DMA implementations for FPGA-based data acquisition systems

Wojciech M. Zabołotny[a]

[a]Institute of Electronic Systems, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland

ABSTRACT

The paper describes three implementations of DMA engines created for different data acquisition (DAQ) systems. The designs are based on vendor (Xilinx) provided IP cores. The emphasis is put on typical problems related to the implementation of high-performance data acquisition systems. The selection or building of proper DMA cores is shown, and typical problems associated with the realization of the device driver are described. Sources of the described systems are publically available.

Keywords: FPGA, DMA, Xilinx, high-performance, data acquisition systems, DAQ, Linux, device drivers

1. INTRODUCTION

To achieve flexible interfacing to different Front-End Electronics (FEE) systems, Field-Programmable Array (FPGA) chips are often used in the input stages of the DAQ. However, if the acquired data must be submitted to any complex processing, a computer system is needed as the next stage of the DAQ. In fact, there is a tendency in modern data acquisition systems to involve computers as near to the input as possible, to minimize the usage of high-cost specialized components. Good examples of this approach may be the proposed upgrades of DAQ in LHCb[1] and ATLAS[2] detectors in the LHC experiment at CERN, and last proposals for the readout chain in the CBM experiment.[3] This paper assumes that the DAQ computer system runs under control of the Linux OS.

To preserve the computational power of the CPU for processing of data, it is desirable that the data be delivered from the FPGA logic to the memory of the computer using the Direct Memory Access (DMA).

Usually, the DMA system must be adjusted to the specific needs of the created data acquisition system. With FPGAs, there is a huge number of solutions to choose from. There are many open source solutions for different FPGA families and for different busses. Just a few examples may be found in [4–7]. The FPGA vendors also provide DMA IP cores for their chips. They are usually well matched to the particular FPGA family, well integrated with the vendor's synthesis tools, but may be not portable to other FPGAs. This paper presents three DMA engine solutions based on Xilinx provided IP cores, to show how the specific features of the DAQ influenced the chosen DMA architecture both in the firmware and in the software layers, and to show typical problems and their solutions.

2. THE DMA SYSTEM FOR PCIE CONNECTED FPGA WITH EXTERNAL DDR MEMORY

In the data acquisition system,[8] the acquired data were transferred by the user logic to the external DDR memory connected to the AXI DDR Memory controller. The FPGA and the memory were located on the PCIe board. The DMA core was responsible for the transfer from the FPGA-connected DDR memory to the memory in the PC computer. This mode of operation may be well suited for situations, where the data stream exhibits huge fluctuations, and the DDR memory may be used to temporarily buffer the huge amount of data, which later on may be transferred to the PC via DMA. The block diagram of the DAQ architecture is shown in Fig. 1

2.1 The firmware

The first tests of the DMA system were performed in the KC705 board[9] and the final tests on a dedicated board equipped with Artix-7 XC7A200T FPGA. As the DMA core, the AXI Central DMA controller[10] was used. As the PCI endpoint, the AXI Memory Mapped to PCI Express Gen2 (AXI MM PCIe) block[11] was used. The preparation of the firmware part was easy and required only the connecting and configuration of IP cores in the Vivado[12] Block Design Editor.

Further author information: (Send correspondence to W.M.Z.)
W.M.Z.: E-mail: wzab@ise.pw.edu.pl, Telephone: +48 22 234 7717

