AGH University of Science and Technology
Faculty of Physics and Applied Computer Science

Master thesis

Tomasz Chronowski

major: **Applied Computer Science**

specialisation: **Data Modeling and Analysis**

Programming and testing of efficient data acquisition system using Ethernet standard

Supervisor: **dr hab. inż. Bartosz Mindur**

Cracow, November 2020

Aware of criminal liability for making untrue statements I decleare that the following thesis was written personally by myself and that I did not use any sources but the ones mentioned in the dissertation itself.

Cracow, 25 November 2020

The subject of the master thesis and the internship by Tomasz Chronowski, student of 5th year major in applied computer science, specialisation in data modeling and analysis

The subject of the master thesis: **Programming and testing of efficient data acquisition system using Ethernet standard**

Supervisor:	dr hab. inż. Bartosz Mindur
Reviewer:	dr hab. inż. Bartosz Mindur
A place of the internship:	WFiIS AGH, Kraków

Programme of the master thesis and the internship

1. First discussion with the supervisor on realization of the thesis.
2. Collecting and studying the references relevant to the thesis topic(s).
3. The internship:
   - getting to know the requirements, hardware platform and software tools needed for realization of the thesis,
   - preparing basic proof of concept demo implementation of the core part of software being the subject of the thesis,
   - discussion with the supervisor focused on the results obtained and further steps,
   - preparation of the internship report.
4. Continuation of implementation of the software.
5. Testing the software.
6. Final analysis of the results obtained, conclusions – discussion with and final approval by the thesis supervisor.
7. Typesetting the thesis.

Dean's office delivery deadline: 30 November 2020

I would like to thank my supervisor
dr. hab. inż Bartosz Mindur for his support
and patience during realization of this thesis.

Contents

Chapter 1

Introduction

Gas Electron Multiplier (GEM) is a type of gaseous ionization detector used in particle physics to detect the presence of ionizing particles. One important application of GEM detectors is 2-D position sensitive measurement of charged particles which utilizes a GEM detectors with 2-D readout strips. For readout of such detectors a dedicated 2-D strip readout ASIC chips are used, one example of such chip is GEMROC (Gas Electron Multiplier Readout Chip) [1]. As its output GEMROC chip provides for each strip a separate channel consisting of analog amplitude signal and digital timestamp signal containing timestamps of amplitude measurements.

For collecting these measurements and sending them to PC for further analysis a readout system dedicated for GEMROC chips was developed [2]. This system consists of Analog-to-Digital converters for digitizing analog signals from GEMROC and FPGA module for collecting and sending data to PC. The connection between system and PC is a raw ethernet connection so communication takes place at second layer of OSI model (Data link layer) without using any higher layer protocol.

In this thesis we present a modification of this system which changes the communication protocol to UDP by replacing FPGA with Xilinx Zynq-7000 System-on-Chip which combines ARM processor and FPGA. Existing FPGA logic handling the communication with GEMROC chips and ADCs will be moved to FPGA part of Zynq chip and TCP/IP stack for UDP communication will be run on ARM part of Zynq chip.

1.1 Readout system

A simplified block diagram of readout system is shown in Figure 1.1 [2, p. 3]. The system consists of two FPGA-ADC modules, one for X coordinate and one for Y coordinate. Each module consists of two ASIC boards with two GEMROC chips on each board, custom designed ADC board containing Analog-to-Digital converters for digitizing amplitude signals from GEMROC and FPGA module mounted as a mezzanine on the ADC board for collecting and sending data to PC and for module-to-module communication. Single GEMROC chip provides 32 readout channels so the module supports total 128 channels. Each module is connected to PC via separate Gigabit Ethernet link.

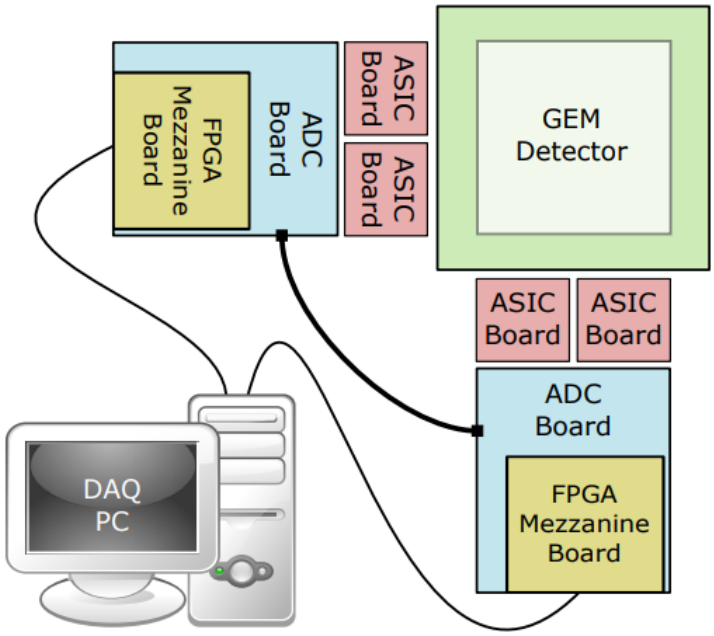

Figure 1.1: Simplified block diagram of the 2-D readout system (Source: [2, p. 3])

Block diagram of FPGA firmware is shown in figure 1.2 [2, p. 8]. Data streams received from each of the four GEMROC chips are put in separate 64-bit width FIFO queues and then combined into single 64-bit data stream by Token Ring Manager. Combined data stream is fed to Sender Module and transmitted out of ethernet interface by Embedded Ethernet MAC Wrapper. Configuration of the system is managed by embedded Picoblaze CPU which receives configuration information from incoming ethernet frames. MAC Wrapper forwards the received frames to Receiver Module which sends them to Picoblaze CPU as 64-bit data stream.

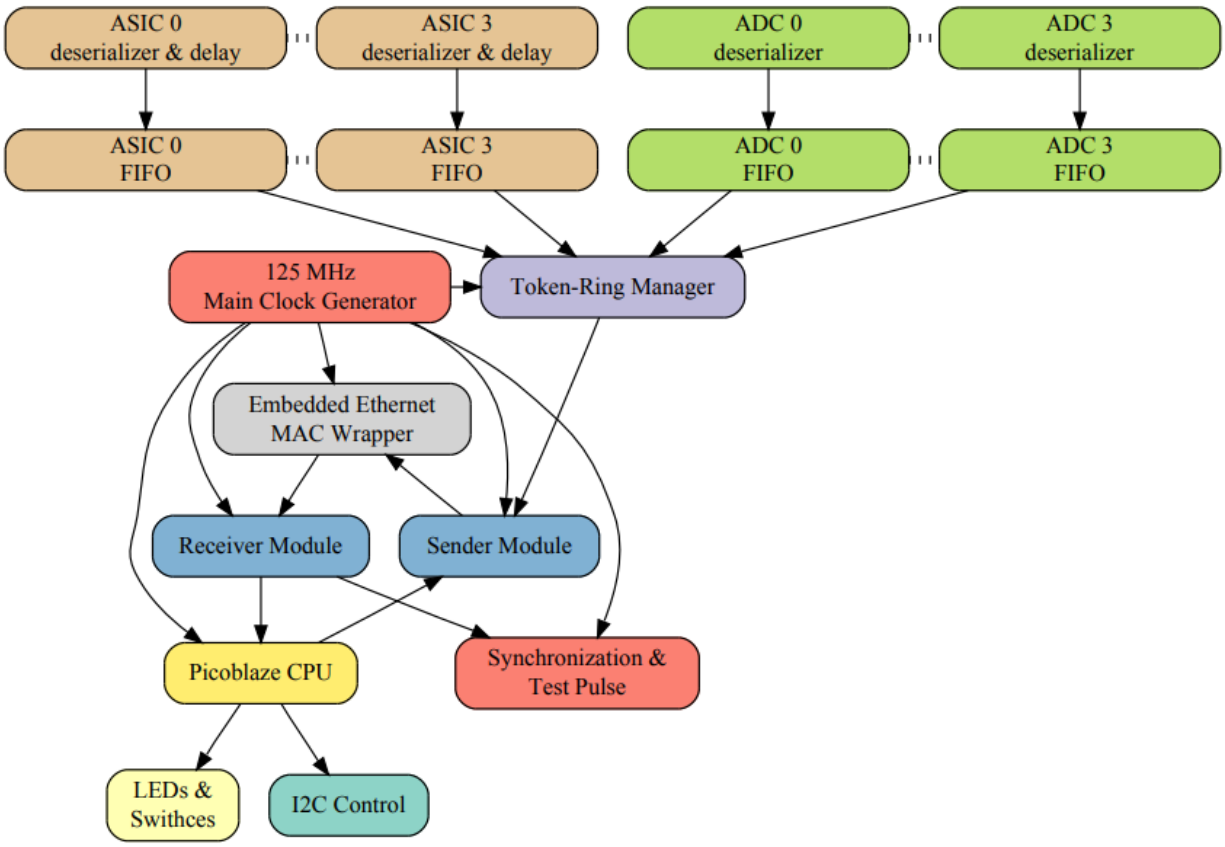

Figure 1.2: Block diagram of FPGA firmware (Source: [2, p. 8])

In modified system blocks handling ethernet transmission, namely the Receiver Module, Sender Module and Embedded Ethernet MAC Wrapper, will be removed. Data stream from Token Ring Manager will be transferred to ARM cores running TCP/IP stack where it will finally be sent to PC using UDP protocol. Similarly data from PC will be received by TCP/IP stack running on ARM cores and sent to FPGA to Picoblaze CPU.

1.2 Zynq SoC

Zynq [3] is a family of Xilinx System-on-Chips (Soc) which integrates ARM Cortex A9 processors and FPGA in single chip. This combination enables applications which needs both processing power and custom hardware logic with performance requirements which two-chip solutions cannot match due to its limited I/O Bandwidth, latency and power budget [3]. ARM processors part of Zynq chip are called Processing System (PS) and FPGA part is called Programmable Logic (PL). The high level architecture of Zynq is shown in Figure 1.3 [3, p. 6]. Main interfaces between PS and PL are:

- four General Purpose AXI Memory Mapped Ports (GP0 - GP3). These are a medium throughput ports connected directly to Central Interconnect. Looking from the PS perspective GP0 and GP1 ports are master ports and GP2 and GP3 are slave ports. Master ports GP0 and GP1 are suitable for accessing register space of peripherals implemented in PL - once peripherals from PL are connected to these ports they can be accessed as standard memory mapped I/O peripherals. The address spaces reserved for these ports are `0x40000000 - 0x7fffffff` for GP0 and `0x80000000 - 0xbfffffff` for GP1. In our system GP0 port is used for configuring all PL hardware blocks.

