ASIC Buffer and PCI Performance Testing

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachgebiet Integrated Electronic Systems
Date: Winter semester 01.02.2020

Contents

1 Introduction

This document explains a small system that uses DRAM as a data buffer for receiving high rate data from an ASIC. Continuous data at high speed will be written to memory and read by the computer through the PCI interface. The design must ensure that no data bytes are lost.

2 Hardware design in Vivado

2.1 Design overview

A very simple idea of the design is illustrated in figure 1. The system includes a data generator IP core, a DRAM interface, and a host PC. The data generator IP core is a simple counter and transfer the value of the counter register continuously to DRAM. This simulates the data stream from the ASIC and could be replaced by an ASIC or connected to ASIC later to transfer data to the host PC. At the host PC, a bash script uses an XDMA driver to read data from DRAM and synchronize the transfer process with the IP core in FPGA.

Figure 1: Design overview

2.2 Data transfer synchronization

A very important step is to establish a synchronization protocol between the host PC and the IP core. The gold is to ensure that data is not lost during transmission. There are some approaches are discussed as follow:

- **Polling address:** the host program will continuously poll the current write address of the IP and wait until the data is accumulated enough in the memory. As soon as the data is sufficient, the PCI starts reading to catch up with the IP.
- **Polling start signal:** almost the same as the first approach, but the IP manages the data accumulation itself. After writing a predefined amount of data, it turns on a signal. The host programm polls this signal to know when it should read data.
- **Interrupt read trigger:** the IP generates an interrupt signal to the host program after writing a predefined amount of data. This one is more complex and requires more time.
- **Predefined pattern:** the IP writes some specific patterns at the start and the end of a predefined amount of data . The host programm bases on these patterns to detect the valid values. By this way, the host can read data continuously and doesn't need to synchronise with the IP. The host can extract the valid data based on the pattern. The idea is illustrated in figure 2:

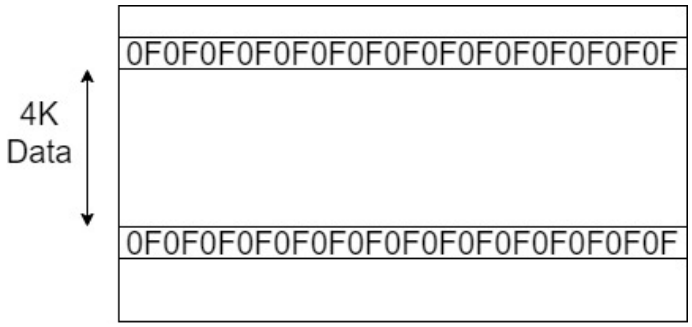

Figure 2: Predefined pattern

The approach is chosen as the first step for the design is Polling start signal. For more details on these approaches, please see the following document: `https://github.com/quangtran2796/asic_data_buffer/blob/main/document/Design_Idea.pdf`.

2.3 ASIC buffer system and bus interface

The main components of the block design and the bus interface between these components are described in the figure 3:

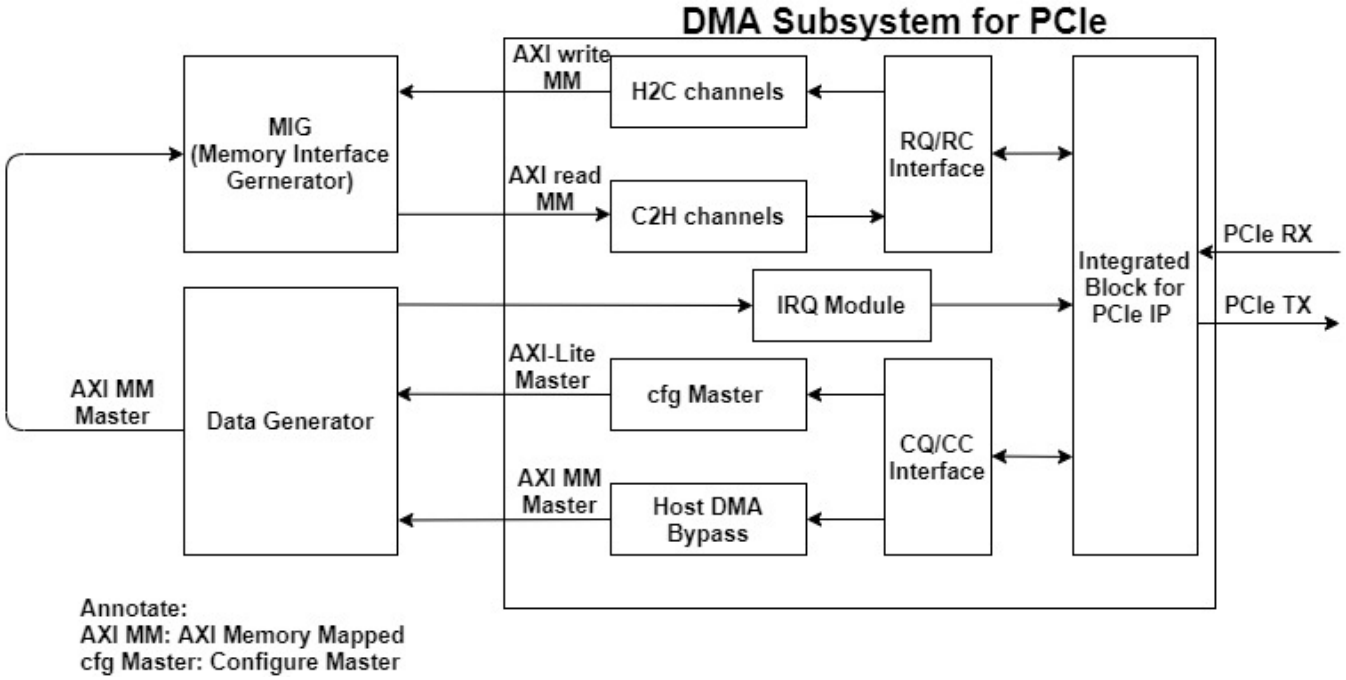

Figure 3: ASIC buffer system and bus interface

Basically, there are have 4 channels that share the same AXI-Full master to access the memory and the IP also has a AXI-Master interface to write data to the memory. Besides that, the host program can get access to our IP (to poll address or start signal) by the AXI-Lite master of the configured Master block or the host DMA bypass. There could also be an interrupt signal to synchronize between our IP core and the host program.

2.4 Data Generator IP Core

The IP core has an internal counter register with a configurable counting rate. In addition, there is one AXI4Master interface for writing to DRAM and one AXI4Lite for communicating with the host PC for synchronization purpose.

The important configurable registers of the IP are listed following:

```
// The trigger signal register for the
// host PC. After writing data to DRAM successfully,
// the IP set this register to 1.
// Host PC polls this register continuously and
// start to read DRAM when this register is equal to 1.
Reg#(Bit#(32)) pci_start <- mkReg(0);

// Host PC set this register to 1,
// When the PC is ready to get the data.
Reg#(Bit#(32)) ip_data_get_start <- mkReg(0);

// Configure the burst of the AXI interface.
Reg#(UInt#(8)) axi_burst_length <- mkReg(3);


// Internal Counter register.
Reg#(Bit#(32)) ip_data <- mkReg(0);

// Configure how much data should be written to DRAM
Reg#(Bit#(32)) ip_data_amount <- mkReg(1000);
```

```
20
   // Configure the counting rate base on frequency.
22 Reg#(Bit#(32)) ip_time_setup <- mkReg(1);

24 // Read only register to check the current writing address in DRAM.
   Reg#(Bit#(64)) ip_current_write_address <- mkReg(0);
26
   // Configure the size of the DRAM,
28 // When the data overflow the DRAM size,
   // the IP begin again to write data to the first adrres.
30 Reg#(Bit#(64)) ddr_size <- mkReg(400);

32 // Register to verify if the IP core works.
   // This register will return the double of the value written to it.
34 Reg#(Bit#(32)) ip_test_register <- mkReg(0);

36 // The FIFO to buffer data from the data stream (Can be later replaced by ASIC data stream buffer),
   // before to be transferred to DRAM.
38 FIFOF#(Bit#(32)) ip_data_buffer <- mkSizedFIFOF(100);
```

Listing 1: IP important registers

The address if the registers are as follow:

- ip_data_get_start: 0x0000.
- pci_start: 0x0004.
- ip_test_register: 0x0008;
- ddr_size: 0x0010.
- ip_data_amount: 0x0020.