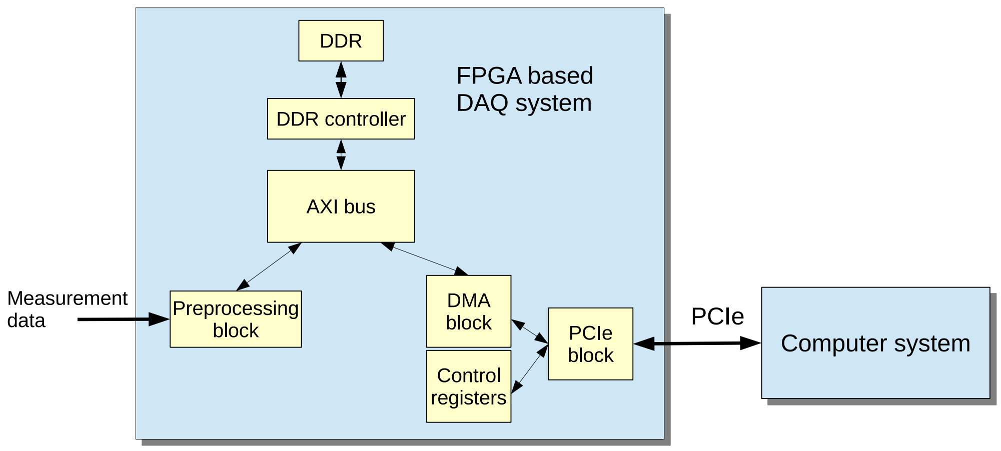

Figure 1: The architecture of the DAQ system with data buffered with shared DDR memory.

2.2 The software

The DMA core was controlled by the dedicated Linux device driver. The DMA buffer was allocated during the initialization of the driver and was later mapped into the memory of the user-space application. That provided direct access to the transferred data. The DMA did not have to work continuously. Consecutive chunks of data were transferred and processed by the user space application. The sources of the design are available on the Xilinx forum.[13]

3. DMA SYSTEM FOR AXI4-STREAM BASED DAQ

The second DMA system was created for the acquisition of data from the video encoder.[14] In this case, both the data source and the computer system were located in the same System on Chip (SoC) on the Xilinx ZCU-102 board.[15] The architecture of the system is shown in Fig. 2

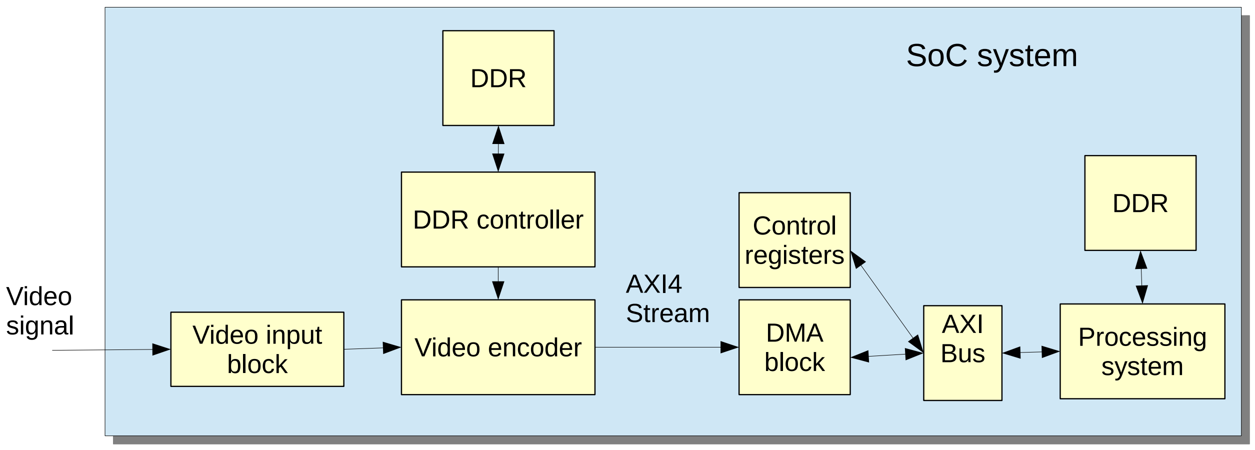

Figure 2: The architecture of the AXI4-Stream based DAQ.

3.1 The firmware

In that system the video data was delivered by the AXI4-Stream interface. Each frame was delivered in a separate AXI4-Stream packet. Because the data was compressed, the frames had different lengths. The first idea was to use the AXI DMA controller.[16] It allows DMA transfer from the AXI4-Stream source to the memory-mapped destination. It offers the scatter-gather (SG) support, that is important for big DMA buffers. It also supports the cyclic DMA mode, that is essential for the continuous data acquisition needed for smooth and stable video streaming. Reception of variable-length packets required to prepare buffers with the size equal to the maximum possible packet length. The transfer to the particular buffer was completed when the TLAST signal was asserted in the AXI4-Stream interface. Unfortunately it appeared that the standard Linux driver does not correctly report the real length of the transferred packet.[17] The analysis of the problem has shown, that the problem may be difficult to solve because the register with the real length of the transfer is quickly overwritten with the length of the next transfer. Therefore, another architecture was considered.