- four High Performance AXI slave ports (HP0 - HP3). These are a high throughput ports connected directly to OCM and DDR memories via dedicated data paths and utilizing elaborate FIFO buffering to achieve high performance. Since they are connected only to memory controllers they cannot access the other blocks of Zynq chip, for example peripherals. In our system the HP0 port is used for transferring both outgoing data stream from GEMROC chips and incoming data stream with Picoblaze configuration.

- 16 PL-to-PS and PS-to-PL interrupt lines. In our system three PL-to-PS interrupt lines are used.

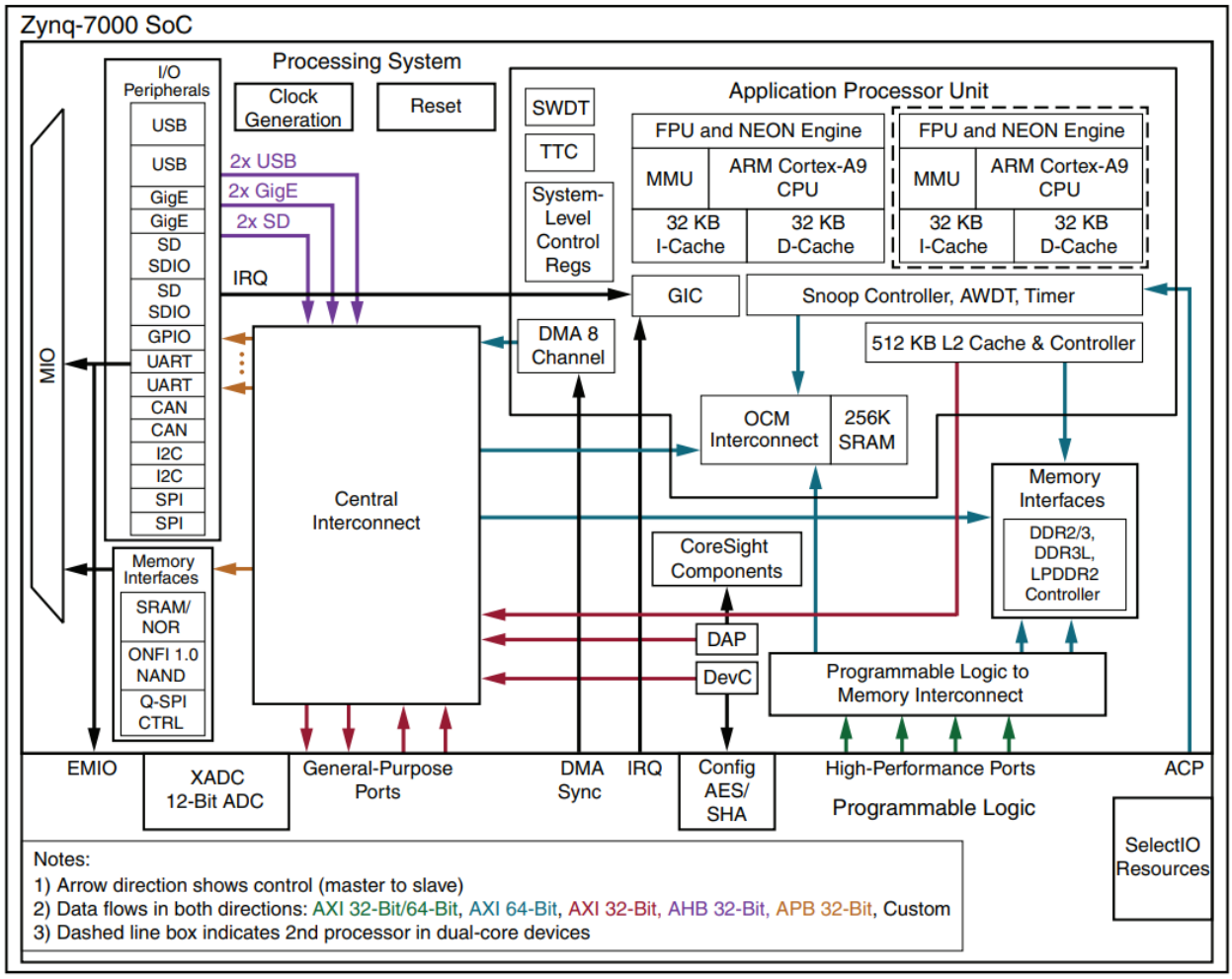

Figure 1.3: Zynq architecture overview (Source [3, p. 6])

Since the existing FPGA logic both receives incoming configuration data and sends outgoing data from GEMROCs as 64-bit wide data streams a standard AXI-Stream protocol was chosen as interface to the existing FPGA logic. Data is transferred between AXI-Strem interfaces and system memory using Direct Memory Access (DMA).

1.2.1 AXI-Stream interface

AXI-Stream is a standard communication protocol from ARM AMBA family of microcontroller buses used for transferring unidirectional data streams. It is one of the types of AXI protocol, the other types are:

- AXI - for high performance memory mapped communication,

- AXI-Lite - for low throughput memory mapped communication. This protocol is used in our system for accessing PL hardware blocks via GP0 interface.

AXI-Stream is a master-slave protocol in which single master is connected to single slave. Master is the sending side and slave is the receiving side. Protocol supports control flow and partitioning data stream into packets. The operation of protocol is shown in Figure 1.4 [4].

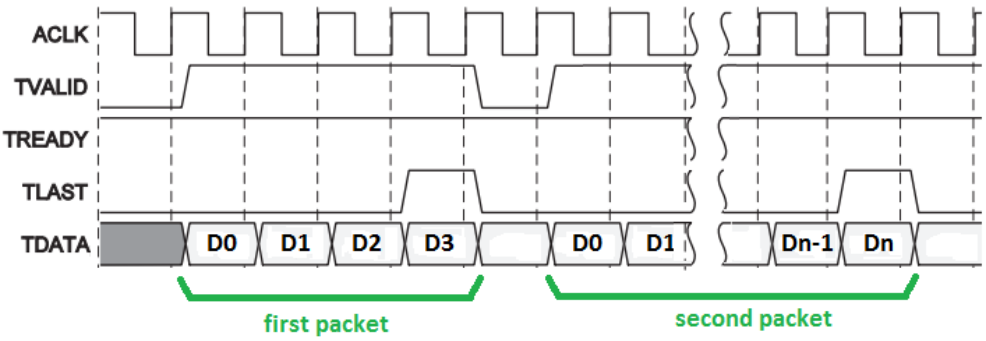

Figure 1.4: AXI-Stream protocol (Source: [4])

Main signals of AXI-Stream interface are:

- `ACLK` - clock signal,
- `TDATA` - data bus. Protocol can be configured with data bus width ranging from 8 to up to 1024 bits with multiplies of 8. In our system 64-bit width bus is used,
- `TVALID` - indicates to receiver side which bytes from TDATA bus are valid and constitutes data stream,
- `TREADY` - indicates to sender side that receiver is ready for accepting new data. This signal implements the control flow mechanism,
- `TLAST` - indicates to receiver side that this is the last byte of packet. This signal implements the packetizing mechanism.

Data transfer begins with the handshake process using `TVALID` and `TREADY` signals. At first master asserts `TVALID` signal and waits for slave to assert `TREADY` signal. From now on the data bytes are transferred via `TDATA` bus on every clock cycle and `TVALID` and `TREADY` signals remains asserted. Optionally master can assert `TLAST` signals during transfer to mark end of packets.

1.2.2 Ethernet controller

Zynq SoC contains two EEE 802.3-200 standard compliant Gigabit Ethernet Controller peripherals supporting half or full duplex transmission at 10, 100 or 1000 Mb/s rates. They support full Gigabit Media-Independent Interface (GMII) exposed through EMIO pins or Reduced Gigabit Media-Independent Interface (RGMII) exposed through MIO pins. They also provide MDIO interface for external Ethernet PHY management. Jumbo frames are not supported which slightly reduces the total data throughput. In the Zynq Mini-Module Plus board used in our system ethernet PHY is connected via MIO pins so RGMII interface is used.

1.2.3 Software stack

Xilinx provides drivers for all Zynq peripherals including Gigabit Ethernet Controller. It also provides port of lwIP library which is an open-source TCP/IP stack designed for use in embedded systems. Xilinx tools are capable of automatically generating drivers for custom hardware blocks implemented using High Level Synthesis (HLS). Zynq SoC is configured at block design level in Vivado IDE which then generates initialization code configuring Zynq SoC according to configuration options selected. This configuration specifies all the hardware configuration including which peripherals are enabled, the clock rates for each peripheral, mapping of external MIO pins for each peripheral, enabling/disabling and configuration of PS-PL interfaces (GP, HP and interrupt interfaces) etc. Basing on hardware configuration Xilinx tools generates a Board Support Package (BSP) layer including drivers, initialization and boot code and various utility functions which provides a convenient and ready to use software platform for development of bare-metal and OS applications.

In development and debug setting where JTAG is used to download and debug user application the initialization code generated from hardware configuration is executed as tcl script which uses JTAG to access Zynq registers. In production setting where application is executed from permanent media such as flash memory initialization code is provided as C routine invoked at the beginning of `main` function.

Chapter 2

Architecture and implementation

The system is implemented as bare metal application running on top of Board Support Package (BSP) layer. Bare metal solution was chosen due to high performance requirements - operating system such as Linux adds too much overhead to achieve gigabit network throughput.

The lwIP library provides two APIs - raw API which provides low level routines for TCP/IP transmission and socket API which provides BSD-style sockets. Socket API is more convenient to use but it adds additional overhead which may hinder performance. Raw API is a less convenient but it provides better performance thanks to its *zero-copy* philosophy - data for transmission is taken by ethernet controller (Emacps) directly from buffers allocated by user. In our system raw API is used to provide better performance.

The block diagram of the system is shown in Figure 2.1. Input and output interfaces to the rest of FPGA logic are 64-bit width AXI-Stream interfaces. Output interface of **Aligner** block it the output interface of the system and input interface of **Limiter** block is the input interface of system. FPGA blocks depicted with dashed lines are used for testing purpose only. Tx and Rx terms are used in the meaning of directions seen from module perspective i.e. Tx is for data send to PC and Rx is for data received from PC. System contains two data paths - Tx and Rx.