The host PC can overwrite these registers to change the IP behavior.
The design is written in Bluespec. All setup, testbench and detail comments can be seen from the github repository:
`https://github.com/quangtran2796/asic_data_buffer`

2.5 Vivado block design

After importing the designed IP to Vivado, the final block design is as in figure:

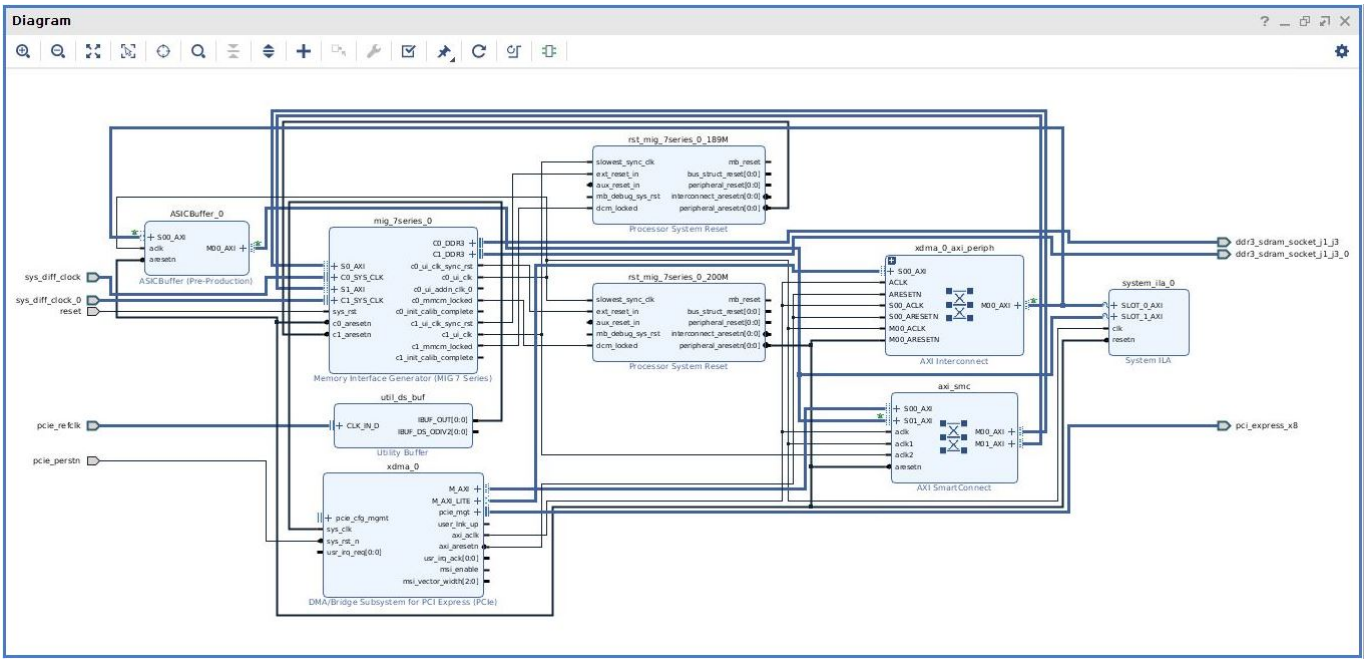

Figure 4: Vivado block design

The designed IP core can be used directly. The user only need to connect the AXI4Master and AXI4Slave interface to the Memory Interface Generator (MIG) and the PCI block correctly. All other configurations of the IP can be configure at run-time through the registers described in section 2.4.