The second solution utilized the AXI Data Mover[18] block, which offers similar functionality like the AXI DMA Controller, but is controlled by the commands transmitted by the "Command" AXI4-Stream interface, and reports the statuses of completed transfers via the "Status" AXI4-Stream interface. To interface those AXI4-Stream interfaces with the CPU AXI bus, the AXI Streaming FIFO[19] IP core is used. However, the length of the commands used by the AXI Data Mover is 72 bits, while the AXI Streaming FIFO uses 32-bit long words. Therefore an additional AXI4-Stream Interconnect block was necessary to connect those components. The block diagram of the whole DMA engine is shown in Fig.3.

3.2 The software

The dedicated Linux device driver creates the DMA buffers for the configurable number of video frames. For each buffer, the transfer command is prepared, and all transfer commands are stored in the array. When the user-space application opens the device and starts the data acquisition, the transfer commands are written to the "commands" FIFO, until the FIFO is full or all commands are written. After the buffer is transferred, the corresponding status packet is written to the "status" FIFO, and the FIFO generates an interrupt. The driver reads the status packets and updates the information about the data available in the corresponding data buffer. The user-space application usually sleeps waiting for the new data and is woken up after the new buffer is filled. When the data processing is finished, the application confirms that and the corresponding transfer command is written again to the "command" FIFO, or put at the end of the list of the commands waiting for the place in the "command" FIFO. The described mechanism warrants that the "overrun" error (filling the buffer with new data before the previous content was processed) will never occur. The DMA buffers are mapped into the application’s memory allowing zero-copy access to the data. The communication between the user-space application and 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 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.

The sources of both versions of the described DMA systems in a reduced 32-bit version for Zynq 7000 SoCs are publically available on [20] in directories "axi_dma_prj1" and "axi_dma_prj2" as VEXTPROJ[21] compatible projects.

4. DMA SYSTEM WORKING AS A PCIE BRIDGE FOR AXI4-STREAM SOURCE

The third DMA system was created, when the system described in section 2 was modified for continuous acquisition of data and providing the real-time feedback.[22] In the continuous acquisition mode, if the DDR memory should be used for data buffering, the memory bandwidth must be equal to the maximum input data bandwidth (to accommodate input rate fluctuations) plus the output data bandwidth. In the Artix-7 based system it was not possible to fulfill those requirements, and therefore another data flow had to be used. The measurement data are not stored into the DDR memory, but are delivered via the AXI4-Stream interface directly to the DMA engine and further via PCIe bus to the memory of the