Tx data path transfers data from FPGA to PC. Data packets produced by FPGA are transferred and put into Tx circular queue by DMA recv channel from which they are pulled out and send as UDP datagrams to PC. Limiter block partitions data stream into packets which is necessary for proper DMA operation because single DMA transaction can transfer only whole packet and target buffer for DMA transfers is obviously always limited. It does so by asserting TLAST signal when length of packet reaches the configured limit. This limit

was set to maximum possible amount of data that can be transferred in single UDP datagram over ethernet link with standard MTU of 1500.

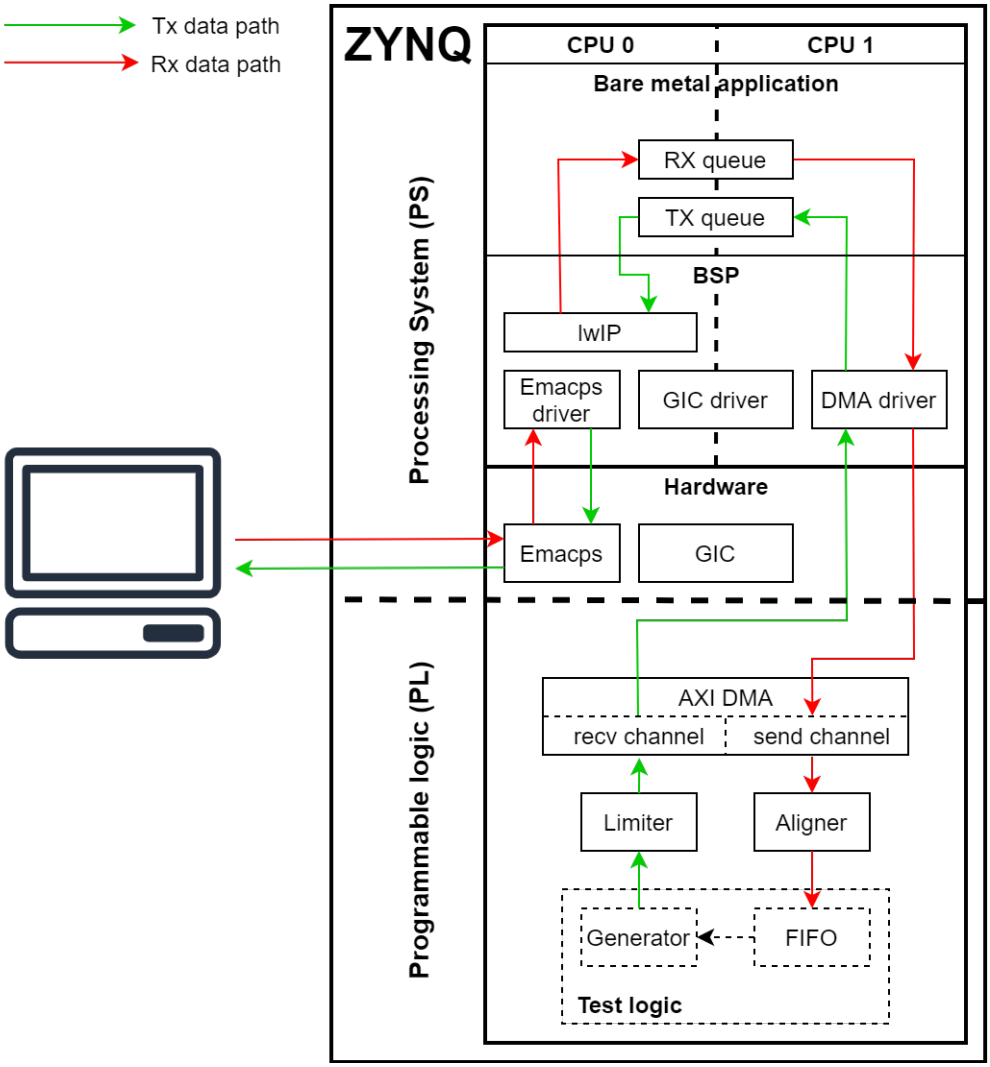

Figure 2.1: Block diagram

Rx data path transfers data from PC to FPGA. UDP datagrams sent from PC are received by lwIP stack and put into Rx circular queue from which they are pulled out by DMA send channel and sent to FPGA. Aligner block facilitates transferring uneven buffers (with size not divisible by output AXI-Stream interface width - 64 bits in our case).

Buffering data packets in Tx and Rx queues allows simultaneous UDP and DMA transfers which improves overall system throughput. To further improve performance the application is run on both ARM cores of Zynq SoC - first core runs TCP/IP stack and second core handles DMA transfers. This solution was necessary to provide maximum gigabit throughput because

frequent interrupts from DMA severely hinders the performance of lwIP stack.

Additionally a new feature was added to lwIP - cache incoherent packet buffers. It allows to create buffers with additional flag indicating that payload need not be flushed from cache to main memory before sending (TCP/IP headers are still flushed). Flushing payload is not necessary in our case because payload is filled by DMA so proper content is already present in main memory. Tests showed that cache can be flushed at rate little over 1Gb/s so this feature greatly improves performance.

It may be the case that FPGA wants to transfer data at rate exceeding maximum data rate of Gigabit Ethernet link. To prevent overwhelming network interface with data from FPGA a flow control mechanism is required. This control flow mechanism needs a way to detect that existing UDP packets are completely send out of physical ethernet port so the TCP/IP stack can accept new packets. Existing raw API of lwIP library does not provide any such mechanism so it was added as new feature to lwIP.

In test configuration two additional blocks - Generator and FIFO - are connected to the ends of Tx and Rx data paths to form a closed loop. Generator can work in two modes: TxRx and Rx. In TxRx mode it just passes input data stream to its output which allows to perform round-trip test checking whether data is transferred correctly without corruption from PC to FPGA through Rx data path and then back to PC through Tx data path. In Rx mode it ignores its input data stream and generates stream of consecutive integer numbers which allows to test Tx data path alone at maximum data rate.

System obtains IP address using DHCP protocol. If DHCP timeout occurs it is configured with default predefined IP address. System also supports discovery mechanism - the client program running on PC that wants to connect to the module can send special broadcast UDP discovery message to obtain IP address of the module.

2.1 LwIP library

LwIP (*lightweight IP*) is an open-source TCP/IP stack designed for embedded systems. It supports all basic TCP/IP protocols including DHCP and UDP which are used in our system.

To start UDP transmission UDP endpoint (in lwIP called protocol control block - PCB) needs to be created first. Each PCB handles transmission for single IP address/UDP port pair. In our system two PCBs are created - control PCB for control plane which is responsible for discovery and control of the module and data PCB for data plane which is responsible for data transfer. UDP port numbers of control and data PCBs are 7 and 8 respectively.

PCB initialization code is shown in Listing 2.1.

Listing 2.1: PCB initialization

```
/* UDP receive callbacks */
static void udp_ctrl_handler(
   void *arg,
   struct udp_pcb *pcb,
   struct pbuf *p,
   const ip_addr_t *addr,
   u16_t port);

static void udp_data_handler(
   void *arg,
   struct udp_pcb *pcb,
   struct pbuf *p,
   const ip_addr_t *addr,
   u16_t port);

...

/* pcb initialization */
struct udp_pcb *ctrl_pcb = udp_new();
udp_bind(ctrl_pcb, IP_ADDR_ANY, 7);
udp_recv(ctrl_pcb, udp_ctrl_handler, NULL);

struct udp_pcb *data_pcb = udp_new();
udp_bind(data_pcb, IP_ADDR_ANY, 8);
udp_recv(data_pcb, udp_data_handler, NULL);
```

Firstly PCB needs to be created using `udp_new` function. Then IP address and UDP port are bound to created PCB using `udp_bind` function, in our system the `IP_ADDR_ANY` is bound which means that this PCB is bound to all network interfaces present in the system (our system uses only one interface). Finally receive callback is connected to PCB using `udp_recv` function. The receive callback takes five parameters:

`arg` - user pointer,

`pcb` - PCB which this callback is connected to,

`p` - received datagram, `pbuf` type (*packet buffer*) is a lwIP data type representing packets,

`addr` - source IP address,

`port` - source port.

This callback is invoked on every received UDP datagram. To start receiving packets the `xemacif_input` function must be repeatedly called in the main loop of application. This

function invokes receive callbacks so they are invoked in the same context as `xemacif_input` function.

The code for sending UDP datagrams is shown in Listing 2.2. Firstly the packet buffer containing datagram to be send needs to be allocated using `pbuf_alloc` function and filled with data. Packet buffer contains single contiguous memory buffer for both header and payload data. First parameter tells the lwIP library for which OSI layer this packet is allocated - lwIP needs this information to allocate enough amount of memory for headers. In this case the packet buffer is allocated for transport layer (layer 4 in OSI model) because this is the layer at which the UDP protocol is defined. Second parameter is length of the buffer and third parameter determines how the buffer is allocated - in this case it is dynamically allocated from memory pool. Other allocation types are for example `PBUF_POOL` which allocates pbuf from pbuf pool which contains preallocated packet buffers with predefined size.

Listing 2.2: Sending UDP datagrams

```
u16_t length;
ip_addr_t *addr;
u16_t port;

struct pbuf *p = pbuf_alloc(PBUF_TRANSPORT, length, PBUF_RAM);
/* fill the packet buffer payload, payload pointer is p->payload */
udp_sendto(pcb, p, addr, port);
pbuf_free(p);
```

After allocating the buffer and filling it with data it can be send to IP address `addr` and port `port` using `udp_sendto` function. Finally it can be freed using `pbuf_free` function. Sending packet is a non-blocking operation so at this point `pbuf_free` call actually does not frees memory reserved for this buffer. It simply decrements the reference counter so now the only owner of this buffer is lwIP library. It will be freed only after it will be completely transmitted out of ethernet port. This means that packets must not be send at a rate exceeding network interface throughput, otherwise we can run out of memory.

The last function from lwIP API used in the system is `pbuf_realloc`:

```
void pbuf_realloc(struct pbuf *p, u16_t new_len);
```

It is used for shrinking the allocated buffer. Buffers for DMA transfers are allocated with size of 2048 bytes which provides a safe margin for MTU of 1500 bytes and after DMA transfer completes they are reallocated to actual DMA transfer length which can be smaller.

2.1.1 Options

To maximize lwIP performance some default options had to be modified.

