Ethernet transport protocols for FPGA

Wojciech M. Zabołotny

Institute of Electronic Systems, Warsaw University of Technology

Previous version available at:
https://indico.gsi.de/conferenceDisplay.py?confId=3080

FPGA and Ethernet is it a good combination?

- In 2011 in one of FPGA related blogs, there was an article published:

  "Designed to fail: Ethernet for FPGA-PC communication"
  http://billauer.co.il/blog/2011/11/gigabit-ethernet-fpga-xilinx-data-acquisition/
- In spite of this multiple efficient Ethernet based solutions for FPGA communication were proposed...

Why Ethernet at all?

- There are many other protocols to use with gigabit transceivers...
- There are some clear advantages:
  - Standard computer (PC, embedded system, etc.) may be the remote site
  - Long distance connection possible
  - Relatively cheap infrastructure (e.g. network adapters, switches)

Where's the problem?

- Ethernet is inherently unreliable
- Ethernet offers high throughput, but also high latency, especially if we consider the software related latency on the computer side of the link.
- To ensure reliability we must introduce acknowledge/retransmission system, buffering all unacknowledged data
- To achieve high throughput we need to work with multiple packets in flight, which makes management of acknowledge packets relatively complex...

Standard solutions...

- The standard solution for reliable transfer of data via packet network is TCP/IP
- Due to the fact, that it is suited for operation in public wide area networks, it contains many features not needed in our scenario, but seriously increasing the resources consumption.
- We don't need:
  - Protection against session hijacking
  - Solutions aimed on coexistence of different protocols in the same physical network

Alternative solution – simplified TCP/IP

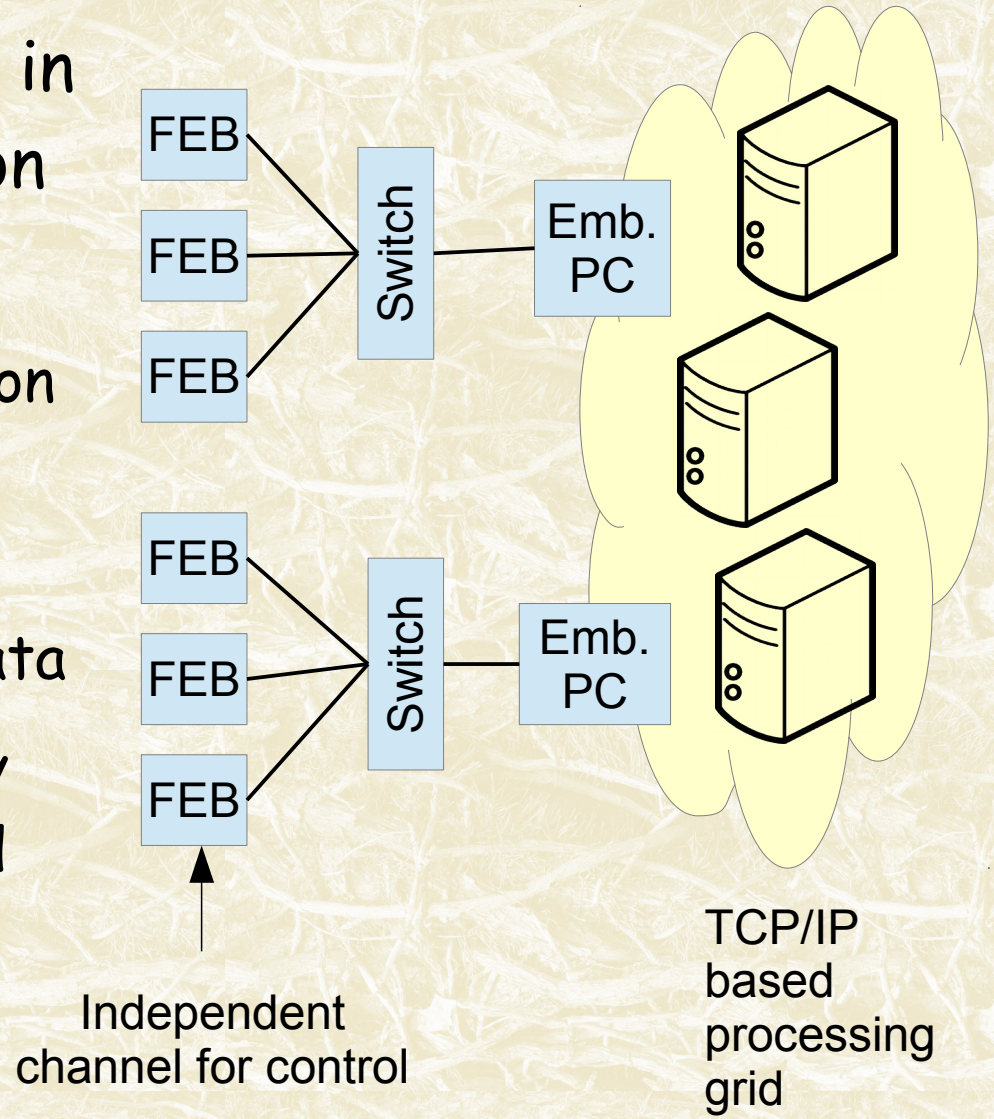
- Article: Gerry Bauer, Tomasz Bawej et.al.: 10 Gbps TCP/IP streams from the FPGA for High Energy Physics
  http://iopscience.iop.org/1742-6596/513/1/012042
- Seriously limited TCP/IP protocol. Unfortunately sources of the solution are not open, so it was not possible to investigate that solution...

