# DMA implementations for FPGA-based data acquisition systems

Presenter: Wojciech M. Zabołotny
Institute of Electronic Systems
Warsaw University of Technology

# FPGA in DAQ

- FPGA chips are a perfect solution for interfacing the FEE in the DAQ systems
  - Flexible communication interfaces (either supported with dedicated cores or possible to implement in the programmable logic)
  - Possibility to operate in hard real-time. No problems with interrupt latencies. It is possible to achieve fully deterministic precise timing.
- There are some disadvantages
  - High cost of FPGA based solution
  - Difficult implementation of more complex data processing algorithms
  - Difficult implementation of more complex communication protocols, especially of those related to buffering and repeated retransmission of huge amount of data (e.g. TCP/IP)
- Solution?

## FPGA + „PC” in DAQ

- The solution is to use the standard computer „PC” or „ES” as early as possible in the DAQ chain.
- Possible architectures include:
  - Using SoCs (e.g. Xilinx Zynq, ZynqMP, Altera SoC FPGAs)
  - Using FPGAs „tightly coupled” with the computer system via high speed interface – e.g. PCIe
- The problem is the efficient delivery of data from the FPGA part to the memory of the computer.
- To spare the CPU computational power for the real processing of data, usage of DMA is advisable.

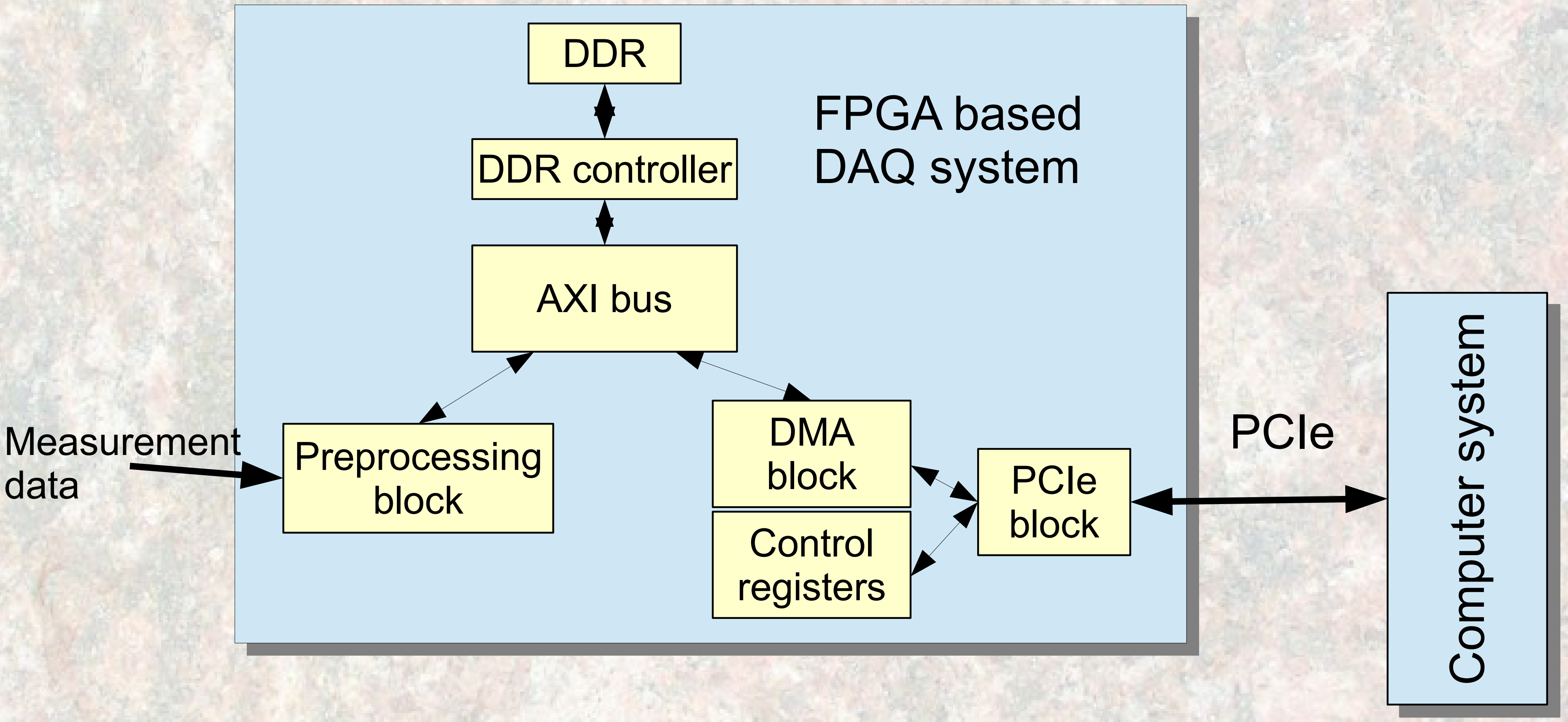
## DMA solutions - embarras de richesse

- There are various portable solutions available, often for free
  - https://opencores.org/project,wb_dma
  - https://opencores.org/project,dma_axi
  - https://opencores.org/project,virtex7_pcie_dma
- There are different DMA IP-cores provided by the FPGA vendors, optimized for their FPGA hardware
- The FPGA implementation offers us an exceptional oportunity to prepare a DMA system carefully adjusted to the specific requirements of the particular DAQ
- The following examples were developed for Xilinx FPGAs (Family 7 or UltraScale+)

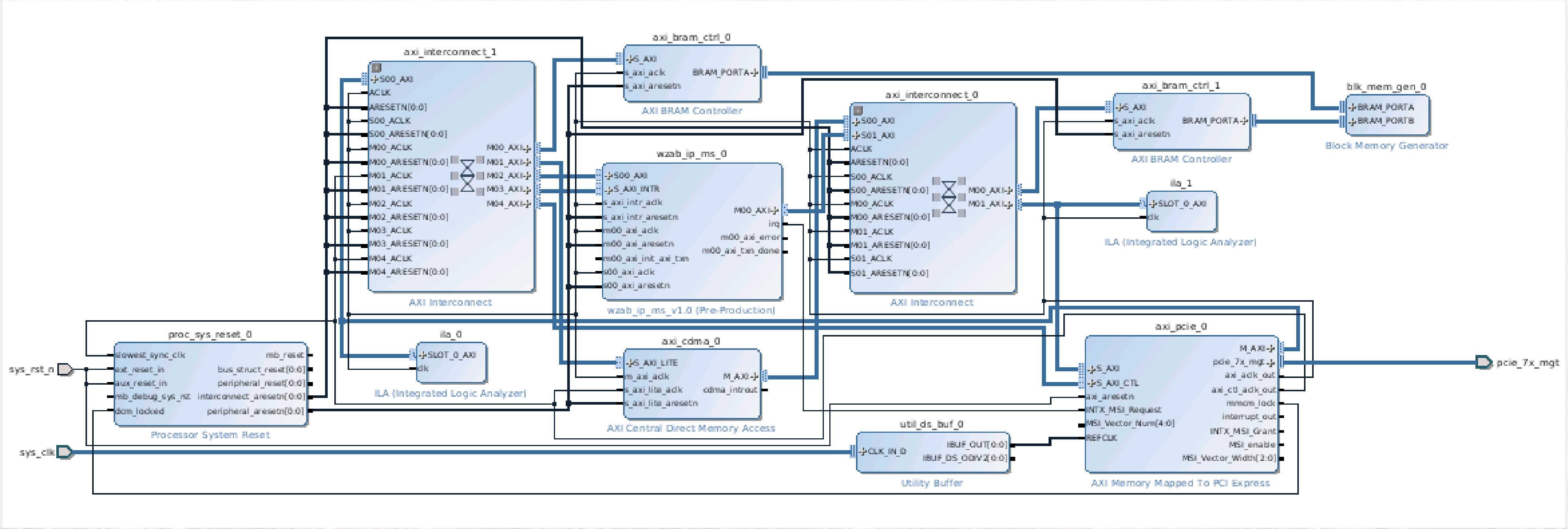
# The first system

- The system was created for the GEM detector DAQ. The hardware platform was the KC705 board.
- The FPGA receives the data from FEE, preprocesses it, and stores the result in the huge DDR4 memory.
- The data must be read from that memory via the PCIe interface.
- This solution is well suited for situations where the avarage data bandwidth is moderate, but it is fluctuating.
- In that architecture the natural solution was to use the AXI Central DMA Controller and the AXI Memory Mapped to PCI Express Gen2 IP cores.