Heap size (*lwip_memory_options* | *mem_size* option) was increased from 128 kB to 1 MB, size of pbuf pool (*lwip_memory_options* | *memp_n_pbuf* option) which contains preallocated pbuf structs was increased from 16 to 2048 and size of pbuf pool which contains preallocated 1700-byte pbufs (*pbuf_options* | *pbuf_pool_size*) was increased from 256 to 2048.

2.2 Data plane

Data plane consists of Tx and Rx data paths. To improve performance it runs on both cores - core 0 runs TCP/IP stack and core 1 handles DMA transactions. It was implemented in such a way that core 0 has exclusive access to lwIP operations (allocating, sending, and freeing packet buffers) and core 1 has exclusive access to DMA driver operations. The only point of communication between cores are the Tx and Rx queues.

It was not possible to implement whole Tx data path on first core and Rx data path on second core because lwIP library is single threaded.

2.2.1 Tx data path

Data buffers transferred from FPGA by DMA are put into Tx queue from which they are pulled out and send to via UDP to PC in main loop of core 0. The structure of Tx queue is shown in Listing 2.3 and its operation is depicted in Figure 2.2. S_{tx} is the size of queue and w_{tx} is the size of UDP transmission window.

Listing 2.3: Tx queue

```
typedef struct {
   tx_q_elem elems[S_tx];

   u32 alloc_tail;
   u32 dma_tail;
   u32 udp_tail;
   u32 udp_cmpl_tail:
} tx_q_t;
```

Tx queue maintains four pointers:

- `alloc_tail` which marks the end of *allocated* area. Buffers from this area are allocated and used as target buffers for DMA transfers,

- `dma_tail` which marks the end of *DMA transferred* area. Buffers from this area are completely filled by DMA and are used as source buffers for UDP transfers,

- `udp_tail` which marks the end of *free* area and `udp_cmpl_tail` which indicates the last buffer that was completely send out of ethernet interface. Buffers from *free* area are the buffers scheduled to transmission via `udp_sendto()` call and freed so they can be reallocated and reused as DMA target buffers. Buffers in *UDP transfer* area the buffers not yet completely send, size of this area never exceeds UDP transmission window size w_{tx}.

Main loop of core 0 continuously checks if there are free buffers available and allocates them all for DMA transfer to keep DMA busy so in practice `alloc_tail` is always one step before `udp_tail`. If there are allocated buffers remaining a new DMA transfer is issued in interrupt handler indicating completion of previous DMA transfer, otherwise `tx_done` flag indicating that no transfer was issued is set. Core 1 continuously watches for `tx_done` flag in its main loop and once it detects it is raised and there are some allocated buffers it restarts the cycle of DMA transfers by issuing single DMA transfer. If core 0 would keep allocating and sending buffers at sufficient rate `tx_done` flag would not be raised and new DMA transfers would be issued directly from DMA interrupt handler. If core 0 stops allocating new buffers `tx_done` flag would be raised and if after some time core 0 starts allocating new buffers core 1 would restart DMA transfer cycle.

Main loop of core 0 continuously checks if some buffers are filled by DMA and sends them to PC but only at most w_{tx} at a time to maintain flow control. This is where the `udp_complete_tail` pointer is used. New buffers are send only if:

$$\texttt{udp\_tail} - \texttt{udp\_cmpl\_tail} < w_{tx} \quad (mod\, S_{tx}) \tag{2.1}$$

The `udp_complete_tail` pointer is incremented in transmission complete callback.

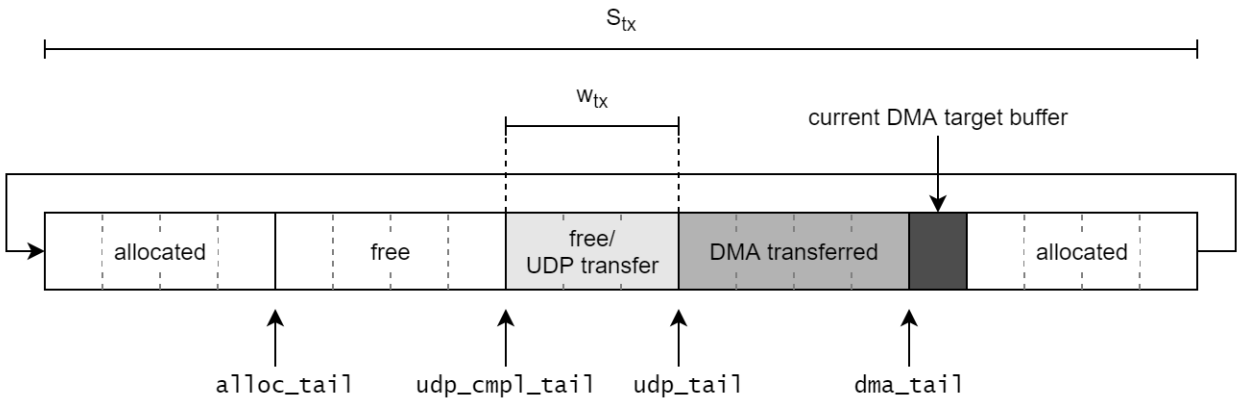

Figure 2.2: Tx queue operation

Maximizing throughput

To maximize throughput the overhead added by protocol headers should be minimized. This is achieved by sending data in UDP datagrams of size as close as possible to Maximum Transmission Unit (MTU). The standard ethernet MTU is 1500 but lwIP library actually limits MTU to 1500 - 14 (ethernet header size). Also, single UDP datagram contains data from single DMA transaction so datagram size must be divisible by 8. Taking this into account the maximum amount of data carried by single UDP datagram is:

$$
\begin{aligned}
& 1500 - 14 && \text{(lwIP MTU)} \\
- \; & 20 && \text{(size of IP header)} \\
- \; & 8 && \text{(size of UDP header)} \\
- \; & 2 && \text{(align to 8-byte boundary)} \\
= \; & 1456\,[B]
\end{aligned}
$$

2.2.2 Rx data path

Data packets received from PC via UDP are put into Rx queue from which they are pulled out and transferred to FPGA by DMA. The structure of Rx queue is shown in Listing 2.4 and its operation is depicted in Figure 2.3. S_{rx} is the size of queue.

Listing 2.4: Rx queue

```
typedef struct {
   rx_q_elem elems[S_rx];

   u32 free_tail;
   u32 dma_tail;
   u32 udp_tail;
} rx_q_t;
```

Rx queue maintains three pointers:

- `free_tail` which marks the end of *free* area. Elements from this area are empty and constitutes free space where buffers received from PC can be put,
- `udp_tail` which marks the end of *UDP received* area. Elements from this area contains buffers received from PC,
- `dma_tail` which marks the end of *DMA transferred* area. Elements from this area are completely transferred to FPGA via DMA and can be deallocated and moved to *free* area.

Similarly as in Tx data path if there are buffers remaining to be transferred from *UDP received* area a new DMA transfer is issued in interrupt handler indicating completion of previous DMA transfer, otherwise `rx_done` flag indicating that no transfer was issued is set. Core 1 continuously watches for `rx_done` flag in its main loop and once it detects it is set and there are some buffers to be transfer it restarts the cycle of DMA transfers by issuing single DMA transfer.

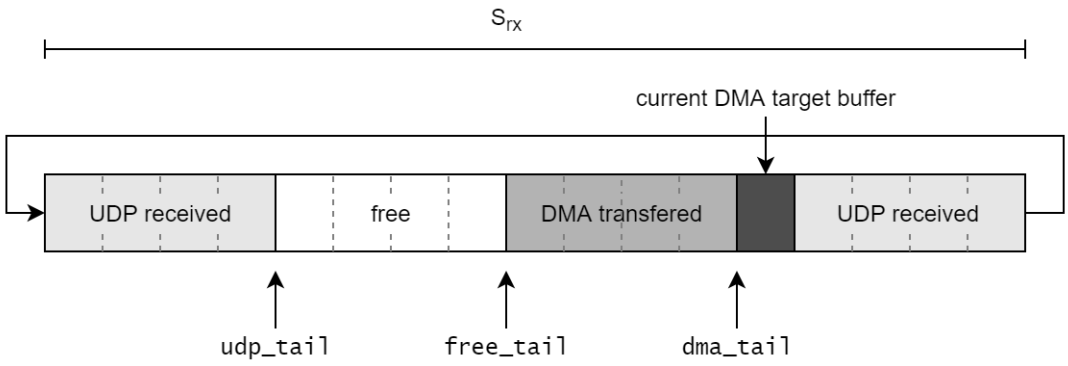

Figure 2.3: Rx queue operation

Data received from PC may possibly be fragmented for example due to IP fragmentation so Rx path must handle uneven chunks of data (with size not aligned to data stream width).

DMA engine does not support uneven transfer sizes so the remaining bytes not fitted in data stream width boundary must be kept and transferred in next DMA transfer. To achieve this without costly memory copying additional hardware block **Aligner** was added after DMA. This block allows to prepend a small chunk of $0 - 2w$ bytes of data to AXI-Stream packet where w is the data stream width. This is depicted in Figure 2.4.

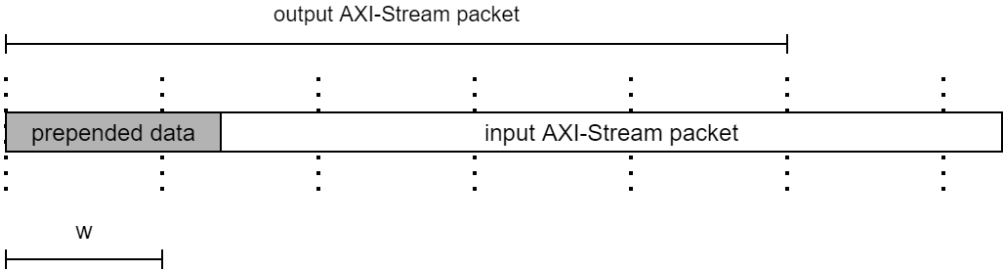

Figure 2.4: Aligner operation

Remaining bytes of uneven packet are saved and prepended to next DMA transfer. Note that part of Aligner input AXI-Stream packet is lost from output AXI-Stream packet so it also must be saved and prepended. The complete cycle of Aligner operation is shown in Figure 2.5.

