.

Driver Initialization

1. Use vendor/device ID: 8086/c148
2. Disable ASPM (Active State Power Management, ref PCIe standard) in PCIe root port of Rocky Lake connection (capability ID 0x10):

   Found PCIe Capability structure @ 0x90
   Link Control Reg @ 0xa0: 0x30410001, setting to 0x30410000
3. Map PCI BAR0 and read / check RCK_ID0_REG (bitstream ID), XLX_DCSR1, XLX_DLWSTAT, XLX_DLTRSSTAT.
4. Keep max payloadsize of PCIe endpoint in XLX_DLTRSSTAT[10:8] for later. Access to PCI BAR 0 will be needed later for DMA configuration.
5. SCEMI reset by toggling RCK_CONFIG_SOFTRESET bit in RCK_CONFIG_REG and setting RCK_TRNCT5_INIT bit in RCK_TRNCT5_REG

6. Switch device to MSI interrupt mode, register ISR that resets global waitForDMA flag.

DMA Read (SCC to MCPC)

1. Read FIFO status from RCK_TRNCT4_REG[15:0], check for error in bit [13] → fail, bailout
2. DataAvail = RCK_TRNCT4_REG[14:0] * 2 // in DWords
3. If (DataAvail == 0) return // nothing to do, no read data
4. Assert / DeAssert Reset bit in XLX_DCSR1[0]
5. Write physical address of DMA buffer (to be filled by HW) to XLX_WDMATLPA
6. TLP_size = 4 // (4 DWords), TLP_count = DataAvail / TLP_size, write to XLX_WDMATLPS and XLX_WDMATLPC respectively
7. Set waitForDMA = 1, then start DMA transfer by setting XLX_DDMACR[0]
8. While (waitForDMA) do_nothing();
9. DMA transfer is done when ISR got called and reset the global waitForDMA, data from hardware is now in DMA buffer to be passed to SCC library

DMA Write (MCPC to SCC)

1. Read FIFO status from RCK_TRNCT4_REG[31:16], check for error in bit [29] → fail, bailout
2. RoomAvail = RCK_TRNCT4_REG[30:16] * 2 // in DWords, no more than RoomAvail words can be transferred to HW w/o data loss
3. DataCount = Number of DWords to write
4. dwordsToWrite = min (RoomAvail, DataCount)
5. Assert / DeAssert Reset bit in XLX_DCSR1[0]
6. write physical address of DMA buffer (containing data to write to HW) to XLX_RDMATLPA
7. TLP_size = 4 // (4 DWords), TLP_count = DataAvail / TLP_size, write to XLX_RDMATLPS and XLX_RDMATLPC respectively
8. Set waitForDMA = 1, then start DMA transfer by setting XLX_DDMACR[16]
9. while (waitForDMA) do_nothing();

DMA transfer is done when ISR got called and reset the the global waitForDMA.

9.4.2 PCIe TX Packet Generator

The TX Packet Generator can be used to access large consecutive memory areas with a minimum of MCPC data transfer required. The module is located between the SCEMI TX FIFO (MIP) and the FPGA router. When it receives a packet with a packet generation command prefix, the content of this packet triggers a state machine that sends multiple packets to the router. The contents of these new packets are derived from the original packet and depend on the specific command prefix. Packets with a NOGEN prefix are simply forwarded to the router.

The Packet generator pauses the generation of request packets when the MIU RXFIFO or the MOP FIFO are almost full to prevent the system from being flooded with packets.

The Packet Generator auto-increments the transfer ID of all packets sent out. It uses TIDs within the MCPC range only.

Packet generator supports the following commands:

BLOCK Transaction: Generates up to 2^{32} packets based on the first one. For example, this can be used to read large chunks of DDR3 memory with just sending one request from the MCPC.

BCMC Transaction: Broadcasts the packet received from MCPC to all 4 memory controllers. For example, this can be used to copy an OS image into all 4 memory controllers.

BCTILE Transaction: Broadcasts the packet received to all 24 tiles. For example, this can be used to pull all resets in the tiles by sending just one packet from MCPC.

9.5 Interface to BMC

The FPGA has a dedicated interface to the Board Management Controller (BMC). This interface is used by the BMC firmware to initialize the FPGA as well as the SCC and therefore allows the realization of a self-booting platform which does not need an MCPC to boot. This interface is an asynchronous microprocessor interface with 16-bit data and address lines. It provides a maximum bandwidth of ~10MB, which is much less than the PCIe interface to the MCPC. You may still want to connect an MCPC to the self-booting platform (for example, when downloading of large data chunks for debugging).

The interfaces to the MCPC and to the BMC can be used exclusively. There is no hardware support to ensure a safe switch from one mode to the other during operation. The software needs to ensure that no transactions are open, before switching between the modes.