# Implementation of the first system



# Results

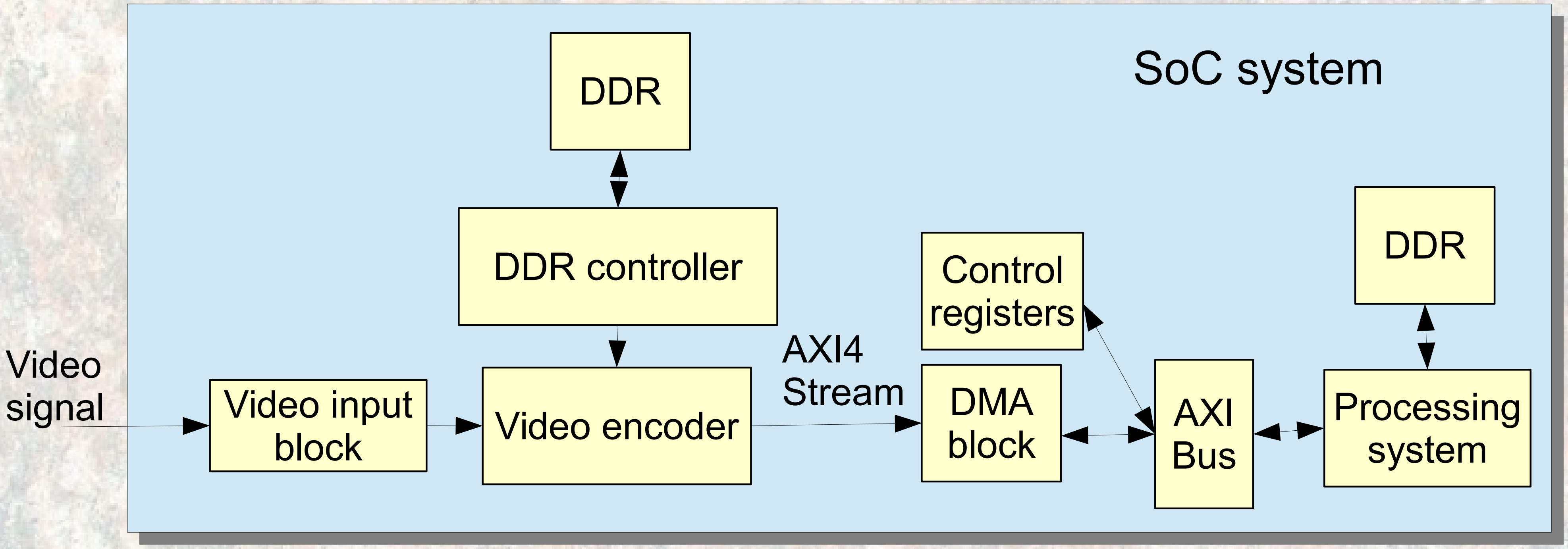


- The implementation can be easily performed in the Vivado Block Diagram Editor
- The Linux driver allowed to allocate the DMA buffer and to mmap it into the applications memory.
- The theoretical throughput of AXI and PCIe was 16Gb/s and of AXI. The maximum achieved throughput was 10.45 Gb/s for writing to DDR and 8.05 Gb/s for reading from DDR.
- For the continuous stream of the data the memory bus may be a bottle neck...

## The second system

- The hardware platform was the ZCU102 board containing both the FPGA and the ARM CPU (SoC)
- The second system was created for the acquisition of data from the hardware video encoder (VSI project)
- The data was delivered by the AXI4 Stream interface
- The data should be written to the memory of the PS connected via AXI4 interface.
- Each fragment was delivered in a separate AXI4 Stream packet, but due to the compression the packets length could differ.
- The natural solution seemed to be the AXI DMA controller