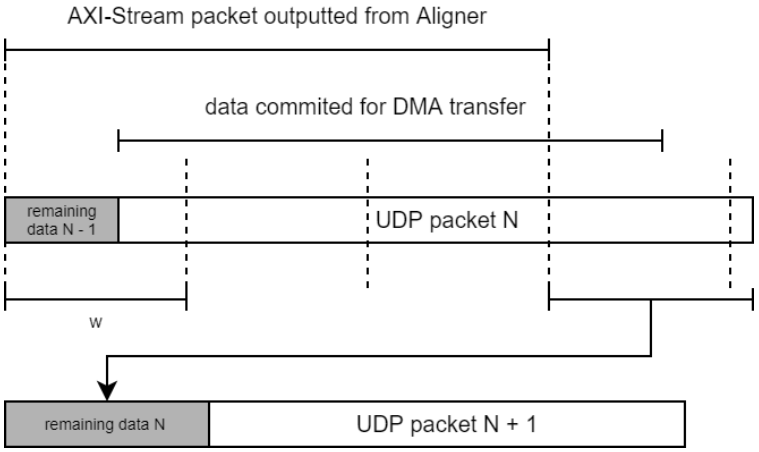

Figure 2.5: Single cycle of Aligner operation

2.2.3 Optimal implementation of queues

To maximize performance both Rx and Tx queues are stored in On Chip Memory (OCM). OCM is a 256 kB of internal RAM memory. Its access latency is comparable to L2 cache latency so it is much faster than DDR memory. First 192 kB of OCM is mapped to low

address range `0x00000000 - 0x00030000` and remaining 64 kB is mapped to high address range `0xffff0000 - 0xffffffff`. Tx and Rx queues were placed in low address range by defining additional `.ocm` section in linker script as shown in Listing 2.5.

Listing 2.5: OCM section declaration in linker script

```
MEMORY
{
   ...
   ps7_ram_0 : ORIGIN = 0x00000, LENGTH = 0x30000
   ...
}

...

.ocm : {
   *(.ocm)
} > ps7_ram_0

...
```

Variables that needs to be placed in OCM can be declared with section attribute like this:

```
static volatile rx_q_t rx_q __attribute__((section (".ocm")));
```

Additionally both Tx and Rx queues are aligned to cache line size (32 bytes in Zynq) to prevent sharing cache lines between them which reduces the overhead added by cache coherence protocol.

To further improve performance both Tx and Rx queues and are implemented in lock-free manner. This kind of circular queues do not require synchronization mechanisms such as mutual exclusion and can be implemented correctly using only memory barriers [5, p. 47]. On the producer side a memory barrier is needed during enqueueing new element between producing new element and increasing `back` pointer. On the consumer side a memory barrier is needed during dequeueing front element between consuming dequeued element and increasing `front` pointer. This is illustrated in pseudocode below:

```
/* procuder code */
elems[back] = produce_new_element();
dmb(); // memory barrier
back = (back + 1) % S

/* consumer code */
consume_element(elems[front]);
dmb(); // memory barrier
front = (front + 1) % S
```

Circular queues with multiple pointers which are used in our system can be correctly implemented in the same way. For example the listing below shows how new buffers are allocated for DMA transfers in Tx data path:

```
tx_q.elems[tx_q.alloc_tail].p = pbuf_alloc(...);
dmb(); // memory barrier
tx_q.alloc_tail = (tx_q.alloc_tail + 1) % S_tx;
```

2.2.4 LwIP improvements

Two new features were added to lwIP library - transmission complete callback and cache incoherent packet buffers. First one is needed for implementing flow control mechanism, second one improves transmission performance.

Transmission complete callback

This feature allows to register to network interface a callback function which will be called on each transmitted UDP datagram. For tracking which callback invocation corresponds to which transmitted datagram the `udp_sendto` function gets additional argument - the sequence number. Later on this number is passed as argument to transmission complete callback after datagram scheduled for sending by given `udp_sendto` call is completely sent. New sequence number can be generated using new `pseq_gen` API function. Process of sending datagram with this feature enabled is shown in Listing below:

```
pseq_t pseq;

...

void transmission_complete_callback(pseq_t pseq_) {
   if (pseq_ == pseq)
      /* datagram with sequence number pseq was completely sent */
}

...

u16_t length;
ip_addr_t *addr;
u16_t port;

struct pbuf *p = pbuf_alloc(PBUF_TRANSPORT, length, PBUF_RAM);
pseq = pseq_gen();
udp_sendto(pcb, p, addr, port, pseq);
pbuf_free(p);
```

This feature is implemented as follows. emacsps driver maintains circular queue of data buffers which are continuously pulled out by ethernet controller and transmitted out of ethernet port. Structure of this queue is depicted in the upper half of Figure 2.6. There are four areas in this queue:

- *free* - this area is a pool of data buffers available for allocation. To send a buffer it needs to be allocated first and this operation moves it to pre-work area,
- *pre-work* - this is a working area where buffers are filled with data and appropriate flags in its descriptor, for example indicating first and last buffer of packet, are set. They are then moved to *hardware* area,
- *hardware* - buffers from this area are under control of ethernet controller which uses them as source buffers for transmission. Once the interrupt indicating that some buffers from *hardware* area were transmitted is issued the driver moves these buffers to *post-work* area,
- *post-work* - here the driver examines the flags set by hardware in transmitted buffers descriptors which indicates status of transmission (for example error conditions) and frees these buffers so they are available for next allocation.

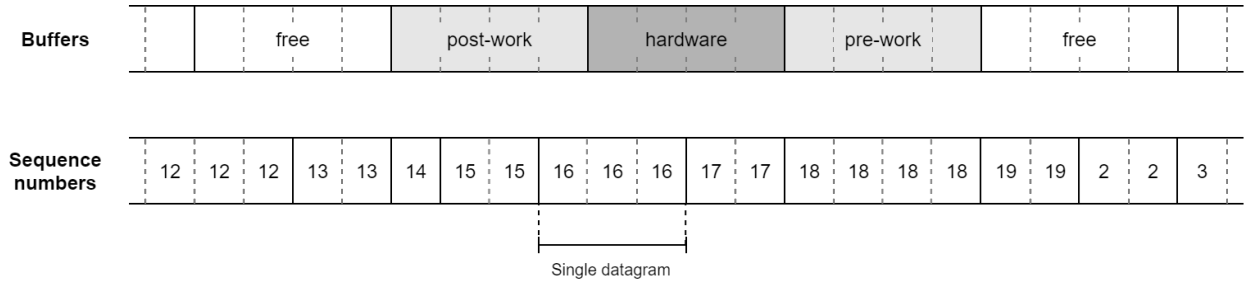

Figure 2.6: Emacps queue

Transmission complete callback was implemented using additional circular queue containing sequence numbers, parallel to existing queue containing data buffers. Every data buffer has its sequence number in sequence number queue. Sequence number n passed to `udp_sendto()` function is transferred all the way down to the lowest level routines of lwIP and stored in this queue so all data buffers corresponding to this `udp_sendto()` call have assigned sequence number n in sequence number queue (single `udp_sendto()` call can be split to multiple buffers for example in case of ipv4 fragmentation). Once the buffers are

transferred and moved to *post-work* area emacps driver iterates over them and if it detects sequence number change it calls transmission complete callback with last sequence number. For example in the Figure 2.6 transmission complete callback will be called two times with sequence numbers 14 and 15 (16 is not yet transmitted).

When packets of other protocols are transmitted the predefined sequence numbers are written to sequence number queue which allows the driver to detect and ignore these packets. Predefined sequence numbers are 1 - 4 used for ARP, ICMP, DHCP and TCP respectively. UDP sequence numbers starts from 5.

Note that placing the responsibility of sequence number generation on user side was necessary. Alternative solution would be to generate this number internally in lwIP and return it from `udp_sendto` function but this could potentially lead to situation in which transmission complete callback is called with sequence number unknown to the user in case when datagram was sent out completely before `udp_sendto` function returns.

Cache incoherent packet buffers

Ethernet controller reads data buffers to be sent from main memory so if buffer is filled by CPU and cache is enabled the cache lines containing buffer data must be flushed to main memory before transmission. In our system buffer is filled by DMA so cache flush is not needed but Xilinx port of lwIP stays on the safe side and always flushes whole buffer. Unfortunately this operation is rather slow, tests showed that cache can be flushed with speed little over 1Gb/s which seems enough for our system but this operation is blocking so it severely limits overall performance of the system.

To mitigate this problem a new feature was added to lwIP - cache incoherent packet buffers. With this feature additional flag `PBUF_TYPE_FLAG_PAYLOAD_CACHE_INCOHERENT` can be passed during buffer allocation like this:

```
struct pbuf *p = pbuf_alloc(
   PBUF_TRANSPORT,
   length,
   PBUF_RAM | PBUF_TYPE_FLAG_PAYLOAD_CACHE_INCOHERENT);
```

This flag tells the driver that payload data is already in main memory and only header data (which user has no control over) must be flushed to main memory before transmission.

This feature was implemented as follows. Structure of packet buffer is depicted in Figure 2.7. LwIP allocates pbuf struct, header data and payload in one contiguous memory block. As was mentioned earlier one of the argument of `pbuf_alloc` is the OSI layer for which this

buffer is allocated which determines amount of space reserved to header. After allocation the `payload` pointer in pbuf structure is set to start of payload and is aligned to configurable alignment size (default 64). Additional field - `payload_orig` was added to pbuf structure and after allocation this new pointer points to the same location as `payload` pointer.

As the buffer is transferred down to the TCP/IP stack `payload` pointer is moved back on every layer which it goes through to make room for header data but `payload_orig` pointer stays intact so `payload` pointer always points to header and `payload_orig` pointer always points to payload of innermost protocol (UDP in our case). When cache incoherent flag is not set the whole range of memory containing header and payload is flushed to main memory before transmission. With cache incoherent flag set only the memory range between `payload` and `payload_orig` pointers (which contains headers) aligned to cache line size boundary (32 bytes on Zynq) are flushed as depicted in Figure 2.7.