Can we get things simpler?

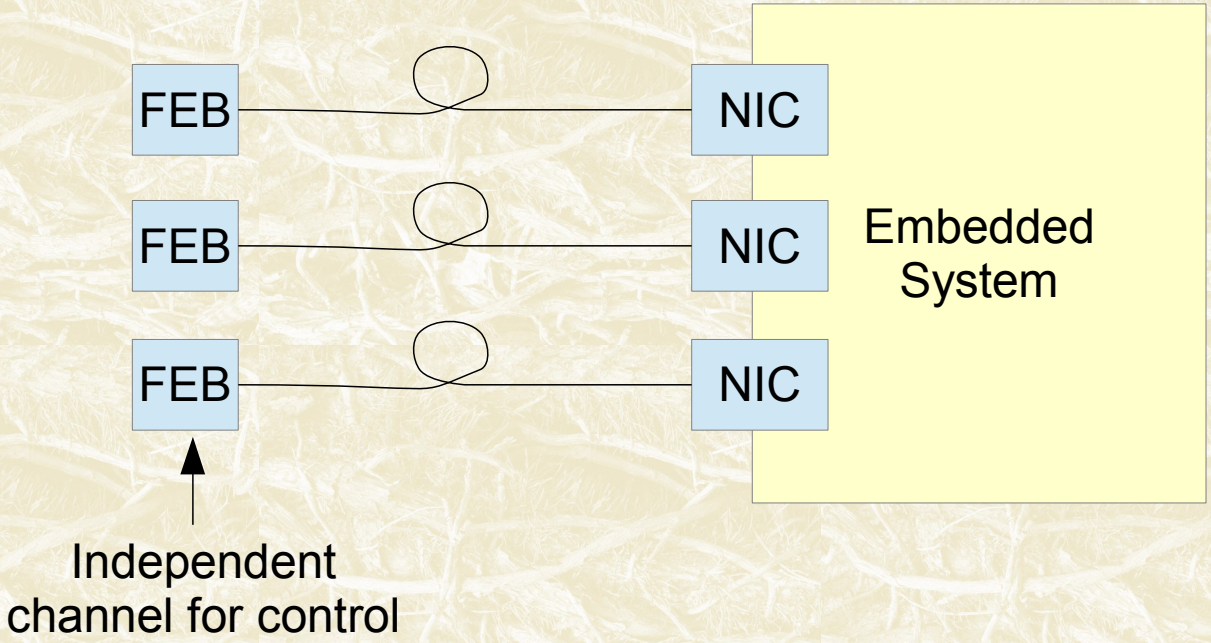
- Aim of the protocol
  - Ensure that data are transferred from FPGA to the computer (PC, embedded system...) with following requirements:
    - Reliable transfer
    - Minimal resources consumption in FPGA
    - Minimal CPU usage in the computer
    - Possibly low latency

Typical use case?