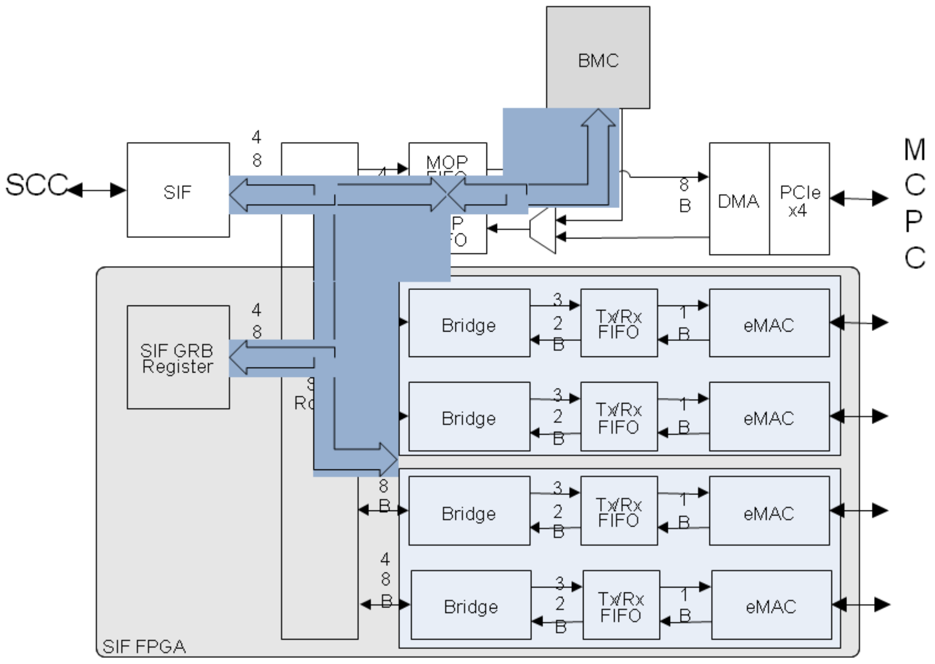

Figure 1: BMC interface

9.6 eMAC

The Ethernet interfaces are built upon IP blocks from Xilinx. The Xilinx Ethernet MAC IP contains two eMAC modules, eMAC0 and eMAC1. The IP supports different physical interfaces and for the Rocky Lake system the Tri-Speed operation RGMII v1.3 interface will be used. These physical interfaces are connected to the external PHY modules on the board.

Internally all eMAC modules have a 4 Kbyte FIFO for the TX and RX directions and are connected to the FPGA router via the client interface. Due to the limitation of the internal buffers, jumbo frames won't be supported.

The conversion from internal FPGA packet format to the Ethernet FIFO format is done by a packetizer and de-packetizer. The register interface of the eMAC IP blocks is connected to the internal FPGA register bank to provide a unified way of accessing all internal registers. The registers are mapped into the FPGA register address space.

Sending and receiving of Ethernet frames to/from SCC cores is done via buffers in DDR3 memory space. The sending core writes its frames into the buffer, informs the FPGA hardware that new frames are present. The hardware then pulls the frames from the buffer and transfers them to the external Ethernet ports. When the HW receives frames on the external Ethernet port, it puts them into the corresponding DDR3 memory buffer, tells the core that new frames are present and the receiving SCC core starts pulling the frames from the buffer. Signaling can be done sending interrupts to the SCC core. Alternatively, the core can poll the head of the buffer to determine a change in the write index.

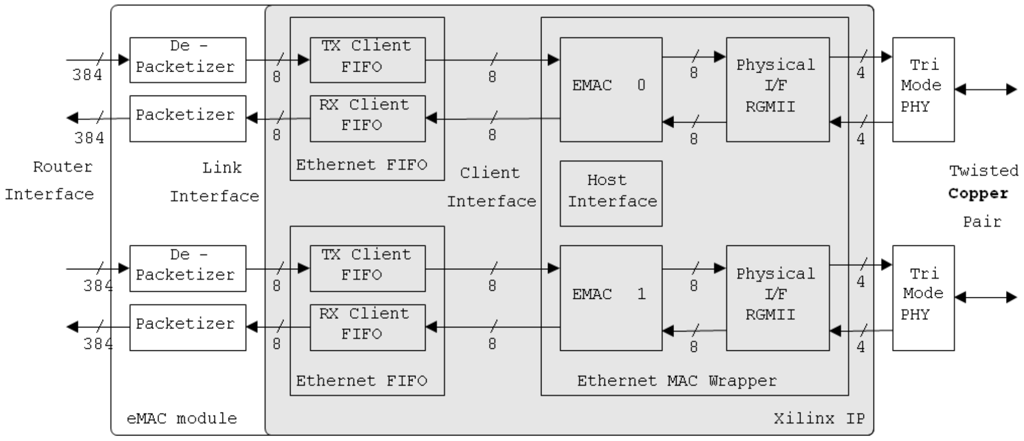

Figure 2: eMAC IP instance IP

9.6.1 Memory Buffer Structure

The receive and transmit buffers are build as ring buffers in DDR3 memory. The read and write indexes are stored in FPGA registers. The head address of the buffer is reserved to store copies of the read/write indexes so that the core can look them up by accessing the DDR3 memory instead of the real hardware registers in the FPGA.