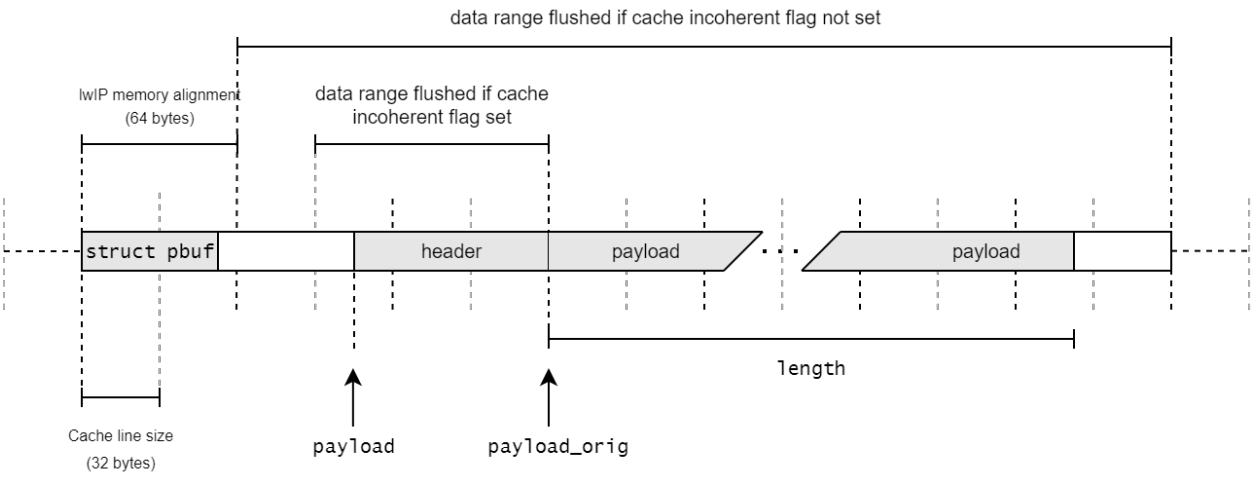

Figure 2.7: Packet buffer

2.3 Control plane

Before data transfer can be started the system must be initialized, in particular the module must obtain its IP address and the IP address of PC. The module obtains its IP address using DHCP protocol and if DHCP timeout occurs it assigns default address 192.168.1.10. The IP address of the PC is obtained using discovery protocol. PC which wants to receive data from the module sends UDP broadcast message DiscoveryReq to control plane port (7) containing module UUID which is:

```
4a765724-1fb2-11eb-adc1-0242ac120002
```

The module responds with DiscoveryRsp message containing the above UUID and at this point the PC and the module knows its IP addresses. Once PC is ready for receiving data it sends StartReq message. In test configuration this message contains the mode of the Generator block, in production this message is empty. If PC wants to stop data transfer it sends empty StopReq message. The complete operation flow is shown in Figure 2.8.

This discovery mechanism works also if multiple modules are present on the network. In this case the PC will receive multiple DiscoveryRsp messages for single DiscoveryReq broadcast message - one for each module. Also note that in case of multiple modules a working DHCP server present in the network is required to avoid IP address conflicts.

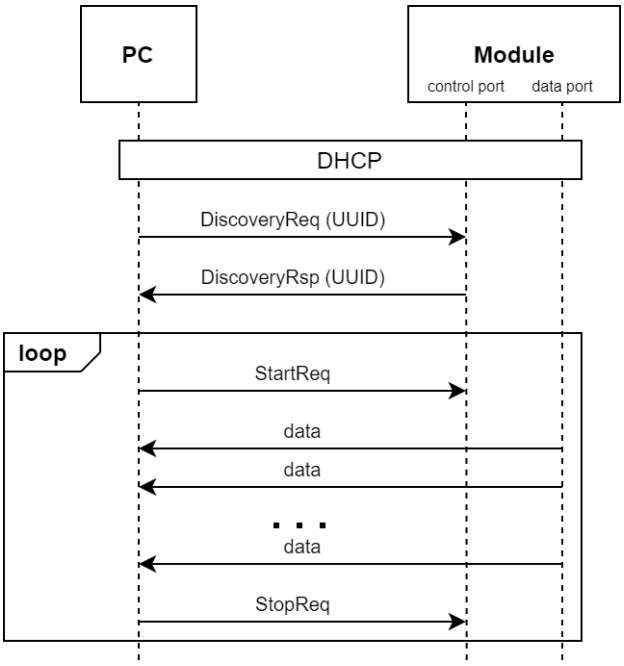

Figure 2.8: Module operation flow

The messages consists of header and payload where header contains single message id field. Single message is transferred in single UDP datagram. The general structure of messages is:

offset	size	field
0	1	message id
1	N	message payload

where message ids are defined as follow:

message	id
DiscoveryReq	1
DiscoveryRsp	2
StartReq	3
StopReq	4

The structure of DiscoveryReq and DiscoveryRsp messages payload is:

offset	size	field	notes
0	16	UUID	version 1 UUID format is used

The structure of StartReq message payload is:

offset	size	field	notes
0	1	Generator mode	used only in test configuration. 0: TxRx mode, 1: Rx mode

StopReq message contains no fields.

2.4 Running application on multiple cores

Xilinx provided solution for running multicore bare-metal application is described in [6]. The Xilinx solution is to load separate Elf binaries on each core and using regions of shared memory for inter-core communication. This solution has some disadvantages - using two separate binaries for single application is somewhat cumbersome and data shared between cores must be placed in special Elf sections so data sharing cannot be realized simply by using global variables as it would be for example in case of multithreaded OS application running on multiple cores. For these reasons a new solution providing support for executing single Elf binary on multiple cores was created.

In this new solution executing code on multiple cores was realized in such a way that each core executes `main` function providing its core id as parameter. Signature of `main` is now:

```
int main(uint32_t core_id)
```

where `core_id` is the id of the core. Cores are numbered starting from 0. Core 0 is the master core and it is the only running core after `main` function starts. Core 1 is started by core 0 explicitly via call to `start_cpu_1()` function. Typically core 0 starts core 1 once it perform all necessary global initialization, for example after initializing global variables.

After reset both cores executes BootROM code which is the stage-0 boot code stored permanently in ROM memory inside Zynq SoC. On core 0 this code ultimately transfers control to user applications, on core 1 it enters the Wait For Event mode by executing `wfe` instruction. From now core 1 is not running and waits for system event, once system event is received it reads the content of address `0xfffffff0` and jumps to that address. As a safety net after reset this address contains the address of `wfe` instruction [7, p. 160]. So to start core 1 the master core firstly needs to write address of application for core 1 to `0xfffffff0` address and then execute signal event (`sev`) instruction [7].

Before main function is entered the boot code is executed. This code performs initialization of the core hardware resources (enables MMU, L1 and L2 cache, initializes SCU etc.) and sets up environment for proper execution of C program (setup stacks, clears .bss section etc.). Some hardware resources are private to each and must be initialized by each core. Private hardware resources are:

- Memory Management Unit (MMU)
- L1 Cache
- Private Timer
- GIC CPU Interface
- Banked registers of GIC Interrupt Distributor (ICD)

Other hardware resources are shared and must be initialized only by master core. Shared hardware resources are:

- Snoop Control Unit (SCU)
- L2 Cache
- Global Timer
- Shared registers of GIC Interrupt Distributor

Similarly some steps required for setup environment for execution of C program are shared between cores and must only be invoked once and some must be set for each cpu. Shared steps are:

- clearing .bss section
- calling global constructors/destructors
- initializing global variables
- calling `exit()` function after `main` completes

Private steps are:

- setting up system and IRQ stack pointers
- preparing `argc` and `argv` arguments for `main` function

Again, only master core performs shared steps. The address of code for core 1 written by core 0 to `0xfffffff0` location is the address of vector table so once core 1 is started it invokes the same boot code as core 0 but skips the shared steps. To obtain core id at runtime the Multiprocessor Affinity Register (MPIDR) from cp15 coprocessor is read using following assembly code which reads core id to `r0` register:

```
mrc p15, 0, r0, c0, c0, 5
ands r0, r0, #0xf
```

The process of booting bare-metal application on multiple cores is shown in Figure 2.9. Greyed-out dots represents steps that are omitted by core 1. Writing address of the application for core 1 is performed at the very beginning of boot process before setting up and enabling caches to avoid the need for cache flush operation. Function `start_cpu_1` simply executes `sev` instruction:

```
void start_cpu_1() {
    __asm__ __volatile__ ("sev" : : : );
}
```

The MMU table marks DDR and OCM memory ranges as inner shareable which means that cache coherency is maintained for these memories by SCU. In ARM procedure call convention [8] first four arguments to function are passed in registers r0-r3 so the `cpu_id` argument is written to r0 register before `main` is called. Similarly, `cpu_id` is written to r0 in interrupt vectors before invoking higher level C interrupt handling routines.