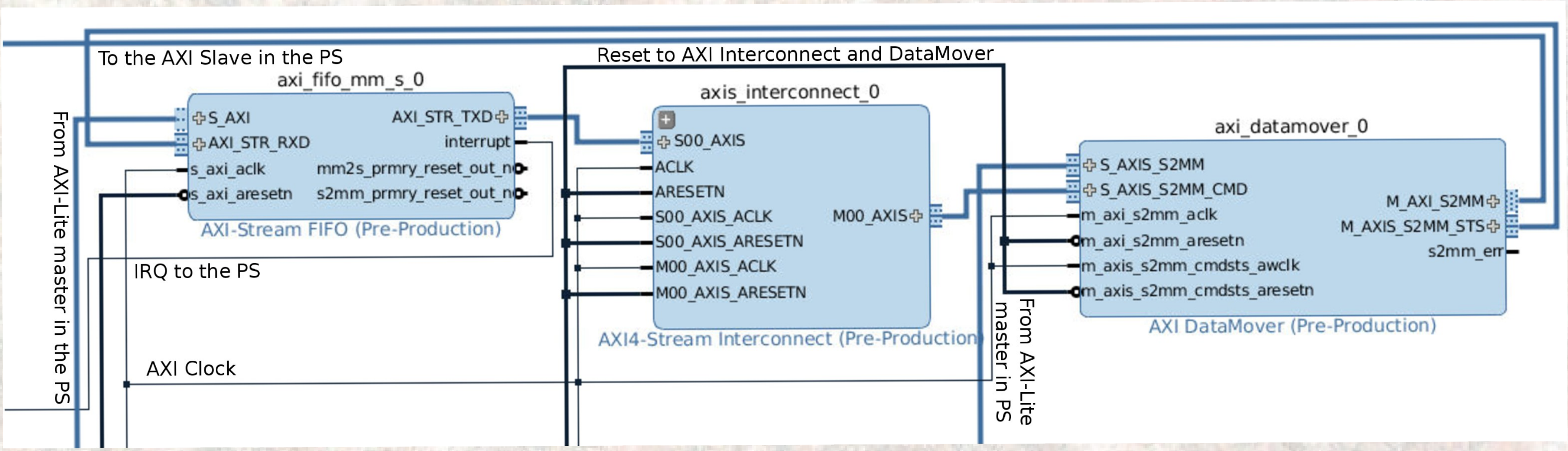
# Topology of the second system



# Problems...

- To receive continuous stream of data, it was necessary to use the controller in a circular mode.
- Unfortunately, the AXI DMA Controller with the original Linux kernel didn’t report correctly the length of the last transfer.
- Thorough investigation has shown, that it may be difficult to reliably fix the problem. (The register holding the length of the transfer gets overwritten when the next transfer starts)
- The good alternative was to use the AXI Data Mover
  - The transfer commands are delivered by AXI4 Stream
  - The status of transfers are delivered back by another AXI4 Stream interface. There is no risk to loose the the information about the length of the transfer!
- How to feed the ADM with the commands, and to receive statuses?
  - The AXI Streaming FIFO is the good choice...

# Implementation with the Xilinx blocks



- The implementation allows to avoid the „buffer overrun" problems.
- There are a few (16) DMA buffers (mmapped to the applications memory), and the transfer request for each buffer is generated in advance and written to the FIFO.
- After the status of the particular transfer is received, the data is delivered to the application for processing.
- Only after the application confirms, that the data is processed, the transfer request may be resubmitted to the FIFO

# Linux driver API

- The DMA buffers are mapped into the application's memory. The length of the single buffer must not be smaller than the maximum length of the frame.
- Communication with the driver is performed via ioctl calls:
- ADM_START - Starts the data acquisition.
- ADM_STOP - Stops the data acquisition.
- ADM_GET - Return the number of the next available buffer with the new video frame. If no buffer is available yet, puts the application to sleep.
- ADM_CONFIRM - Confirms that the last buffer was processed
- ADM_RESET - This command resets the AXI Data Mover and AXI Streaming FIFO. It is necessary before the new data acquisition is started to ensure that no stale commands from the previous, possibly interrupted transmission are stored in those blocks.
- The ADM_GET and ADM_CONFIRM ioctls ensure the appropriate synchronization of the access to the DMA buffers.