The buffer is organized in 32-byte chunks, the size of one cache line. The main reason for this is to optimize the data transfer between FPGA and SCC by transferring 32-byte data chunks only. Ethernet frames that are stored in the buffer always start on 32-byte boundaries. When the frame length is not a multiple of 32 bytes, the remaining bytes should be filled with 0.

The start address of the buffer can be somewhere in DDR3 memory or in the on-die MPB. The size of the buffer is limited to ~2 MB. In front of each frame the first two bytes are reserved for the frame length. The hardware as well as software needsto store the frame length there as the L/T field in the Ethernet frame header is usually used for type information. The frame length value must be set to the length of the Ethernet frame and does not include the two bytes of the frame length field itself.

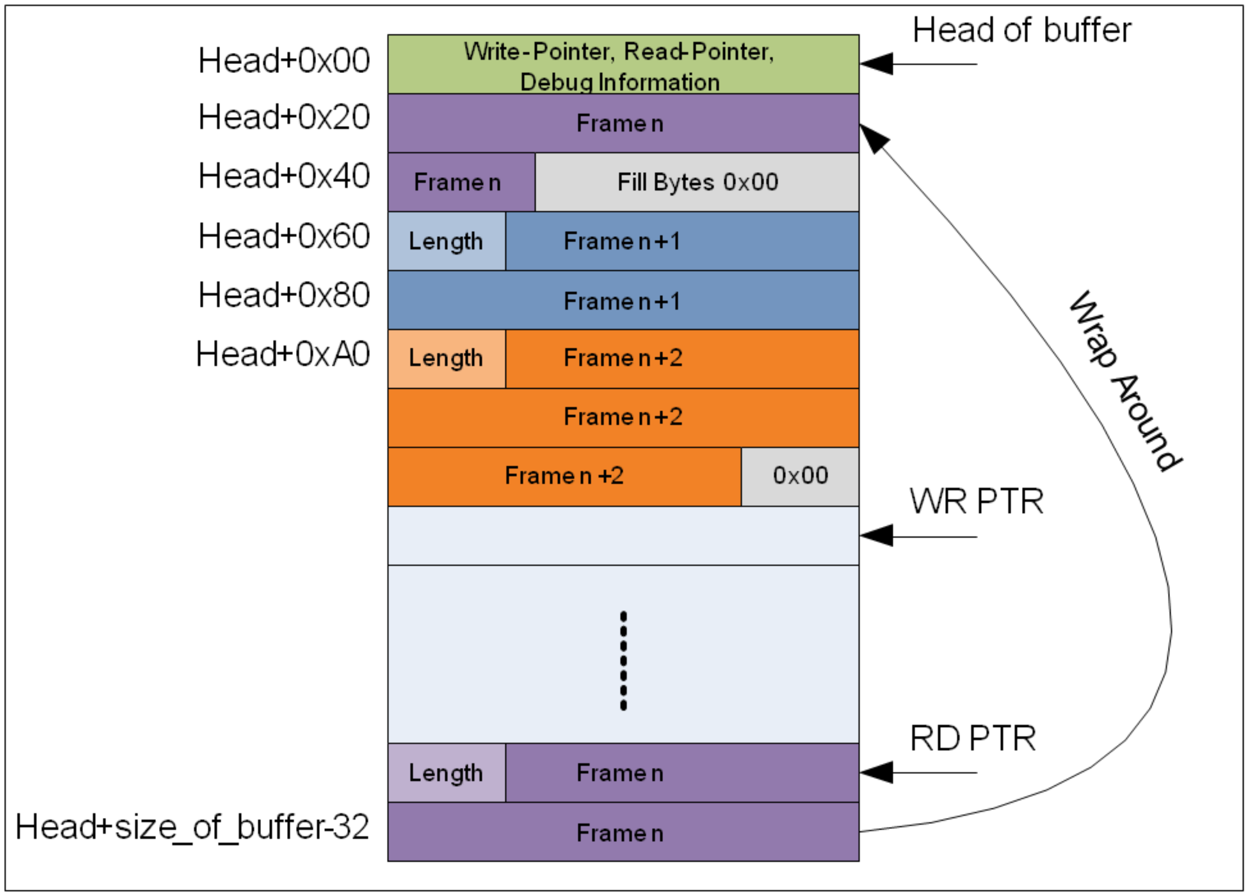

Figure 3: Ethernet Frame Buffer Organization

9.6.2 Ethernet Network Setup

Each core can have multiple independent network interfaces, connected to each of the four eMAC modules. The eMAC modules contain up to 48 independent register sets which can be shared among all cores. This results in a maximum of 192 different network interfaces with individual MAC addresses for one SCC and provides a maximum of flexibility to the software.