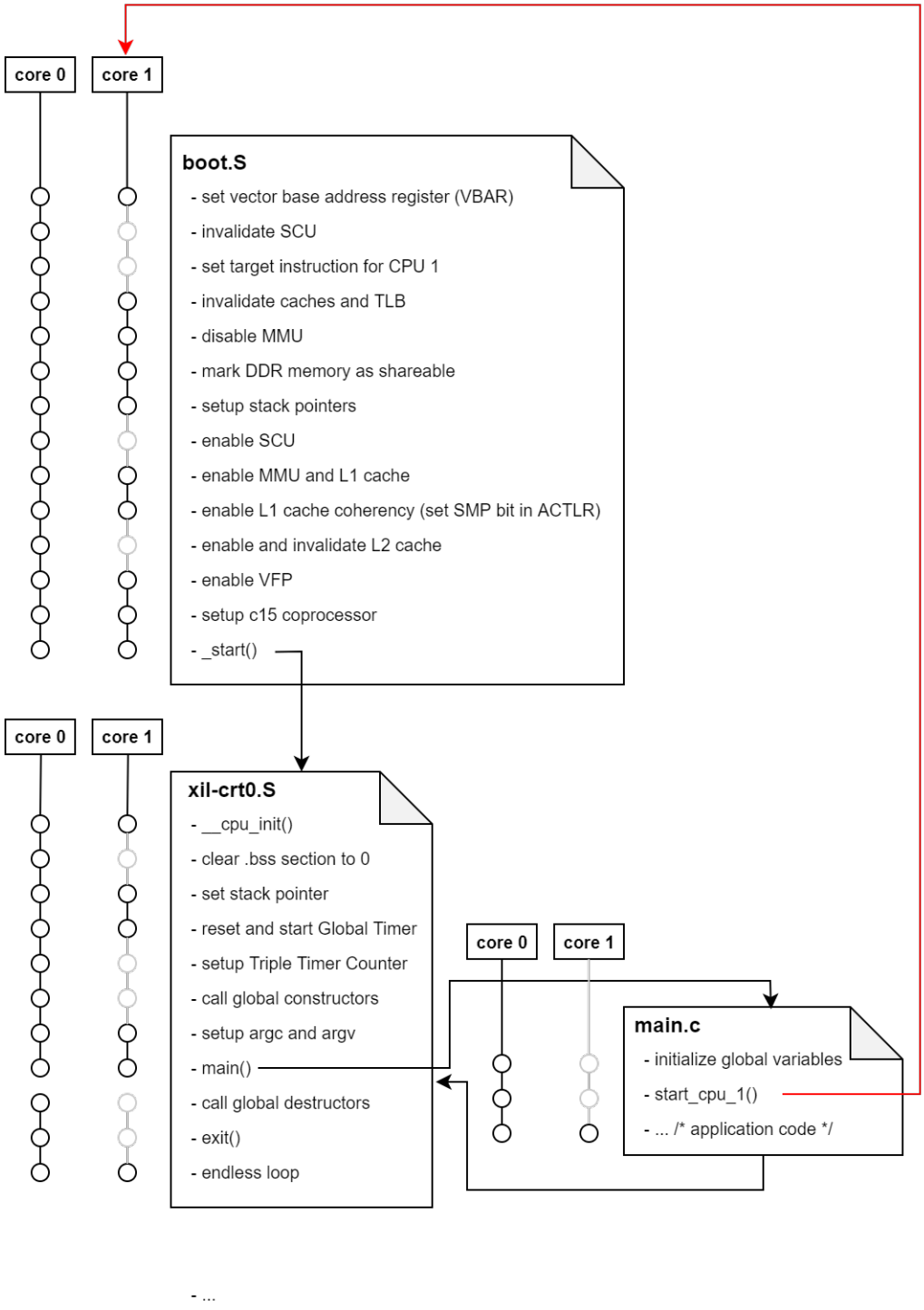

Figure 2.9: Multicore boot

2.4.1 Memory layout

Memory layout of application is shown in Figure 2.10. To ensure proper execution of code by multiple cores each core has its own system and interrupt stack. Stack for remaining CPU modes are also duplicated. Boot code sets stack pointers to appropriate location based on which core is executing it. All other sections: code (`.text`), data (`.data`, (`.bss`) and heap (`.heap`) are shared. Functions for dynamic memory allocation (malloc, free) are not reentrant but dynamic memory allocation is not used in the system (lwIP has its own allocation scheme) so heap section was leaved shared. Data sections contains data stored in both DDR and OCM memories.

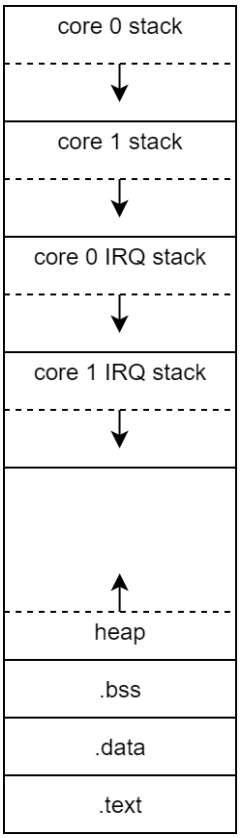

Figure 2.10: Memory layout

2.4.2 Synchronization primitives

BSP provided by Xilinx does not provide any synchronization primitives. In our system two primitives are needed: mutex and barrier. Mutex is used to protect API of drivers of shared hardware resources such as Generic Interrupt Controller (GIC) [9] and also as building block of barrier. The implementation of mutex is shown in Listing 2.6.

Listing 2.6: Mutex implementation

```
typedef struct {
   volatile u32 flag;
   bool intr_enabled[8];
   volatile u32 n;
} mut_t;

void mut_init(mut_t* m, u32 n) {
   m->flag = 0;
   for (u32 i = 0; i < n; ++i)
      m->intr_enabled[i] = false;
   m->n = n;
}

void mut_lock(mut_t* m) {
   /* disable interrupts if not in IRQ mode and they were enabled */
   u32 reg = mfcpsr();
   if ((reg & 0b11111) != 0b10010) {
      u32 cpu_id = get_cpu_id();
      m->intr_enabled[cpu_id] = (reg & XIL_EXCEPTION_IRQ) == 0;
      if (m->intr_enabled[cpu_id])
         Xil_ExceptionDisable();
   }
   isb();

   __asm__ __volatile__ (
         "1:                          \n"
         "\t ldrex r0, [%0]           \n"
         "\t cmp r0, #0               \n"
         "\t moveq r1, #1             \n"
         "\t strexeq r0, r1, [%0]     \n"
         "\t cmpeq r0, #0             \n"
         "\t bne 1b"
         :
         : "r" (&m->flag)
         : "r0", "r1", "memory"
   );

   isb();
}

void mut_unlock(mut_t* m) {
   isb();

   m->flag = 0;
   isb();

   /* reenable interrupts if not in IRQ mode */
   u32 cpu_id = get_cpu_id();
   if ((mfcpsr() & 0b11111) != 0b10010 && m->intr_enabled[cpu_id])
      Xil_ExceptionEnable();
}
```

It is classical compare-and-set (CAS) spinlock which additionally disables all interrupts. This kind of simple spinlock is not very effective [5, p. 145-146] but it was chosen because of its simplicity. More complex and effective locks are not needed in our systems because they are not used in performance critical parts of code (Tx and Rx queues are implemented as lock-free). ARM instruction set actually does not provide CAS instruction, the Load and Store Exclusive (`ldrex` and `strex`) instructions are provided instead. These instructions are more primitive than CAS instruction so they can also be used to implement CAS spinlock.

Instruction synchronization barrier (ISB) instruction is needed between disabling interrupts and executing spinlock to prevent executing of spinlock code partially by first core before interrupts was disabled which may unnecessary block the other core for the whole duration of handling interrupt by first core. For similar reasons ISB instruction is needed before clearing the flag and re enabling interrupts. ISB instructions are also needed at entry and exit points of critical section to prevent execution of protected code before spin lock is taken which would break the mutual exclusion.

The implementation of barrier is shown in Listing 2.7. Barriers are used to synchronize cores in operations that must be performed by both cores at the same time (for example GIC initialization). Implementation is based on implementation from [10]. Memory barrier instruction `dmb()` between clearing `b->cnt` counter and setting `b->flag` flag is necessary because if these operations were reordered, for example by core 0 then core 1 could exit while loop and immediately re enter barrier incrementing the counter before counter is cleared by core 0. After clearing counter by core 0 increment done by core 1 is lost. ISB instruction at the exit point of barrier is needed to prevent execution of instructions occurring in the program order after the barrier before barrier is entered.

Listing 2.7: Barrier implementation

```
typedef struct {
   volatile u32 cnt;
   bool local_sense[8];
   volatile bool flag;
   volatile u32 n;
   mut_t m;
} barr_t;

void barr_init(barr_t* b, u32 n) {
   b->cnt = 0;
   for (u32 i = 0; i < n; ++i)
      b->local_sense[i] = false;
   b->flag = false;
   b->n = n;
   mut_init(&b->m, n);
}

void barr(barr_t* b) {
   u32 cpu_id = get_cpu_id();

   b->local_sense[cpu_id] = !b->local_sense[cpu_id];
   mut_lock(&b->m);
   if (++b->cnt == b->n) {
      mut_unlock(&b->m);
      b->cnt = 0;
      dmb();
      b->flag = b->local_sense[cpu_id];
   } else {
      mut_unlock(&b->m);
      while (b->flag != b->local_sense[cpu_id]);
   }

   isb();
}
```

2.4.3 Interrupts

Similarly as in case of `main` function executing on multiple cores interrupt handlers accepts additional `cpu_id` parameter. Vector table and low level interrupt handling routines are shared between all cores - it is possible because they are stateless. Generic Interrupt Controller (GIC) driver provided by Xilinx was also extended to provide multicore support. To connect interrupt handler to particular CPU existing `XscuGic_Connect` function is used but now it accepts additional `cpu_id` parameter indicating which CPU will be handling this interrupt. To handle particular interrupt on multiple cores we need to invoke this function for each core. Initialization of GIC driver must be performed by each core because some of

GIC registers are banked (there are separate instances of these registers for each core and each core can access only its own instance of registers). Initialization is performed as series on steps, where in each step a particular group of registers is initialized. In each group some registers are shared and some are banked. Each core initializes its own banked registers and then only master core initializes shared registers.

For example one of the steps is disabling all interrupts by writing logical ones to Interrupt Clear-Enable Registers (ICDICER). ICDICER registers are 32-bit wide and each bit corresponds to one interrupt id. In Zynq there are 95 interrupt ids so three such registers are used. ICDICER0 corresponds to first 32 interrupts from which interrupts 0-15 are Software generated interrupts (SGIs) and interrupts 16-31 are private peripheral interrupts (PPIs). Both SGIs and PPIs are private to each core so ICDICER0 is banked, ICDICER1 and ICDICER2 are shared.

To ensure proper synchronization between cores barrier is invoked after initialization of each group of registers. This means that once particular core starts GIC initialization it is blocked until other cores starts GIC initialization. This is not the optimal solution but it was chosen because of simplicity. Example usage of barriers for initialization of ICDICER registers is shown as pseudocode in Listing 2.8.

Listing 2.8: Example barrier usage

```
/* configure private registers */
ICDICER[0] = 0xffffffff;

/* ensure that shared registers are initialized after all cores
   initializes theirs own private registers */
barr();

if (cpu_id == 0) {
    /* only master cpu configures shared registers */
    ICDICER[1] = 0xffffffff;
    ICDICER[2] = 0xffffffff;
}

/* ensure that next steps of initialization are started only after
   previous steps are completed by all cores */
barr();
```

2.5 FPGA side

FPGA side contains three blocks: AXI DMA, Aligner and Limiter. All three blocks are connected to PS side via AXI General Purpose interface GP0 and their interrupts are connected via IRQF2P PL-to-PS interrupt interface. AXI DMA uses two interrupt lines - one for send and one recv channel. Aligner uses one interrupt line for signalling the end of operation.

