# PCIe improvement

Jaroslaw Szewinski

National Centre for Nuclear Research, Otwock-Świerk, Poland

Świerk, 2013

# Agenda

1. Introduction
2. Structure VHDL implementation
3. Software implementation

# Status

- Existing PCIe core written by Wojtek Jalmuzna needs upgrade, because the code structure is is not clear
- New PCIe interface is made to be as much as possible similar to the old one from the user point of view.
- New core seems to be ready, it need only testing and deployment in particular projects (whenever we will have to do it).
- All major projects still use old PCIe cores

# Netlists

- To improve the compilation time, Wojtek has provided the PCIe core as **netlist**. This is fine, except that there are 2 different PCIe netlist (one in SIS project, one in uTC project).

# Netlists

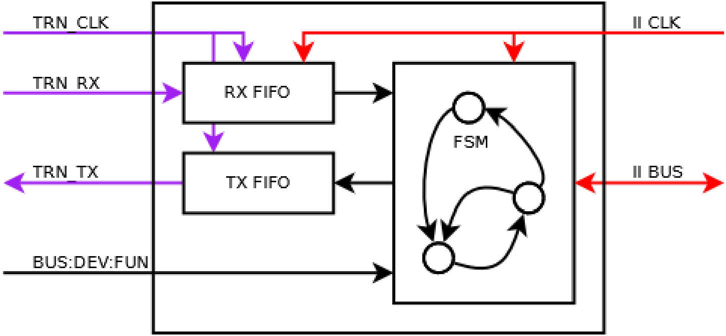
- To improve the compilation time, Wojtek has provided the PCIe core as **netlist**. This is fine, except that there are 2 different PCIe netlist (one in SIS project, one in uTC project).
- The problem: we have no (or I haven't found yet) source code for the SIS PCIe core netlist
- Adopting new core to the SIS project will be more difficult, and will take more time.

# Netlists

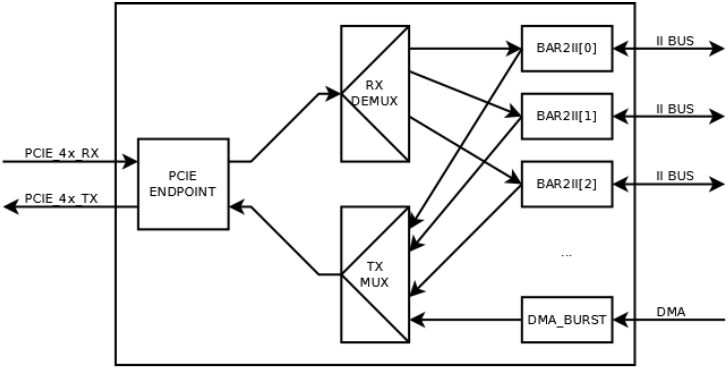
- To improve the compilation time, Wojtek has provided the PCIe core as **netlist**. This is fine, except that there are 2 different PCIe netlist (one in SIS project, one in uTC project).
- The problem: we have no (or I haven't found yet) source code for the SIS PCIe core netlist
- Adopting new core to the SIS project will be more difficult, and will take more time.

General remark: If you want to improve the project by using precompiled netlist - please, do it wisely and think twice.