The software driver is responsible for configuring and enabling the network ports. Each driver does this independently for its own register sets.

1. Disable flow control, enable transmitter, set full duplex mode, set MAC speed and filter mode in Xilinx IP
2. Set up ring buffer space in DDR3 memory
3. Initialize *start address, last index, write/read index, threshold, route* and *MAC address* in FPGA register set
4. Setup signaling mode (interrupt/polling)
5. Activate network port by setting enable bit

Determining the right MAC addresses can be done in different ways. The easiest way is to use the MAC Address Base Register (0x07e00 to 0x07e04) in the FPGA and to calculate the MAC address by adding the core number to it. This address then needs to be written to the MAC address register. Using this approach requires a continuous range of MAC addresses reserved for SCC. The base address can be programmed by the BMC during boot-up of the system.

Another approach could be to store a list of MAC addresses somewhere in the OS image. The cores drivers then look them up and write them to the appropriate registers.

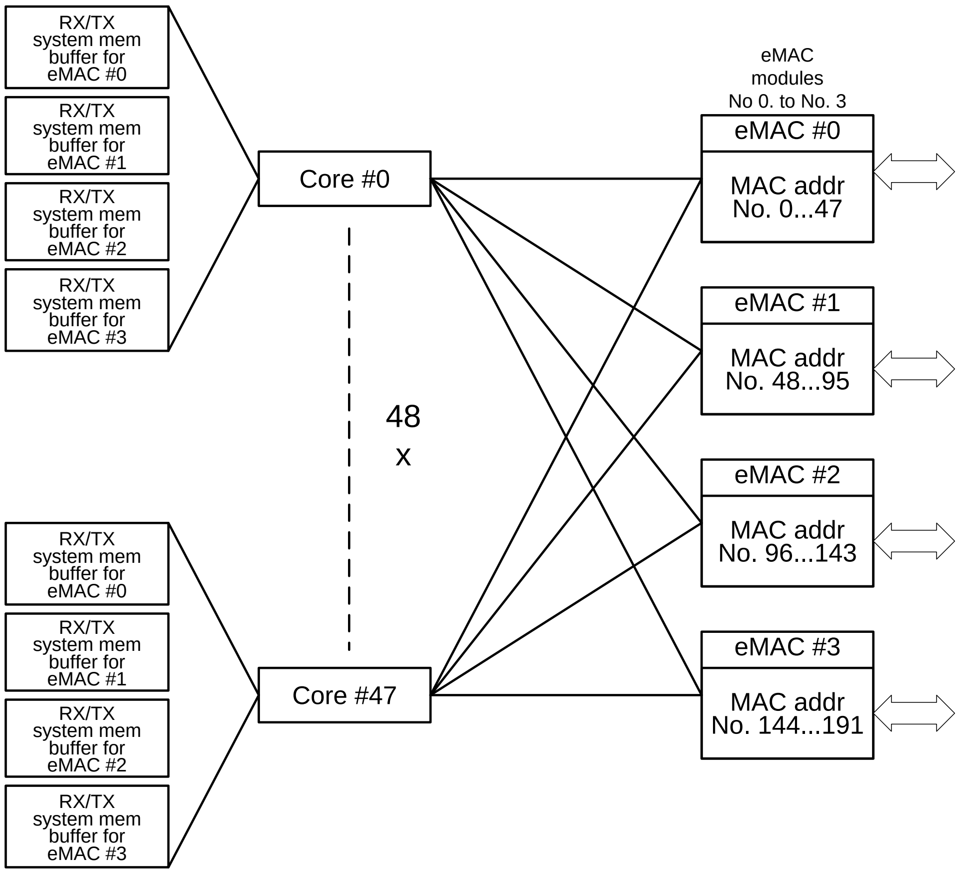
RX/TX system mem buffer for eMAC #0
RX/TX system mem buffer for eMAC #1
RX/TX system mem buffer for eMAC #2
RX/TX system mem buffer for eMAC #3
Core #0
48 x
Core #47
eMAC modules No 0. to No. 3
eMAC #0
MAC addr No. 0...47
eMAC #1
MAC addr No. 48...95
eMAC #2
MAC addr No. 96...143
eMAC #3
MAC addr No. 144...191

Figure 4: MAC address assignment

9.6.3 Receiving Ethernet Frames

Frames that are received on the external Ethernet ports are first stored in the internal 4KB buffer of the eMAC IP. When the IP block is ready to provide the data, it signals a start of frame on the client interface. The HW then starts pulling out the data from the IP block, generating FPGA packets and writing them to the DDR3 memory.