Aligner and Limiter block were implemented in C++ using High-level synthesis (HLS). Aligner implementation is shown in Listing 2.9. HLS compiler synthesizes top level `aligner` function into hardware block whose interfaces are created from function arguments and return value. Interface generation is controlled by `#pragma HLS INTERFACE` directives. In this case the argument `M_ASIX` is synthesized to output AXI-Stream interface, argument `S_ASIX` is synthesized to input AXI-Stream interface and rest of arguments and return value are synthesized to separate registers accessed through memory mapped AXI-Lite interface `S_AXI_LITE`. Synthesizing multiple arguments to single interface was achieved using `bundle` argument of interface directive. Aligner block registers space is shown below:

offset	size	field	description
0x00	8	`res0`	bytes 0-7 of prepended data
0x08	8	`res1`	bytes 8-15 of prepended data
0x10	4	`res_size`	length of prepended data

Listing 2.9: Aligner implementation

```
#include "ap_int.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"

#define CHANNEL_WIDTH 64
#define CHANNEL_WIDTH_B (CHANNEL_WIDTH / 8)
#define MASK(n) (~(~ap_uint<CHANNEL_WIDTH>(0) << n))

struct ap_axi {
   ap_uint<CHANNEL_WIDTH> data;
   ap_uint<1> last;
};

void aligner(hls::stream<ap_axi> *M_AXIS, hls::stream<ap_axi> *S_AXIS,
      ap_uint<CHANNEL_WIDTH> res0, ap_uint<CHANNEL_WIDTH> res1,
      unsigned res_len) {
#pragma HLS INTERFACE axis      port=M_AXIS
#pragma HLS INTERFACE axis      port=S_AXIS
#pragma HLS INTERFACE s_axilite port=res0    bundle=S_AXI_LITE
#pragma HLS INTERFACE s_axilite port=res1    bundle=S_AXI_LITE
#pragma HLS INTERFACE s_axilite port=res_len bundle=S_AXI_LITE
#pragma HLS INTERFACE s_axilite port=return  bundle=S_AXI_LITE

   ap_uint<CHANNEL_WIDTH> res;
   if (res_len >= CHANNEL_WIDTH_B) {
      ap_axi out;
      out.data = res0;
      out.last = 0;
      M_AXIS->write(out);
      res_len -= CHANNEL_WIDTH_B;
      res = res1;
   } else
      res = res0;

   while (true) {
      ap_axi in = S_AXIS->read();
      ap_axi out;
      out.data = (res & MASK(res_len * 8)) | (in.data << (8 * res_len));
      res = in.data >> (8 * (CHANNEL_WIDTH_B - res_len));
      out.last = in.last;
      M_AXIS->write(out);
      if (in.last)
         break;
   }
}
```

Implementation of Limiter block is shown in Listing 2.10. Limiter block has one register `len` accessed at offset `0x00` which determines the maximum length of the output packet. The unit is data stream width (8 bytes) so value 1456/8 = 182 needs to be written to this register to limit packets to desired 1456 bytes.

Listing 2.10: Limiter implementation

```
#include "ap_int.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"

#define CHANNEL_WIDTH 64

struct ap_axi {
   ap_uint<CHANNEL_WIDTH> data;
   ap_uint<1> last;
};

void limiter(hls::stream<ap_axi>* M_AXIS, hls::stream<ap_axi>* S_AXIS,
   unsigned len) {
#pragma HLS INTERFACE axis      port=M_AXIS
#pragma HLS INTERFACE axis      port=S_AXIS
#pragma HLS INTERFACE s_axilite port=len    bundle=S_AXI_LITE
#pragma HLS INTERFACE s_axilite port=return bundle=S_AXI_LITE

   unsigned i = 1;
   while (true) {
      ap_axi e = S_AXIS->read();
      if (e.last)
         i = 1;
      else if (i == len) {
         i = 1;
         e.last = true;
      } else
         ++i;
      M_AXIS->write(e);
   }
}
```

2.6 Booting the system

Boot process of Zynq Soc is described in detail in [7], here we give only a brief summary.

After reset core 0 begins executing BootROM code which main responsibility is to load and transfer control to the user application. Firstly, BootROM code reads bootstrap pins and depending on their state decides to enter either JTAG boot mode or master boot mode. In JTAG boot mode it performs only minimal system configuration and enables JTAG interface. From now on all the steps needed for running the application i.e. configuring FPGA, loading application to RAM etc. are performed via JTAG. This mode is suitable for debugging purposes but in production deployment we usually want to program (flash) the module permanently. This is where the master boot mode in which the system boots from flash memory can be used. In master boot mode BootROM reads from flash memory

the BootROM header which contains partition table. Each partition contains a different portion of the software - one partition contains user application, another one contains FPGA bitstream etc. There may be more partitions, for example when running linux kernel the bootloader and kernel image are in separate partitions. In our systems three partitions are used: one for bitstream, one for First Stage Bootloader (FSBL) and one for bare metal application. FSBL is needed to configure FPGA with bitsream before application starts (BootROM does not configure the FPGA).

BootROM searches for first partition containing user code (FSBL in our case) and then loads it into OCM memory and transfers control to it. After configuring FPGA FSBL transfers control to bare metal application. FSBL provided by Xilinx can't load applications which uses OCM memory because it rejects Elf binaries with data sections placed outside of DDR memory range. This is probably for safety reasons - before executing user application FSBL clears to zero all data sections so if any data section is placed in OCM memory the FSBL would overwrite itself because it is also placed in OCM memory. To overcome this limitation the OCM was partitioned between FSBL and application - first 176 kB of 192 kB low OCM was assigned to FSBL and the remaining 16 kB was assigned to application. This was achieved by setting OCM memory range in FSBL linker script to `0x00000000` - `0x0002C0000` and in application linker script to `0x0002c000` - `0x00030000`. Remaining 64 kB of high OCM memory is assigned to FSBL.

Chapter 3

Testing

For verification of system performance and correctness a simple test application was created in C++ language using boost Asio library for network communication. The application can perform both round-trip correctness test of Tx and Rx data paths and Tx data path performance test. At startup it performs a discovery protocol and sends the StartReq message containing Generator mode appropriate for executed test.

In round-trip test it configures Generator block in TxRx mode and then continuously sends small chunks of random data at 200ms interval to the module in sender thread. The size of data chunks is provided as test parameter. In receiver thread it receives incoming data and checks if it is the same data sent to module. UDP is unreliable protocol which can reorder the datagrams so data chunks received out of order but containing valid data are considered valid. Running this test with uneven data chunk size (not divisible by data stream width - 8 bytes) allows to verify the correctness of transferring uneven data chunks in Rx data path. Test application continuously displays sent and received data.

In Tx data path test it configures Generator block in Tx mode and then starts receiving incoming UDP datagrams. In Tx mode Generator block generates consecutive 64-bit integers starting from 5. If test applications detects the unexpected difference in two consecutive generated values (not equal to 1) it marks the generated value as invalid and adds 8 bytes to statistic of incorrectly received bytes but if this difference was detected at the beginning of datagram it reports lost datagram. Test application displays current data rate, amount of incorrectly received bytes and number of lost datagrams.

Test can be terminated by pressing Ctrl-C combination which causes test application to send StopReq message and terminate. This was achieved by implementing custom SIGINT signal handler.

3.1 Results

The 24-hour round trip test with data chunks size of 11 bytes was executed. No errors were detected which provides a strong basis for conclusion that at low data rates data is transferred without corruption in both Tx and Rx data paths.

The throughput achieved in Tx data path test was approximately 113.8 MB/s which is:

$$\begin{aligned} & 113.8 & \\ \times\ & 1024 \times 1024 & \text{(number of bytes in Megabyte)} \\ \times\ & 8 & \text{(number of bits in byte)} \\ =\ & 954623590\ [b/s] & \\ \approx\ & 0.9546\ [Gb/s] & \end{aligned}$$

The theoretical maximum UDP data rate of Gigabit Ethernet link can be calculated as follows. Assuming the maximum MTU of 1500 single ethernet frame takes:

$$\begin{aligned} & 1500 & \text{(MTU)} \\ +\ & 8 & \text{(ethernet preamble and start of frame delimiter)} \\ +\ & 14 & \text{(size of ethernet header)} \\ +\ & 4 & \text{(frame check sequence)} \\ +\ & 12 & \text{(minimal interframe gap)} \\ =\ & 1538\ [B] & \end{aligned}$$

The amount of bytes of UDP payload that can be transferred in single ethernet frame is:

$$\begin{aligned} & 1500 & \text{(MTU)} \\ -\ & 20 & \text{(size of IP header)} \\ -\ & 8 & \text{(size of UDP header)} \\ =\ & 1472\ [B] & \end{aligned}$$

So the maximum theoretical data rate is:

$$\frac{1472}{1538} \approx 0.957\ [Gb/s]$$

As we can see module can send data at $\frac{0.9546}{0.957} \approx 99.75\%$ of theoretical maximum data rate. At this rate there were no incorrectly recived bytes and approximately 1.6 datagrams lost per second which is about 0.0019% of all datagrams.

Chapter 4

Conclusions

It was demonstrated that Zynq SoC can be successfully used for implementation of high throughput data acquisition system operating at rates saturating Gibabit Ethernet link but such performance couldn't be achieved with stock BSP provided by Xilinx - many modifications must have been done in drivers, boot code, lwIP library and FSBL. Particularly surprising was the finding that data caches in Zynq SoC can be flushed at such limited rate - little over 1Gb/s which would severely limit the performance of system if there would be for example the need to do some processing of data before sending it to PC. Besides that Xilinx provides no ready-to-use solution for running single Elf binary on multiple cores so support for this must have been created from scratch. To make this solution fully functional in other applications support for multicore dynamic memory allocation must be provided. It was also necessary to implement basic synchronization primitives from scratch because Xilinx BSP does not provide these.

Two useful new features were added to lwIP library - a transmission complete callback and cache incoherent packet buffers. First one adds a control flow mechanism to lwIP UDP raw API, second one mitigates the problem of low rate of cache flush.

Bibliography

[1] T. Fiutowski, W. Dąbrowski, B. Mindur, P. Wiącek, and A. Zielińska, "Design and performance of the gemroc asic for 2-d readout of gas electron multiplier detectors," in *2011 IEEE Nuclear Science Symposium Conference Record*, pp. 1540–1544, 2011.

[2] B. Mindur, W. Dąbrowski, T. Fiutowski, P. Wiącek, and A. Zielińska, "A compact system for two-dimensional readout of gas electron multiplier detectors," *Journal of Instrumentation*, vol. 8, pp. T01005–T01005, jan 2013.

[3] *Zynq-7000 SoC Overview*. Xilinx, 2018.

[4] "Xilinx axi stream tutorial - part 1." `http://fpgasite.blogspot.com/2017/07/xilinx-axi-stream-tutorial-part-1.html`. Online; accessed: 2020-11-25.

[5] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[6] *Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors*. Xilinx, 2014.

[7] *Zynq-7000 SoC Technical Reference Manual UG585 (v1.12.2)*. Xilinx, 2018.

[8] *Procedure Call Standard for the Arm Architecture*. ARM, 2020.

[9] *ARM Generic Interrupt Controller Architecture Specification, Architecture version 1.0*. ARM, 2008.

[10] "Implementing barriers." `http://15418.courses.cs.cmu.edu/spring2013/article/43`. Online; accessed: 2020-11-07.