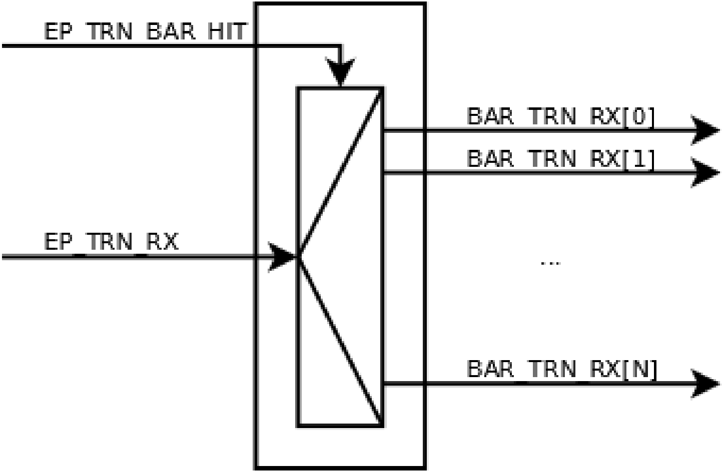
# PCIe BAR-to-II bus converter



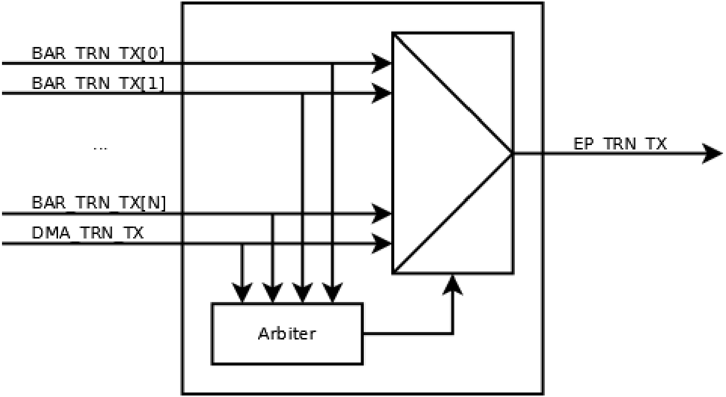
# PCIe Main Component



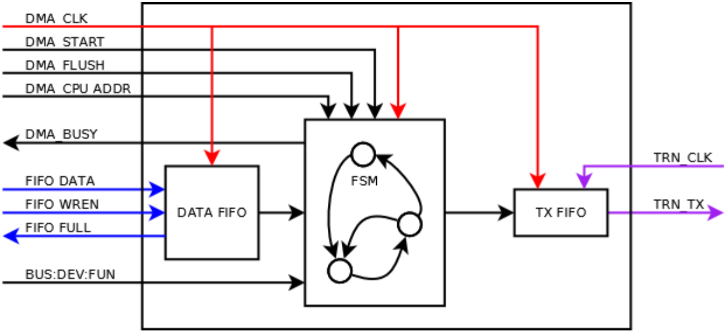
# PCIe Rx demultiplexer



# PCIe Tx multiplexer

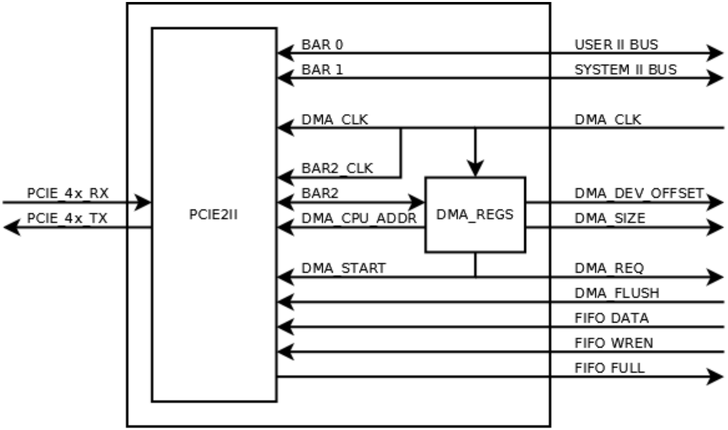


# DMA burst



DMA_CLK
DMA_START
DMA_FLUSH
DMA_CPU_ADDR
DMA_BUSY
FIFO DATA
FIFO WREN
FIFO FULL
BUS:DEV:FUN
DATA FIFO
FSM
TX FIFO
TRN_CLK
TRN_TX

# Standard PCIe Interface



# DMA transfer rules

- Max payload in one TLP (PCIe frame) is 32 DWORDs (32-bit values)
- New frame is transmitted whenever there is at least 32 values in the Data FIFO
- If last chunk of data is less than 32 values, the `dma_flush` forces sending frame with payload size equal the actual content of the Data FIFO
- Core has internal address counter of for address in CPU memory
- Each transaction increments CPU address by amount of sent data
- `dma_start` moves the CPU address counter back to provided CPU base address

# Using DMA (1)

When new pulse comes:

1. Generate positive pulse on `dma_start` to roll-back the CPU address pointer (one clock cycle is enough)
2. Push all data to Data FIFO using std. FIFO interface (data, wren, full)
3. After placing all data in Data FIFO, pulse `dma_flush`, to ensure that incomplete (<32 DWORDS) will be send.
4. If overall amount of data is integer multiplication of 32, `dma_flush` does not has to be touched.

# Using DMA (2)

When new pulse comes, and You are in hurry :-) :

1. Start placing data in Data FIFO immediately
2. Generate positive pulse on dma_start, not later when 32 DWORDs are placed in FIFO
3. After placing all data in Data FIFO, pulse dma_flush (if needed)

# Traditional DMA transfer

In the separate address space (BAR), there are 4 registers:

- `DMA_STATUS` - used to check if last interrupt was a “DMA finished” acknowledge
- `DMA_CPU_ADDR` - absolute destination[1] address in the CPU memory
- `DMA_DEV_ADDR` - relative source[1] offset in FPGA address space (i.e. internal or DDR2 memory)
- `DMA_SIZE` - amount of data to be transferred

[1] - We assume write-only to CPU DMA transfers only

# Traditional DMA transfer

In the separate address space (BAR), there are 4 registers:

- `DMA_STATUS` - used to check if last interrupt was a “DMA finished” acknowledge
- `DMA_CPU_ADDR` - absolute destination[1] address in the CPU memory
- `DMA_DEV_ADDR` - relative source[1] offset in FPGA address space (i.e. internal or DDR2 memory)
- `DMA_SIZE` - amount of data to be transferred

1 - We assume write-only to CPU DMA transfers only

Procedure:

1. User has to set from CPU both CPU and FPGA addresses
2. Transfer is triggered when `DMA_STATUS` is written
3. `DMA_STATUS` may be checked during next interrupt, before performing next access

# Traditional DMA transfer

In the separate address space (BAR), there are 4 registers:

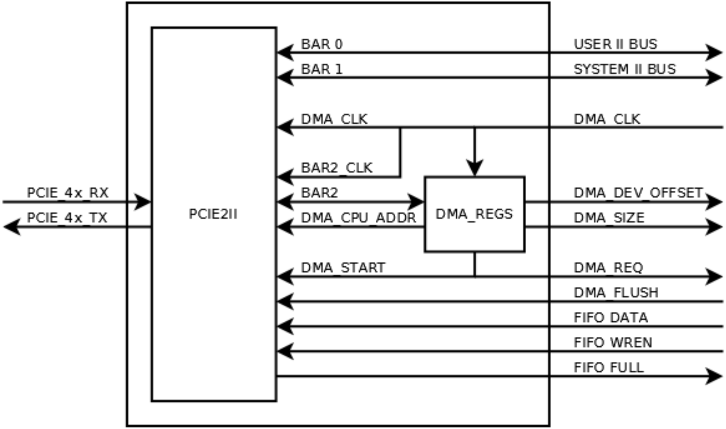
- `DMA_STATUS` - used to check if last interrupt was a "DMA finished" acknowledge
- `DMA_CPU_ADDR` - absolute destination[1] address in the CPU memory
- `DMA_DEV_ADDR` - relative source[1] offset in FPGA address space (i.e. internal or DDR2 memory)
- `DMA_SIZE` - amount of data to be transferred

[1] - We assume write-only to CPU DMA transfers only

Procedure:

1. User has to set from CPU both CPU and FPGA addresses
2. Transfer is triggered when `DMA_STATUS` is written
3. `DMA_STATUS` may be checked during next interrupt, before performing next access
4. Repeat for each $\sim$ 1MB data chunk (limitation of dynamic DMA mem. allocation)

# Standard PCIe Interface (again)



## DMA remarks

- DMA may be extremely dangerous, because due to achieve maximal performance it allows unrestricted access to raw physical memory (operating system pages, newly typed-in passwords, etc. - Please google for “DMA Attack”)
- Accidental DMA access may crash the OS, or make or other unpredicted behaviour.

# Traditional DMA transfer from Linux kernel point of view

- Allocate DMA capable (continuous physical) memory - (obtain CPU mem. addr)
- Set BAR2 registers (CPU_ADDR, FPGA_ADDR, MEM_SIZE)
- Stay idle, and wait for interrupt
- On interrupt, check DMA_STATUS, if the DMA is finished
- Release DMA buffer
- Repeat for each $\sim$ 1MB chunk

# New proposed approach for DMA transfers

Initial:

- Allocate DMA capable, continuous physical memory

# New proposed approach for DMA transfers

Initial:

- Allocate DMA capable, continuous physical memory during **boot-time**

# New proposed approach for DMA transfers

Initial:

- Allocate DMA capable, continuous physical memory during **boot-time**
- Tell the FPGA the base address of reserved physical memory
- On hardware interrupt (end of RF pulse), FPGA will start DMA writes by it self, will transfer all data, and generate interrupt

# New proposed approach for DMA transfers

Initial:

- Allocate DMA capable, continuous physical memory during **boot-time**
- Tell the FPGA the base address of reserved physical memory
- On hardware interrupt (end of RF pulse), FPGA will start DMA writes by it self, will transfer all data, and generate interrupt

User application and driver:

- Do a **Memory mapping** of reserved physical mem.
- Wait for interrupt
- Take a pointer, and read memory ! (your data is already there)

# Advantages of the new approach

- No 1MB CPU buffer limit
- No allocate/release of the memory
- Transfer automated on the FPGA side (no waiting for several requests)
- Software waits for 1 interrupt (no waiting for requests completions)
- Several applications can read the same reserved physical memory (shared memory, shared not only between processes on CPU, but even with the FPGA itself :-)
- The timing module and it's server does not have to be used at all - our driver can generate SIGUSR1 for us, on the interrupt generated by our board from the physical trigger signal (maybe more constant latency could be achieved).

# The end

Thank You