Before the HW starts generating packets, the SW must enable the network port for this core. After reset it is disabled to allow the SW on the SCC core to configure the network interface properly and setup the buffer space before the HW starts writing packets to the buffer.

There are two ways for the SCC core to determine the existence of new frames in the buffer. The first is polling. The SCC core polls the head of the buffer to find out when the write index value has changed. It can then decide when it wants to start reading out first frames from the buffer. The second approach is to use signaling via interrupt. The HW would trigger an interrupt event when it has written a new frame to the buffer.

The core SW stops reading when the buffer is empty while the HW stops writing when the buffer is full. Usually the SW should be able to read out faster than the HW can write into the buffer. Nevertheless when the buffer in DDR3 memory overflows, the HW drops packets because no additional space exists in the FPGA. A bit in the status register section of the eMAC module will be set to indicate an overflow.

When the HW receives an Ethernet broadcast packet, it will send it to all enabled register sets of the physical interface.

1. FPGA HW receives start of new frame
2. Get MAC address from frame header
3. Compare MAC address with all available addresses stored in table
    1. If no match, packet gets dropped
    2. If match, get assigned core number from table
4. Check if network port for this core is enabled
    1. If enabled, continue processing of frame
    2. If not enabled, packet gets dropped
5. Put frame data in FPGA packets and write to according core buffer
    1. Write packet length to buffer
    2. If „end of frame“ update write index and copy value to head of buffer
6. Signal interrupt to core, if in interrupt mode
7. Core starts servicing interrupt or in if in polling mode detects update of write index and starts reading frame data
8. Core updates read index in FPGA to free up buffer space

The following state diagrams describe the control flow on the core side and in the hardware