The setting for PCI block can be seen from the following document:
`https://github.com/quangtran2796/asic_data_buffer/blob/main/document/PCI_XDMA_VC709.pdf`

3 PCI driver

3.1 Host PC PCI driver

The first requirement to use the PCI on the host PC is to install the XDMA driver from Xilinx. The install instructions can be seen in the following document:

After installing and testing the driver successfully. Add the following bash script to the directory `\dma_ip_drivers\XDMA\linux-kernel\tests`:

```
#!/bin/bash
#----------------------------------------------------------------------------
# Script variable
#----------------------------------------------------------------------------
tool_path=../tools


./load_driver.sh

# Configure the DRAM size.
$tool_path/reg_rw /dev/xdma0_user 0x0010 w 0xFFFFFFF0

# Read the new configured DRAM size value.
$tool_path/reg_rw /dev/xdma0_user 0x0010 w

# Configure the amount of data stream.
$tool_path/reg_rw /dev/xdma0_user 0x0020 w 0xFFFFFFF0

# Set register ip_data_get_start to begin writing data to DRAM.
$tool_path/reg_rw /dev/xdma0_user 0x0000 w 0x01

#$tool_path/reg_rw /dev/xdma0_user 0x0004 w

# Testing purpose register. This register will
# double the value written to it. The value can be
# read back to verify if the IP works.
$tool_path/reg_rw /dev/xdma0_user 0x0008 w 0x06

# Read value of testing register.
$tool_path/reg_rw /dev/xdma0_user 0x0008 w

# Read the amount of data stream back
$tool_path/reg_rw /dev/xdma0_user 0x0020 w

#----------------<polling register 4 to get pci start signal>----------------
$tool_path/reg_rw /dev/xdma0_user 0x0004 w
returnVal=$?

# Polling the pci_start register from the IP core.
# Perform a DRAM read when the register is equal to 1.
while [ $returnVal -eq 0 ]; do
    $tool_path/reg_rw /dev/xdma0_user 0x0004 w
    returnVal=$?
    echo "waiting for starting signal..."
done

# Read DRAM from PCI and save in ddr_data.bin file
$tool_path/dma_from_device -d /dev/xdma0_c2h_0 -f data/ddr_data.bin -s 4294967296 -a 0000 -c 1
returnBW=$?
echo $returnBW
#$tool_path/dma_from_device -d /dev/xdma0_c2h_0 -f data/ddr_data.bin -s 1024 -a 0000 -c 1

# Show the data on the terminal window.
xxd data/ddr_data.bin

rm data/ddr_data.bin
```

Listing 2: ASIC buffer bash script

The bash script describes an example of how to use the IP. In general, the registers of the IP should be configured first as desired and polling the pci_start signal afterward to read the data back from the DRAM.

More detailed information about the commands to use the Xilinx driver can be found in the following document: `https://github.com/quangtran2796/asic_data_buffer/blob/main/document/Xilinx_Answer_65444_2016_4.pdf`

3.2 PCI performance testing

The performance is displayed after executing each read-write command. The calculation of the performance can be seen in the directory: `dma_ip_drivers\XDMA\linux-kernel\tools`.

There are two approaches are tested in this design. On one hand, the 4GB data stream will be written to DRAM and after that, the PCI read this 4 GB back. During this time, the IP will buffer the data in the FIFO. After completing the PCI read process, the IP continues to write a new value to the DRAM. On the other hand, the data is divided into small data chunks. The IP and the host will alternately access DRAM to write and read these data chunks. The tested chunks are 4kB, 1MB, and 1GB.

For testing the performance of the second approach. The driver `\dma_ip_drivers\XDMA\linux-kernel\tools\dma_from_device.c` could be modified as follow:

```
if (verbose)
 fprintf(stdout,
   "dev %s, addr 0x%lx, aperture 0x%lx, size 0x%lx, offset 0x%lx, "
   "count %lu\n",
   device, address, aperture, size, offset, count);

float avBW = 0;
for(int j = 0; j < 4; j++){
   avBW += test_dma(device, address, aperture, 1073741824, offset, count, ofname);
   address += 1073741824;
}

avBW = avBW/4;
printf("Avarage BW 4KB = %f\n", avBW);

return 0;0
```

Listing 3: Modified dma driver

The full source code can be found in the github repository.

The results of the test run on the hardware show that the larger the data packet, the better the performance. The best case is to send 4GB of data one time. However, this required a larger buffer on the IP side to store the continuous data stream, while the host PC reads the 4GB data from DRAM. Depend on the real hardware and the data rate of the ASIC, the size of the data chunk could be configured to archive the best performance and save resources.