- Proposed solution may be used in price optimized data acquisition systems
  - Front End Boards (FEBs) based on small FPGAs
  - Standard network cabling and switches used to concentrate data
  - Standard embedded system may be used to concentrate data and send it further via standard network...



Simplified use case



- If we have separate NIC cards for each FEB, the task is even mor e simplified
- We have granted bandwidth for each connection (as long, as the CPU power in the embedded system is sufficient)
- We mainly need to transfer data, however simple control commands are welcome...

To keep the FPGA resources consumption low...

- We need to minimize the acknowledgement latency
- We need to keep the protocol as simple as possible

Main problem – acknowledge latency

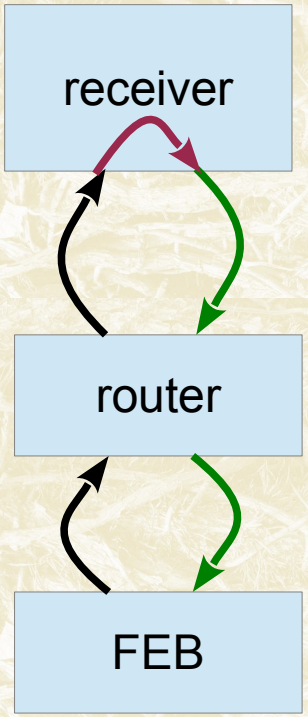
- If we need to reliably transfer data via unreliable link (e.g. Ethernet) we have to use the acknowledge/retransmit scheme
  - Other approach may be based on forward error correction, but it is less reliable
- If the transfer speed is equal to R_{transm}, and the maximum acknowledge latency is equal to t_{ack}, than the necessary memory buffer in the transmitter must be bigger than:

$$S_{buf} = R_{transm} t_{ack}$$

- If we are going to use a small FPGA with small internal memory, we need to minimize the latency

Standard protocols and latency

- To minimize latency, we must give up routing of packets. The first node should acknowledge reception of the packet.
- Routability, which is a big advantage of standard protocols is therefore useless for us.
- Standard protocols are handled by a complex networking stack in Linux kernel which leads to relatively high acknowledge latency
- Routing increases the latency even more...
- Use of protocols designed for routing (eg. IP) is not justified!



Finally chosen solution

- Use of own Layer 3 protocol
- Use of own protocol handler implemented as a Linux kernel driver
- Use of memory mapped buffer to communicate with data processing application
- Use of optimized Ethernet controller IP core in the FPGA communicating directly with Ethernet Phy

Alternative solution

- Article: B. Mindur and Ł. Jachymczyk: The Ethernet based protocol interface for compact data acquisition systems
  http://iopscience.iop.org/1748-0221/7/10/T10004
- Similar, but more complex solution (multiple streams with different priorities)
- Communication with user space via netlink
- Sources not available, so it was difficult to evaluate the solution...