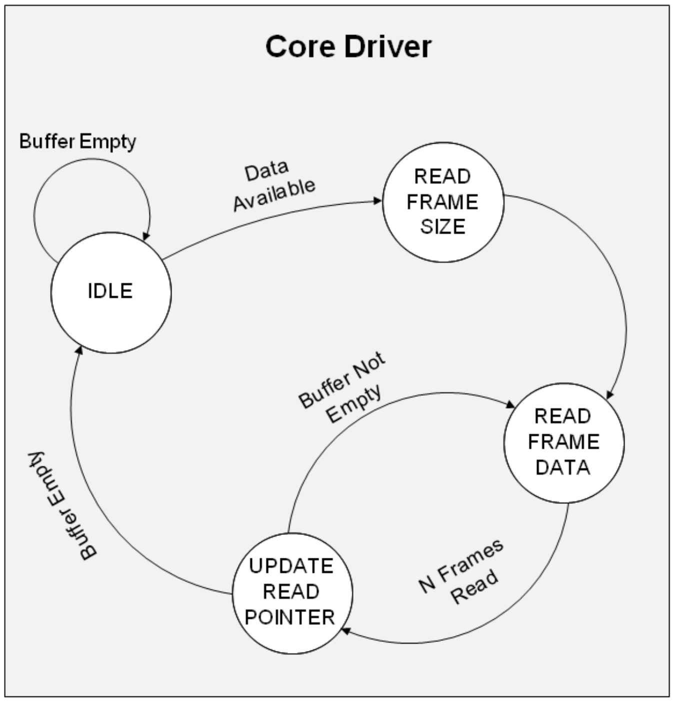

Figure 5: Receiving ethernet frames - control flow core driver

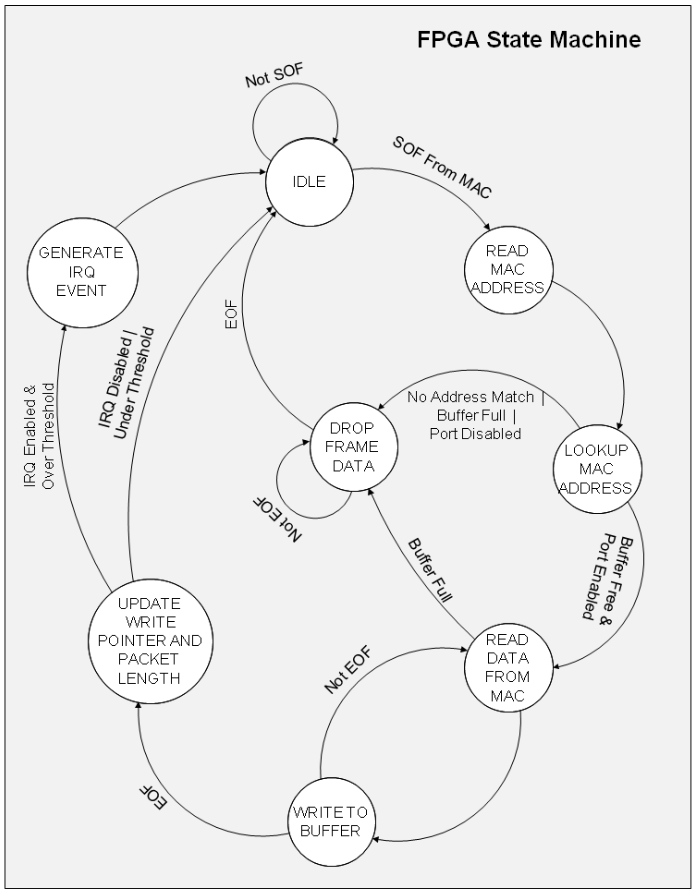

Figure 6: Receiving ethernet frames - control flow FPGA

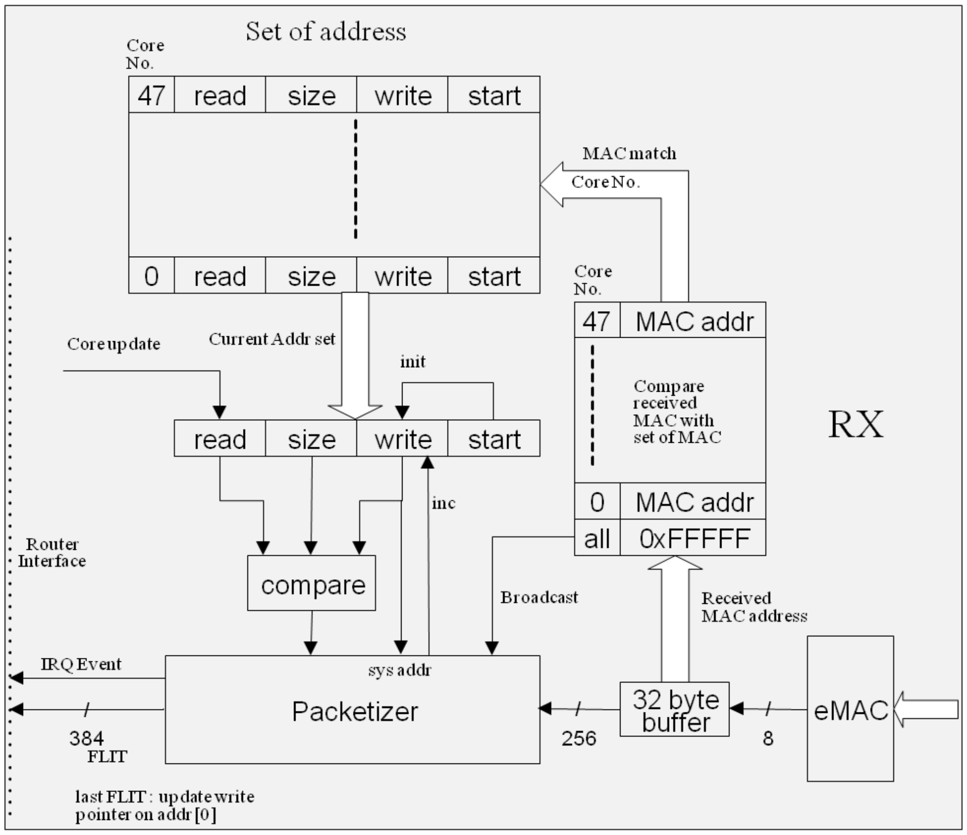

Figure 7: Ethernet packetizer block diagram

9.6.4 Transmitting Ethernet Frames

Ethernet frames that should be sent out through the external network port are written to the transmit buffer by the SCC core. The HW pulls the frames from the transmit buffer by sending 32 byte read requests, unpacks the data and transmits them to the internal buffer of the eMAC IP.

Before the hardware starts pulling data from the buffer, it must be enabled by setting the appropriate bit in the eMAC TXCTRL register. After reset all network ports are disabled.

The HW determines new packets in the buffer by looking for updates in the write index register. As soon as the value changes, the HW starts reading out new packets. The de-packetizer block extracts the frame data from the FPGA packets and transmits them to the eMAC IP block. When the buffer is empty the HW stops reading and waits for the next index register update. When the buffer is full, the SCC core stops writing new packets to the buffer.

The eMAC IP block can signal to the hardware when it is not ready to receive new data. In this case, the HW stalls until the block is ready again accepting new packets.