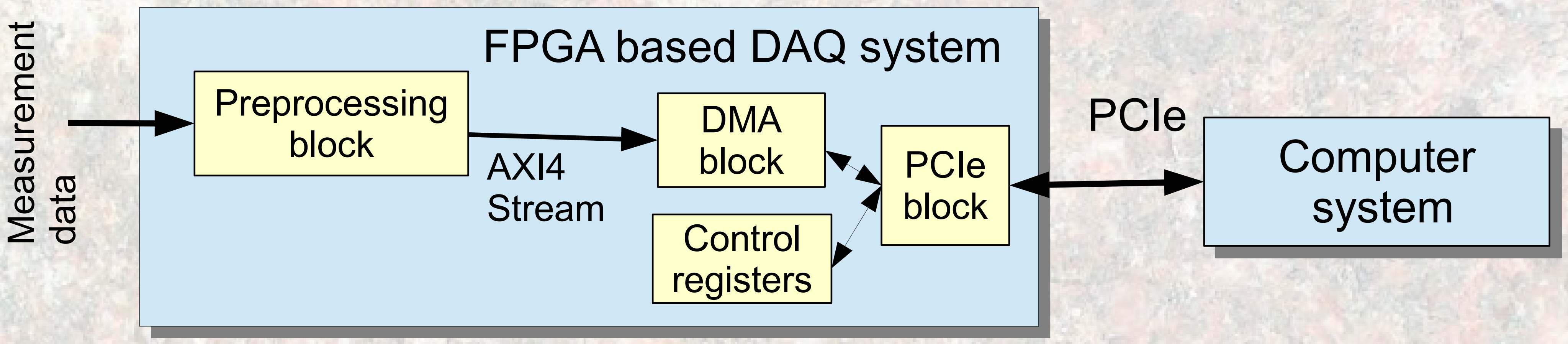
## Results

- The DMA system and the driver was carefully tested, and is currently used in the VSI system.
- Due to the specific features of the data source no maximum throughput tests were performed.
- It was stated, that even at the maximum frame size of 4MB and frame rate of 60 fps, the CPU load realted to reception of data was below 1%.

# The third system

- The third system combined the features of the first two.
- The hardware platform was a purpose-developed Artix-7 based PCIe card.
- It was the DAQ for the same GEM detector measurement system used in case 1, but now configured for the continuous operation. Therefore, the DDR buffering of data was useless…
- The data was delivered by the AXI4 Stream interface, but the packets could be bigger than any reasonable single DMA buffer.
- Therefore it was necessary to use another architecture

# Topology of the third system



- The IP-core used as a DMA engine and PCIe block was the Xilinx DMA for PCIe also known as XDMA.
- The block supports 64-bit addressing at the PCIe side, so it could be used with huge (above 4GB) sets of DMA buffers.
- The block is so complex, that it was practically necessary to use the driver provided by Xilinx. Unfortunately, it required certain modifications...

# Driver corrections

- The original driver supported the cyclical transfer only with read/write operations – no zero-copy transfer was possible
- For cyclical transfer the driver didn’t implement any overrun protection
  - The driver checks the „MAGIC number” of the transfer request
  - After the transfer is finished, its status is written back to the memory as „metadata writeback” with another „MAGIC number”.
  - It is possible to configure the same transfer request and writeback addresses. So the status overwrites the request, and blocks a possibility to perform the same transfer again.
  - After the application processes the data, the transfer request should be rewritten, with the „MAGIC number” written as the last word. That ensures that the overrun condition will generate a transfer error.
- Another problem was related to handling of huge data in a circular buffer

## Buffer mapping

- Received data are organized in structures for direct access from the C-language
- The scattered DMA buffers were mapped so that they create a huge continuous buffer in a virtual address space
- To allow efficient direct processing – caching was switched on for the buffer (so synchronization between CPU and DMA was necessary via ioctls)
- The processing library may simply use the pointer to the data
  - But what about cyclic buffer?