Short description of the FADE-10g protocol

- Packets sent from PC to FPGA:
  - Data acknowledgements and commands
- Packets sent from FPGA to PC:
  - Data for DAQ
  - Command responses/acknowledgements (if possible, they are included in the data packets)

Structure of the packets

- Command packet (PC to FPGA)
  - TGT (6 bytes), SRC (6 bytes)
  - Protocol ID (0xfade), Protocol ver. (0x0100) – 4 bytes
  - Command code (2 bytes), Command sequence number (2 bytes)
  - Command argument (4 bytes)
  - Padding (to 78 bytes)

Structure of the packets

- ACK packet (PC to FPGA)
  - TGT (6 bytes), SRC (6 bytes)
  - Protocol ID (0xfade), Protocol ver. (0x0100) – 4 bytes
  - ACK code 0x0003 (2 bytes)
  - Packet repetition number (2 bytes)
  - Packet number in the data stream (4 bytes)
  - Transmission delay (4 bytes)
  - Padding (to 78 bytes)

Structure of the packets

- Command response (FPGA to PC)
  - TGT (6 bytes), SRC (6 bytes)
  - Protocol ID (0xfade), Protocol ver. (0x0100) – 4 bytes
  - Response ID (0xa55a) – 2 bytes
  - Filler (0x0000) – 2 bytes
  - CMD response – 12 bytes
    - Command code (2 bytes), Command seq number (2 bytes)
    - User defined response – 8 bytes
  - Padding to 78 bytes (may be shortened to 64 bytes)

Structure of the packets

- Data packet (FPGA to PC)
  - TGT (6 bytes), SRC (6 bytes)
  - Protocol ID (0xfade), Protocol ver. (0x0100) – 4 bytes
  - Data packet ID (0xa5a5) (2 bytes)
  - Embedded command response (12 bytes)
  - Packet repetition number (2 bytes)
  - Number of packet in the data stream (4 bytes)
  - Data (1024 8-byte words = 8KiB)