1. Core writes new frames to the memory buffer
2. Core updates write index in TXCTRL register to indicate presence of new frames
3. FPGA checks if network port is enabled
    1. If enabled, hardware starts reading of frame data from buffer
    2. If not enabled, hardware stalls

4. FPGA reads packet length from buffer
5. Frame data extracted from FPGA packets and send to eMAC module
    1. If last byte reached, „end-of-frame“ indication generated for eMAC
    2. Read index register gets updated
6. HW checks if other cores has data to transfer
    1. “Round-Robin” across all enabled cores – one frame per core

The following state diagrams describe the control flow on the core side and in the hardware

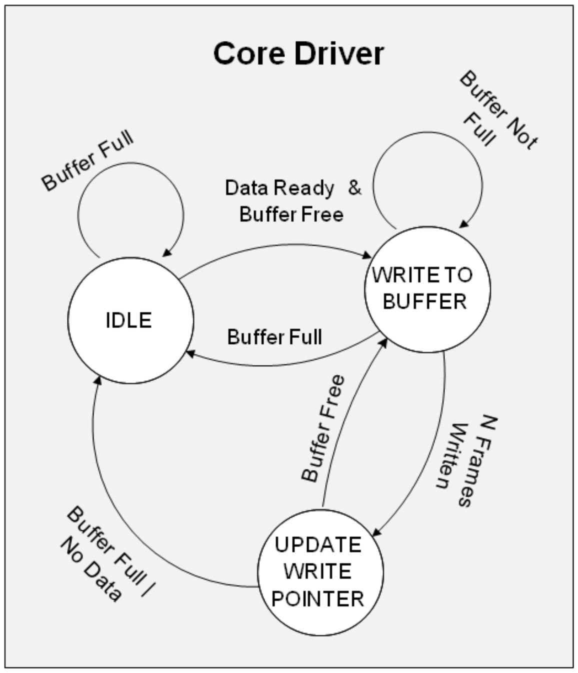

Figure 8: Transmitting ethernet frames - control flow core driver

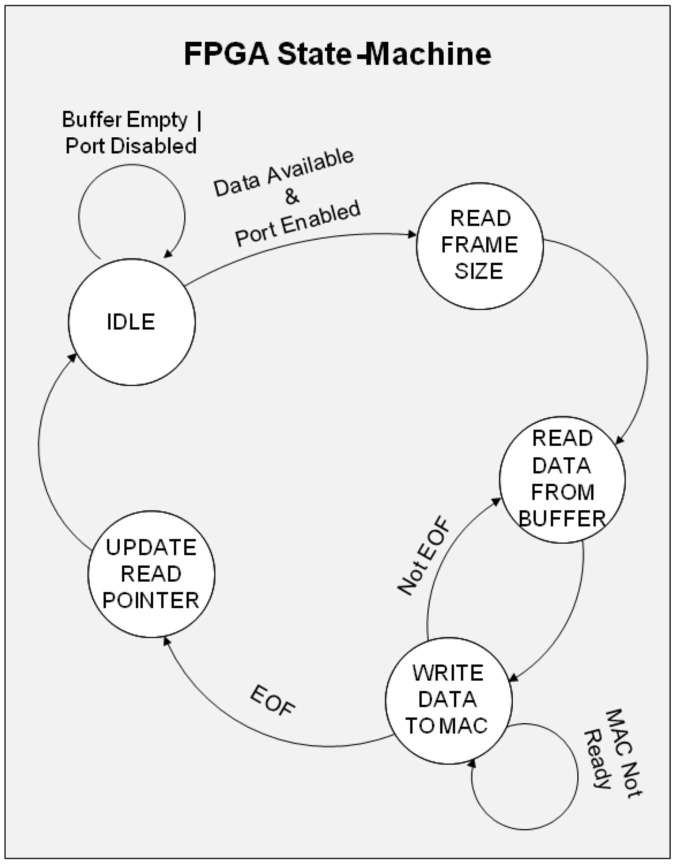

Figure 9: Transmitting ethernet frames – control flow FPGA

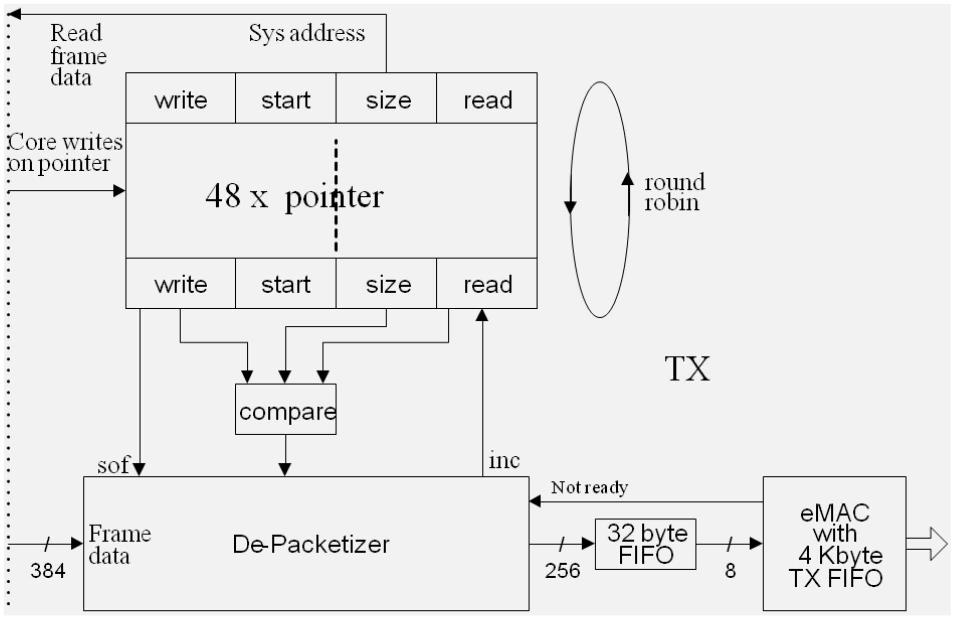

Figure 10: Ethernet de-packetizer block diagram

9.6.5 eMAC Driver Description

For the SCC Linux a driver that handles the transfer of Ethernet packets exists. The following section describes the steps that the driver takes to send and receive packets through the eMAC interfaces.

Please note that the write index registers in the RX controllers as well as the read index registers in the TX controllers are read only by the software and can be written by the hardware only.

9.6.5.1 Initialization Xilinx IP

The following description is an example for eMAC#0

Map Global Register Bank (GRB) of FPGA (0xF9000000) to local memory, length 0x10000

1. Disable TX and RX flow control (GRB+0x32C0):

   Set top 3 bits of the flow control configuration to zero, therefore disabling tx and rx flow control.

2. Setting the TX configuration bit to enable the transmitter and set to full duplex mode (GRB+0x3280):

   Half duplex (Bit 26) = 0, Transmit enable (Bit 28) = 1, Reset (Bit 31) = 0.

3. Setting the RX configuration bit to enable the transmitter and set to full duplex mode (GRB+0x3240):

   Length/Type Error Check (Bit 25) = 1, Half duplex (Bit 26) = 0, Receiver enable (Bit 28) = 1, Reset (Bit 31) = 0.

4. Setting the speed to 1Gb/s (GRB+0x3300):

   Bit 31 = 1, Bit 30 = 0.

5. Set promiscuous mode (GRB+0x3390):

   Promiscuous mode (Bit 31) = 1.

9.6.5.2 Initialization RX

1. Set RX buffer address (GRB+0x9000):

   Shift 5 times to the right and set address.
2. Read RX write index (GRB+0x9200):

   Read write index (note that the write index registers are read only by SW, thus cannot be set to an initial value).
3. Set RX write index to RX read index (GRB+0x9100):