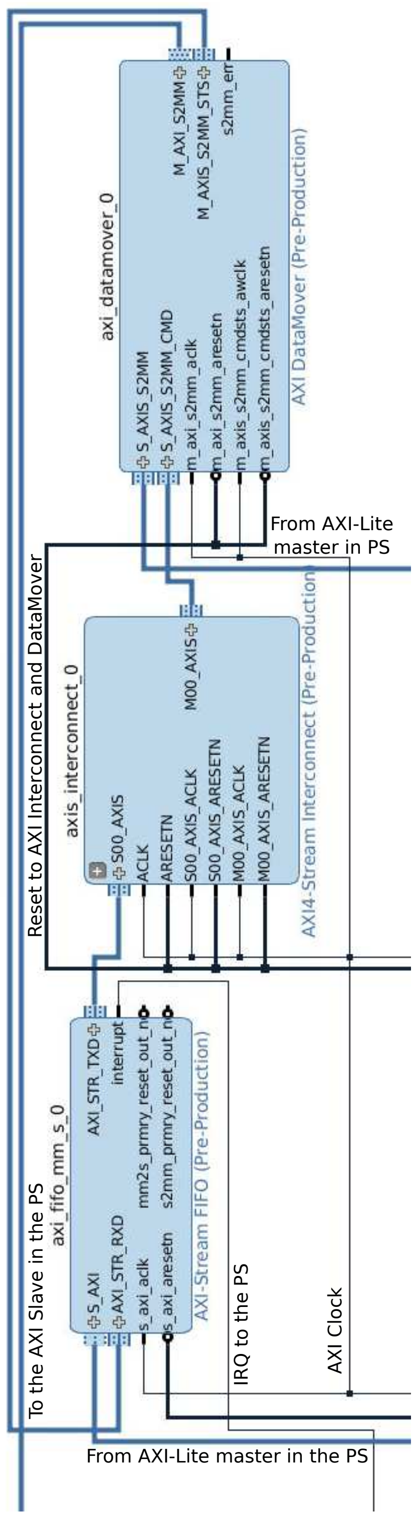

Figure 3: The block diagram of the DMA engine based on AXI Data Mover and AXI Streaming FIFO. The AXI4-Stream Interconnect IP core is used to convert a sequence of 32-bit words from the FIFO into 72-bit commands used by the AXI Data Mover.

computer. The input rate fluctuations must be handled by the internal block RAM resources, used by the AXI FIFO. Not implementing the DDR controller (in Artix 7 chip it was implemented in programmable logic) frees resources needed for preprocessing of data. The block diagram of the DMA system is shown in Fig. 4.

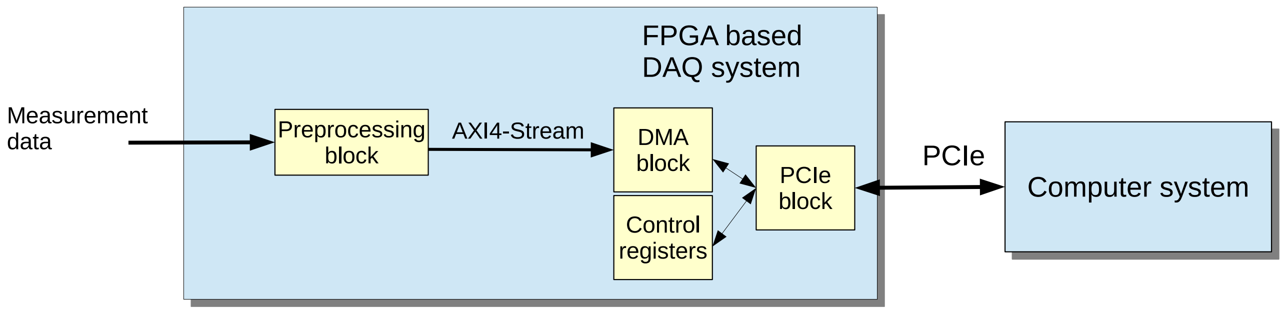

Figure 4: The architecture of the DAQ system with the DMA working as a PCIe bridge for the AXI4-Stream source.

4.1 The firmware

The first attempts to create the DMA core were based on the system described in section 3. The only difference was the addition of the AXI Memory Mapped to PCI Express Gen2 (AXI MM PCIe) block[11] as the bridge between the AXI and the PCIe busses. Unfortunately, it appeared that the AXI MM PCIe block does not support 64-bit addressing in its AXI Slave bridge, which was necessary to support data buffers with capacity above 4 GB in the computer memory.

The next proposed DMA system was based on the new DMA for PCI Express Subsystem (DMA for PCIe),[23] introduced first in the Vivado 2016.4. That IP core supports access to the whole 64-bit address space at the PCIe side. It also allows direct connection of the AXI4-Stream interface to the DMA engine simplifying the design. The use of the IP core itself in the firmware design was very simple and required only configuration of the core via GUI in the Vivado Block Designer. However, implementation of the software layer was much more complex.