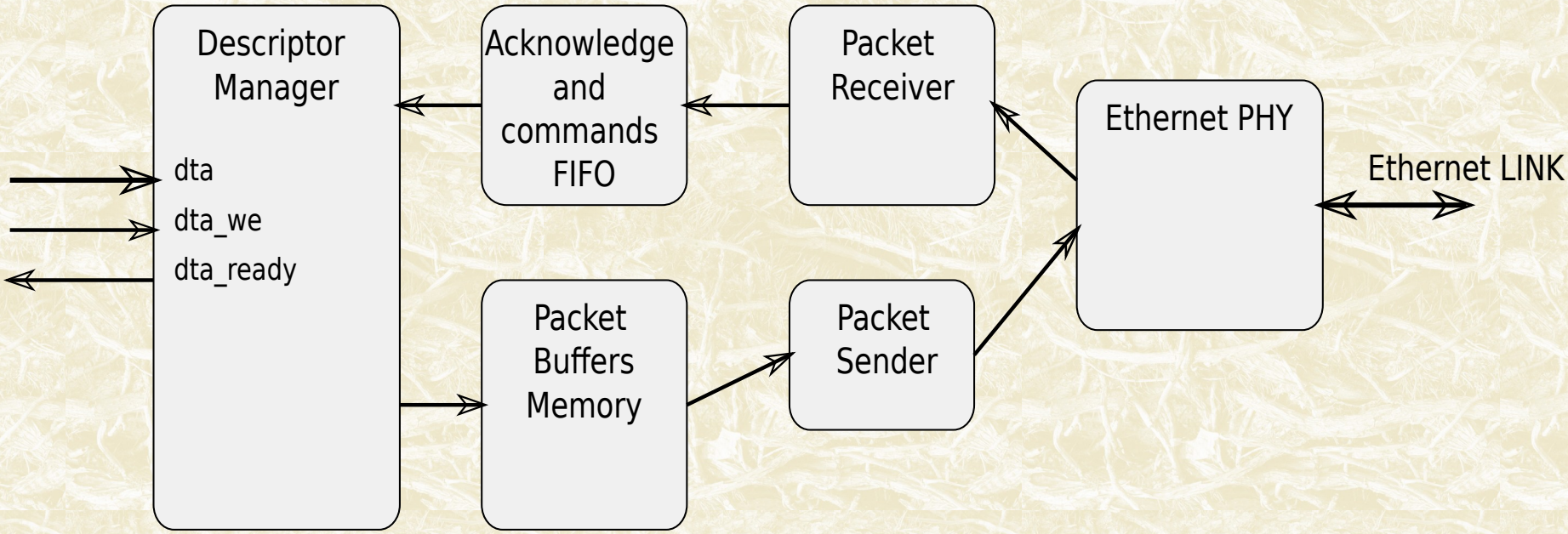
Structure of the packets

- Last data packet (FPGA to PC)
  - TGT (6 bytes), SRC (6 bytes)
  - Protocol ID (0xfade), Protocol ver. (0x0100) – 4 bytes
  - Data packet ID (0xa5a6) (2 bytes)
  - Embedded command response (12 bytes)
  - Packet repetition number (2 bytes)
  - Number of packet in the data stream (4 bytes)
  - Data (1024 8-byte words = 8KiB), but the last word encodes number of used data words (always less than 1024).

Meaning of „repetition numbers"

- To minimize latency, protocol tries to detect lost packets as soon as possible.
- We assume, that packets are delivered, and ACKs are transmitted in order.
- If FPGA receives ACK not for the last unacknowledged, but for one of next packets, it immediately knows, that the previous one(s) where lost must be retransmitted.
- However if the loss of the next packet will be also detected, it is necessary to retransmit only those unconfirmed packets, which have not been retransmitted yet.
- In case of multiple packets in flight, and multiple retransmissions, things get complicated
- The repetition numbers are a simple way to avoid unnecessary retransmissions of already retransmitted packets (we retransmit only the packets with "repetition number" lower than the one of received ACK packet).

Short description of FPGA core



Short description of protocol handler

- The packet should be serviced and acknowledged as soon as possible
- We want to use the network device driver (to assure wide hardware compatibility)
- The right method to install our handler is *dev_add_pack* so our packet handler is called as soon as packet is received
- The dedicated kernel module servicing the packets with Ethernet type 0xfade has been implemented