   Set read index (both point to the same index now, thus the buffer is empty) and store value as driver internal read index.
4. Set RX last index (GRB+0x9300):

   Set last index to define buffer size (size = last index * 32).
5. Set RX route/destination (GRB+0x9500):

   Set route and destination depending on selected core (e.g. for core0 0x600).
6. Set RX hi MAC address (GRB+0x9600):

   Set high MAC address (e.g. 0x0045).
7. Set RX lo MAC address (GRB+0x9700):

   Set low MAC address (e.g. 0x414D4500).
8. Activate RX network port (GRB+0x9800):

   Set to 1.

9.6.5.3 Initialization TX

1. Set TX buffer address (GRB+0x9900):

   Shift 5 times right and set address.
2. Read TX read index (GRB+0x9A00):

   Read TX read index (note that the read index is read only for SW, thus cannot be initialized).
3. Write TX write index (GRB+0x9B00):

   Set write index (both point to the same index now, thus the buffer is empty) and store value as driver internal write index.
4. Set TX last index (GRB+0x9C00):

   Set last index to define buffer size (size = last index * 32).
5. Set TX route/destination (GRB+0x9D00):

   Set route and destination depending on selected core (e.g. for core0 0x600).

6. Activate TX network port (GRB+0x9E00):

   Set to 1.

9.6.5.4 Receiving Frame Data

1. Check for updated write index in head of buffer (or FPGA register) (RX buffer+0x00)
   a. Read 4 bytes from RX buffer at address 0 and compare the write index value to the internal read index value stored in the driver
2. Only if these indexes differ a new frame is ready and needs to be processed
   a. Read first 2 bytes (length of frame) and then the rest of data from the driver internal read index
   b. Set RX buffer read index to new position (GRB+0x9100)
   c. Set driver internal read index to new position
   d. Repeat step 2. as often as the internal read index and the write index register value still differs

9.6.5.5 Sending Frame Data

1. If enough free space is available, write frame length and data to tx buffer (tx buffer+(driver internal write index value+1)*32)
   a. Set frame length to the first 2 bytes, append the frame data
   b. Add padding where necessary
   c. Set TX write index register to new position (GRB+0x9B00)
   d. Set driver internal write index value to new position

9.7 Voltage and Current Measurement

The FPGA supports the direct measurement of voltages and currents of several SCC supply domains. Through FPGA register access, the user application can read the actual voltage/current values as well as select the domains that should be measured individually.

This feature is still under development.