4.2 The software

The DMA for PCIe (XDMA) core is so complex that it was necessary to use the xdma driver provided by Xilinx.[24] However, it appeared, that the driver and accompanying software are mainly oriented on the demonstration of the functionalities of the core, not on the continuous high-speed data acquisition. For example, the original driver supported the cyclical transfer only via *read* and *write* functions of the corresponding character device, which made continuous data acquisition impossible. Therefore, the modified version of the driver has been created in [25], in the directory "v2_xdma/software". The implemented mechanisms for the communication with the user-space application is similar like in the system described in section 3. The *ioctl* function is used to control the data acquisition, to wait for availability of the data and to confirm processing of the data. However, there also some significant differences.

The DMA for PCIe core supports the scatter-gather mode via the list of transfer descriptors located in the memory. If the list is circularly linked, it is possible to implement the continuous data transfer. Unfortunately, the DMA for PCIe used in so implemented continuous mode does not implement any protection against the “overrun” error. If the application does not process the delivered data quickly enough, the data will be silently overwritten by the next transfer using the same descriptor. It is even possible that data are overwritten during the processing, which may result in an attempt to process corrupted data, and cause errors in the user-space application. That problem was worked around using the mechanism of “metadata writeback” used by the DMA for PCIe core to report the status of the transfer. It is possible to set the writeback address in the descriptor to the address of the descriptor itself.[26] In such situation, the descriptor can not be reused until it is rewritten with its original content. Otherwise, the core detects the incorrect descriptor’s header and stops the transfer.

In the DMA system described in section 3 it was possible to use buffers bigger than the maximum expected size of the video frame. In that DMA system, the maximum length of packets with the detector data may be very huge, and their size may vary significantly. Even for the maximum reasonable size of a single DMA buffer of 4 MB, it is required, that a single packet may span across a few consecutive buffers. To handle that, the description of data in circular buffer had to be changed. Now the position of the packet is described with:

1. Number of the first buffer
2. Number of the last buffer
3. Number of words in the last buffer

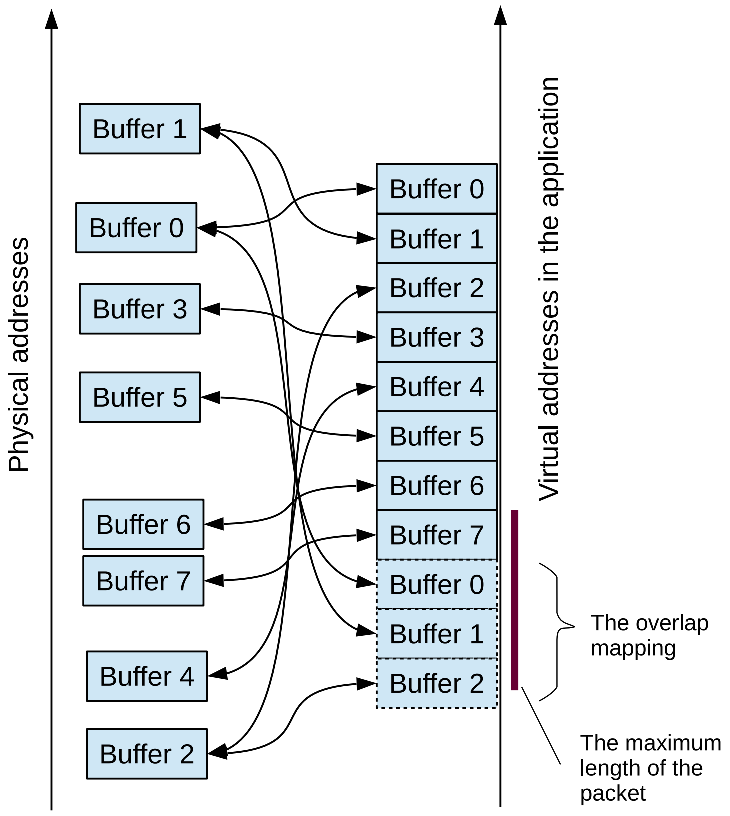

Figure 5: The mapping of the DMA buffers with the "overlap" area ensuring, that each packet may be accessed as a continuous object in the memory.