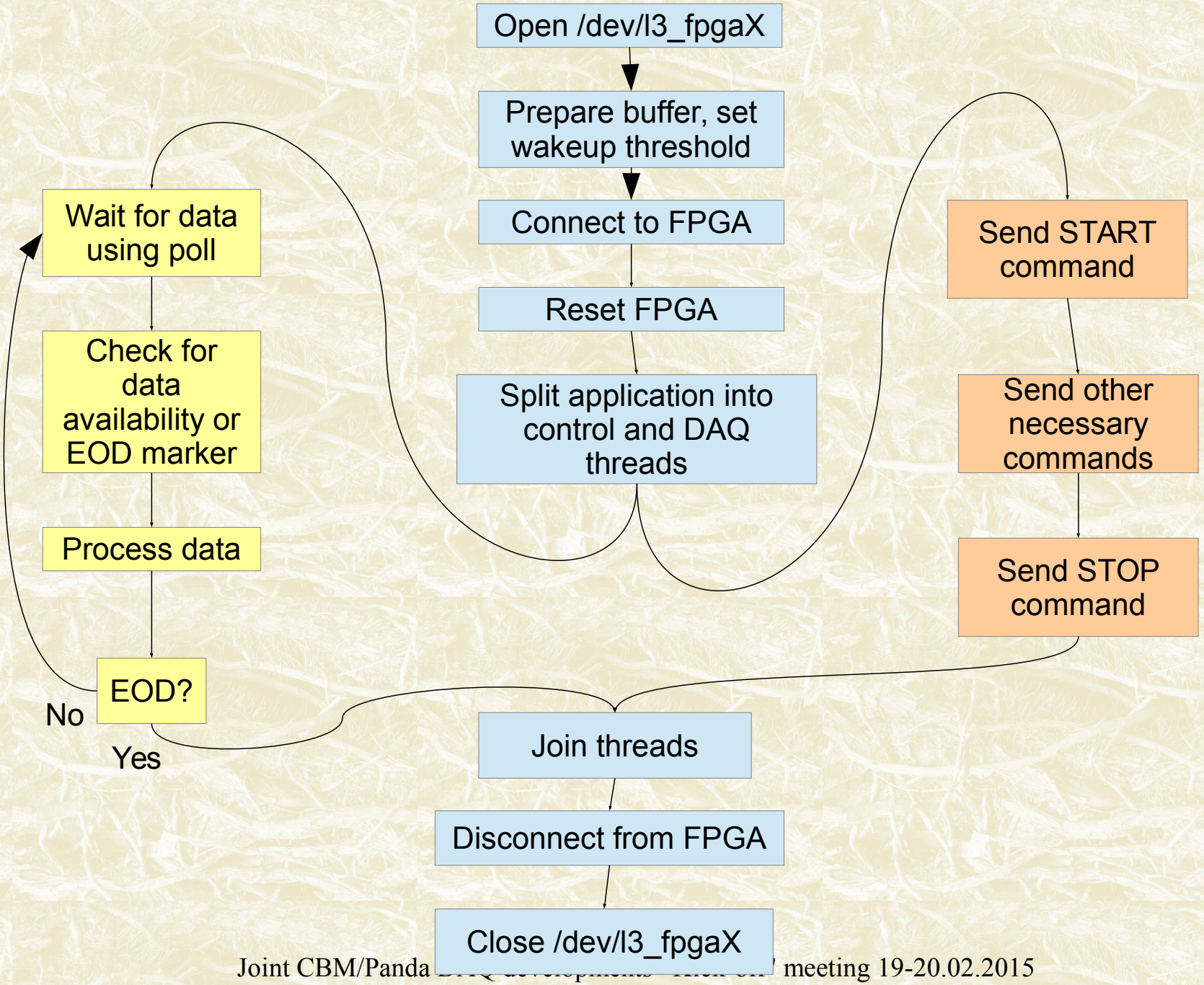
Transfer of data to the user space

- To avoid overhead associated with transferring data to the user space, the driver uses a kernel buffer mapped into the user space
- Data are copied directly from the *sk_buff* structure to that buffer, using the *skb_copy_bits* function.
- Synchronization of pointers between the kernel driver and the user space application is assured via *ioctl* command

Synchronization of the application with the data flow

- The application may declare the amount of data which should be available, when application is woken up
- The application uses the *poll* function to wait for data (or timeout).
- Control commands sleep, until the commands is executed and confirmed, therefore it is suggested to split application into threads

Suggested organization of an application



Synthesis results

- For KC705 with 32 packet buffers
  - LUT usage: 3.04%
  - BRAM usage: 16.5%
- For KC705 with 16 packet buffers
  - LUT usage: 3.02%
  - BRAM usage: 9.32%
- Internal logic working at 156,25MHz

System used for testing

- Intel(R) Core(TM) i5-4440 CPU @ 3.10GHz
  - (however during tests it operated at 800MHz-1GHz)
- Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection

  Intel Corporation Ethernet Server Adapter X520-2

Results

- Throughput: 9.80Gb/s
- CPU load: 29%-42% (each word checked) in a single core handling the reception
- Acknowledgement latency: ~3μs
- Comparison TCP/IP (iperf, PC-PC):
  - Throughput: 9.84Gb/s (TCP/IP used longer frames for MTU=9000, packets captured by Wireshark had even 26910 bytes!)
  - CPU load: 42% (just reception!)
  - Acknowledgement latency: ~8μs

Conclusion...

- Advantages of the solution:
  - Simple, open source solution, may be easily adjusted and extended (BSD/GPL license)
  - Small resources consumption in FPGA, low CPU load in receiving computer
  - Standard NIC may be used at the receiving side
- Disadvantages of the solution:
  - Small developer's base
  - Still not fully tested and mature

Availability of sources

- Current sources are available at:

  http://opencores.org/project,fade_ether_protocol

  - Please use the "experimental_jumbo_frames_version"
  - Projects for KC705 (10Gb/s), for AFCK (10Gb/s multiple links) and for Atlys (1Gb/s) are prepared

Thank you for your attention!