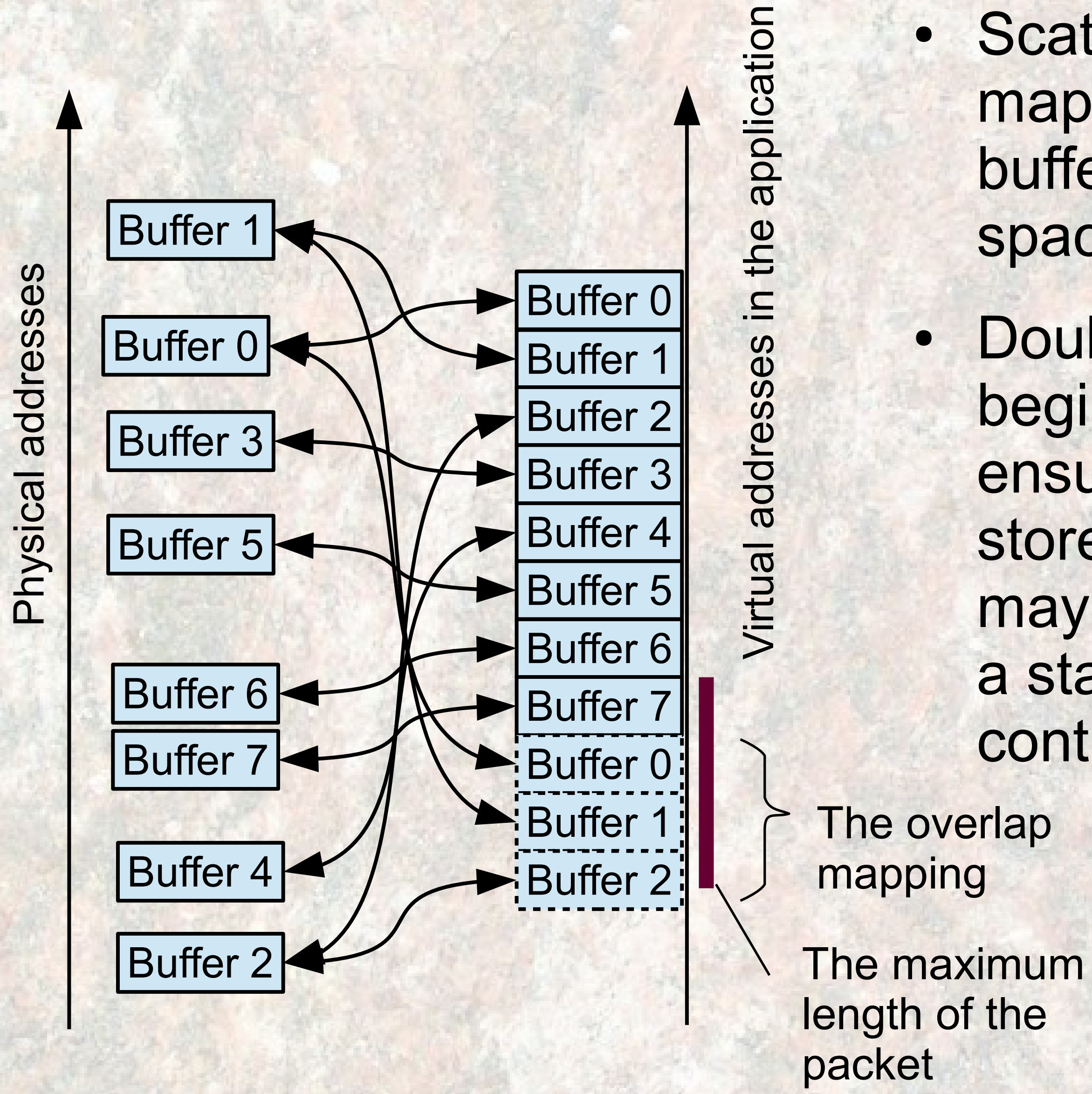
# Buffer mapping

- Received data are organized in structures for direct access from the C-language
- The scattered DMA buffers were mapped so that they create a huge continuous buffer in a virtual address space
- To allow efficient direct processing – caching was switched on for the buffer (so synchronization between CPU and DMA was necessary via ioctls)
- The processing library may simply use the pointer to the data
  - But what about cyclic buffer?
  - The first solution is usage of the buffer with length of 2^N bytes, and using the modular arithmetic to access the contents

# Buffer mapping

- Received data are organized in structures for direct access from the C-language
- The scattered DMA buffers were mapped so that they create a huge continuous buffer in a virtual address space
- To allow efficient direct processing – caching was switched on for the buffer (so synchronization between CPU and DMA was necessary via ioctls)
- The processing library may simply use the pointer to the data
  - But what about cyclic buffer?
  - The solution is the „overlap mapping”

# Overlap mapping



- Scattered DMA buffers are mapped as a continuous buffer in the virtual address space.
- Double mapping of the beginning of the buffer ensures, that each object stored in the cyclic buffer may be reliably accessed via a standard pointer as a continuous entity.

## Results

- The third system was tested with the simulated data.
- The achieved throughput was 14.2 Gb/s (89% of the theoretical throughput 16 Gb/s for 4 lanes PCIe Gen 2.
- Long term (28 h) tests has proven the error-free transmission.

# Conclusions

- Three DMA systems adjusted to different architectures of the data acquisition systems and different requirements are presented.
- The simplest version performs DMA transfers on request from the data-processing application.
  - no problems related to cyclic mode, possible overruns, and synchronization between the DMA and the processing threads.
- The third version is the high-performance system able to almost fully utilize the bandwidth of the PCIe bus for delivery of the continuous stream of data for a long time.
- The possibilities to work around deficiencies of the IP-core design have been presented.
- All presented DMA systems have been successfully synthesized, implemented and tested. They may be reused in different DAQ systems - both based on SoC chips using only the AXI bus, and in PCIe-based systems with the PCIe endpoint blocks.
- The presented solutions are based on Xilinx provided IP cores. However, similar blocks are available also for FPGA or SoC chips from other vendors. The described techniques used in the Linux kernel drivers should also be portable to other hardware platforms.

Thank you for your attention!