That change significantly affected the mapping of the memory. It is not possible to map each buffer independently. To allow easy processing of data, each packet spread across a few consecutive buffers should be visible as a continuous object in the virtual memory of the application. Therefore it was necessary to perform the mapping using the *vm_fault* handler. The system is prepared for working with huge buffers. A typical setup uses 1024 or even 2048 of 4 MB buffers (4 GB or 8 GB of memory). As the buffers are accessed only by the DMA and by the user-space application, it is not necessary to create the kernel-space mapping for buffers. That also can be achieved by mapping in the *vm_fault* handler. There is still one situation when the packet may be not continuous. That happens at the end of the buffer. There are two possible solutions. If the length of the buffer is equal to the power of two, the modular arithmetics can be performed on the word indices to transparently handle the wrapping of the object. However, if the processing of the data involves library functions, that require the processed data to be strictly continuous in the memory, it is possible to create an "overlap mapping". If the number of buffers required to store the biggest packet is equal to N, then $N-1$ initial buffers must be additionally mapped at the end of the virtual area (see Fig. 5).

The sources of the system are available in [25] as a VEXTPROJ[21] compatible project.

5. RESULTS

The CDMA based DMA system described in section 2 was tested with the simulated data. The achieved throughput was equal to 10.45 Gb/s for transfer from the FPGA connected DDR to the PC memory, and 8.05 Gb/s for transfer from the PC memory to the FPGA connected DDR.

The DMA system described in section 3 was tested both with simulated data and with the real encoder. Tests with the simulated data have shown that even with packets slightly below 4 MB long at 60 packets/s rate, the CPU usage was negligible (below 1%).

The XDMA based DMA system described in section 4 was tested with the simulated data, and the achieved throughput via 4 lanes PCIe Gen 2 interface was equal to 14.2 Gb/s (ca. 89% of the theoretical throughput).

All DMA systems provided reliable, error-free data transmission during the long-time tests.

6. CONCLUSIONS

The presented DMA systems are adjusted to different architectures of the data acquisition systems and different requirements. The simplest version performs DMA transfers on request from the data-processing application. Therefore there are no problems related to cyclic mode, possible overruns, and synchronization between the DMA and the processing threads. The third solution is the high-performance system able to almost fully utilize the bandwidth of the PCIe bus delivering the continuous stream of data for a long time. It also demonstrates possibilities to work around deficiencies of the IP core design. 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.

ACKNOWLEDGMENTS

This scientific work was partially supported by Polish Ministry of Science and Higher Education within the framework of the scientific financial resources in the year 2016 allocated for the realization of the international co-financed project - agreement no. 332380/PnH/2016; partially supported by National Centre for Research and Development of Poland - grant DOB-BIO7/01/02/2015; and partially supported by the statutory funds of Institute of Electronic Systems.

REFERENCES

[1] Cachemiche, J., Duval, P., Hachon, F., Gac, R. L., and Réthoré, F., "The PCIe-based readout system for the LHCb experiment," *Journal of Instrumentation* **11**(02), P02013 (2016).

[2] Anderson, J., Bauer, K., Borga, A., Boterenbrood, H., Chen, H., Chen, K., Drake, G., Dönszelmann, M., Francis, D., Guest, D., Gorini, B., Joos, M., Lanni, F., Miotto, G. L., Levinson, L., Narevicius, J., Vazquez, W. P., Roich, A., Ryu, S., Schreuder, F., Schumacher, J., Vandelli, W., Vermeulen, J., Whiteson, D., Wu, W., and Zhang, J., "FELIX: a PCIe based high-throughput approach for interfacing front-end and trigger electronics in the ATLAS upgrade framework," *Journal of Instrumentation* **11**(12), C12023 (2016).

[3] Zabołotny, W. M., Kasprowicz, G. H., Byszuk, A. P., Emschermann, D., Gumiński, M., Poźniak, K. T., and Romaniuk, R., "Selection of hardware platform for CBM Common Readout Interface." this volume.