4 Aurara hardware replacement

When the ASIC is not available or changes the buffer configuration, the Aurora IP could be used to simulate the data stream of the ASIC. An idea of how to develop this system is as the following picture:

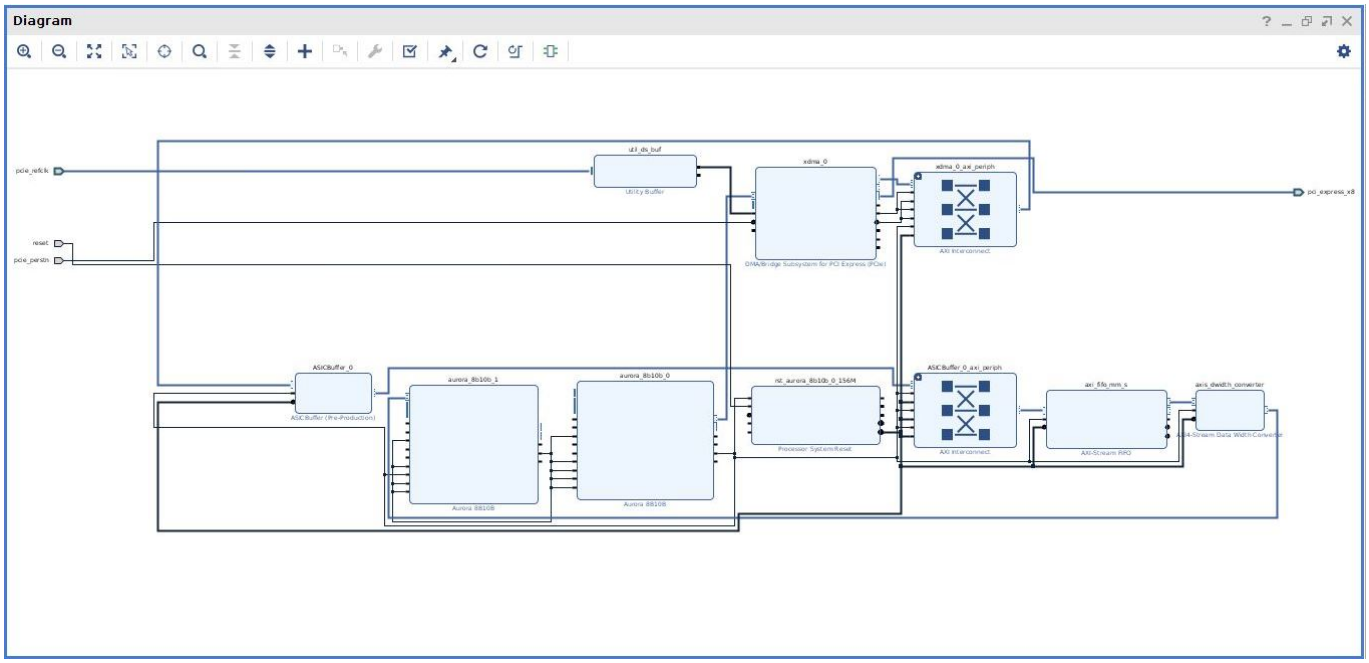

Figure 5: Aurora system

In this design, one Aurora core is used to transfer data generated from the IP core in section 2.4. Another Aurora core will receive the back data and transfer it as a data stream directly to PCI without involving DRAM. This can reduce the overhead from reading and write from DRAM.

However, the design cannot run successfully until now, because of the clock distribution problem and the limited time. Moreover, on the PCI side, if the data get in as a data stream, how can the driver of Xilinx manage the data and store them in the correct way? These are still open questions, however, These ideas can continue to be developed in the future.

5 Conclusion

The system can now work on basic functionalities, such as writing to DRAM, read data back over PCI, configure the IP core over PCI, synchronize IP Core and the host PC.

The transfer process archive better performance, when a large amount of data is sent at one time. For that reason, the chunk size should be as large as possible.

One large disadvantage is the process of accessing the DRAM takes a large latency. The idea of transferring data stream directly over PCI without DRAM would give a better performance. Part of this idea is done in the Aurora project and could be developed further in the future.