[4] de la Chevallerie, D., Korinth, J., and Koch, A., "ffLink: A lightweight high-performance open-source PCI Express Gen3 interface for reconfigurable accelerators," *SIGARCH Comput. Archit. News* **43**, 34–39 (Apr. 2016).

[5] Usselmann, R., "WISHBONE DMA/Bridge IP Core." https://opencores.org/project,wb_dma .

[6] Hoc, E., "AXI DMA 32 / 64 bits." https://opencores.org/project,dma_axi .

[7] Borga, A., Schreuder, F., and Kharraz, O., "PCIe Gen3x8 DMA for virtex7." https://opencores.org/project,virtex7_pcie_dma .

[8] Wojenski, A., Pozniak, K. T., Kasprowicz, G., Kolasinski, P., Krawczyk, R., Zabolotny, W., Chernyshova, M., Czarski, T., and Malinowski, K., "FPGA-based GEM detector signal acquisition for SXR spectroscopy system," *Journal of Instrumentation* **11**(11), C11035 (2016).

[9] Xilinx, "Xilinx Kintex-7 FPGA KC705 Evaluation Kit." https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html .

[10] Xilinx, "AXI Central DMA Controller." https://www.xilinx.com/products/intellectual-property/axi_central_dma.html .

[11] Xilinx, "AXI Memory Mapped to PCI Express (PCIe) Gen2." https://www.xilinx.com/products/intellectual-property/axi_pcie.html .

[12] Xilinx, "Vivado Design Suite - HLx Editions." https://www.xilinx.com/products/design-tools/vivado.html .

[13] Zabołotny, W. M., "CDMA, AXI MM to PCIe and DDR in KC705." https://forums.xilinx.com/t5/PCI-Express/Central-DMA-and-AXI-MM-to-PCIe-freezes-after-DMA-transfer/m-p/683283#M7367 .

[14] Zabołotny, W. M., Sokół, G., Pastuszak, G., et al., "Implementation of multistandard video signals integrator." this volume.

[15] Xilinx, "Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit." https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-es2-g.html .

[16] Xilinx, "AXI DMA Controller." https://www.xilinx.com/products/intellectual-property/axi_dma.html .

[17] Zabołotny, W. M., "AXI DMA - checking the number of really transferred bytes for particular dma_async_tx_descriptor." https://forums.xilinx.com/t5/Embedded-Linux/AXI-DMA-checking-the-number-of-really-transferred-bytes-for/m-p/715662 .

[18] Xilinx, "AXI Datamover." https://www.xilinx.com/products/intellectual-property/axi_datamover.html .

[19] Xilinx, "AXI Streaming FIFO." https://www.xilinx.com/products/intellectual-property/axi_fifo.html .

[20] Zabołotny, W. M., "Z-turn examples." https://github.com/wzab/Z-turn-examples .

[21] Zabołotny, W. M., "Version control friendly project management system for FPGA designs," *Proc. SPIE* **10031**, 1003146–1003146–9 (2016).

[22] Krawczyk, R., Linczuk, P., Kolasiński, P., Wojeński, A., Kasprowicz, G. H., Poźniak, K., Romaniuk, R., Zabołotny, W., Zienkiewicz, P., Czarski, T., and Chernyshova, M., "The speedup analysis in GEM detector based acquisition system algorithms with CPU and PCIe cards," *Acta Physica Polonica B Proceedings Supplement* **9**(2), 257–262 (2016).

[23] Xilinx, "DMA for PCI Express (PCIe) Subsystem." https://www.xilinx.com/products/intellectual-property/pcie-dma.html .

[24] Xilinx, "AR# 65444 Xilinx PCI Express DMA Drivers and Software Guide ." https://www.xilinx.com/support/answers/65444.html .

[25] Zabołotny, W. M., "Artix-DMA1." https://gitlab.com/WZab/Artix-DMA1 .

[26] Zabołotny, W. M., "DMA/Bridge Subsystem for PCI Express v3.0 - usage in cyclic mode ." https://forums.xilinx.com/t5/PCI-Express/DMA-Bridge-Subsystem-for-PCI-Express-v3-0-usage-in-cyclic-mode/m-p/751088#M8456 .