fairwaves

33c3 / 27 December 2016

XTRX SDR:
Chasing PCIe performance

Alexander Chemeris
CEO, Fairwaves, Inc.

Sergey Kostanbaev
Head of engineering, Fairwaves, Inc.

Background

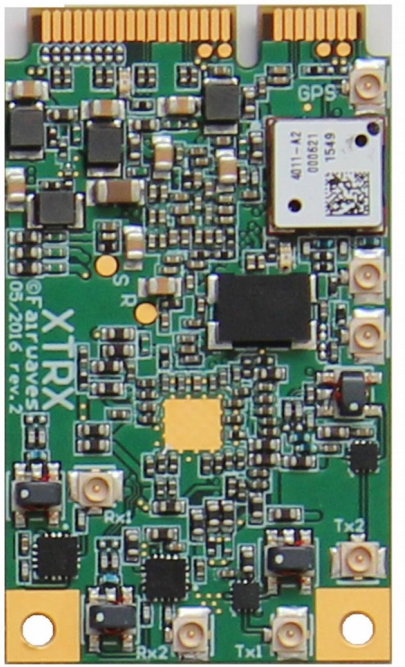
2008	Learnt about Software Defined Radio (SDR) and OpenBTS, got USRP1
2009	Developped ClockTamer to stabilize USRP1 clock to meet GSM requirements (presented at 26c3!)
2011	Started UmTRX development – SDR for GSM inspired by USRPs
2012	Transitioned from OpenBTS to Osmocom
2013	Deployed first GSM Base Stations based on UmTRX
2016	XTRX development

XTRX goal

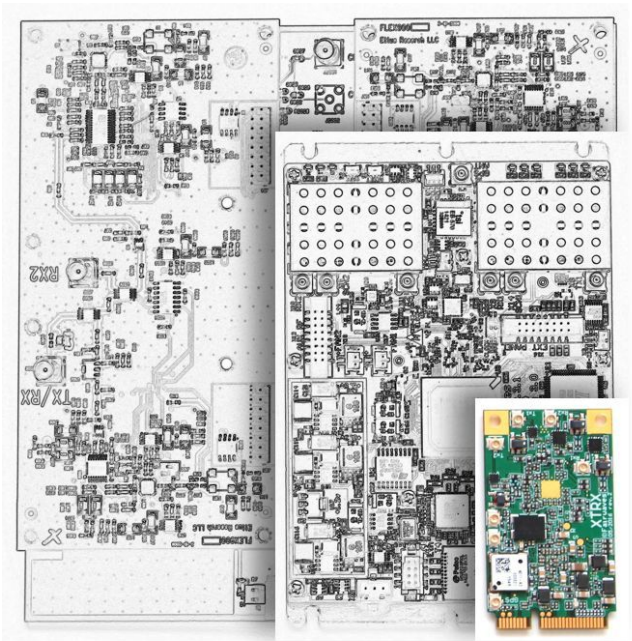
- Small
- Embedded friendly
- Inexpensive
- High performance
  - 120 MHz bandwidth
  - 160 MSPS
  - 2x2 MIMO

What can you pack into 30 × 50.95 mm

- MiniPCIe form factor
- PCIe bus
- Artix7 FPGA (XC7A12T to XC7A50T)
- LMS7002M frontend (2x2 MIMO, DSP inside)
- GPS chip for GPSDO & NMEA
- DAC to tame VCTCXO
- Thermal sensor, SIM card interface, external I2C, RF switches, etc.



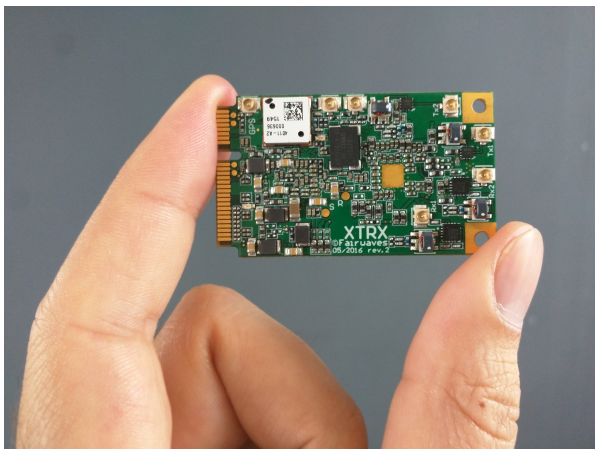
Size in perspective



2006 USRP 1

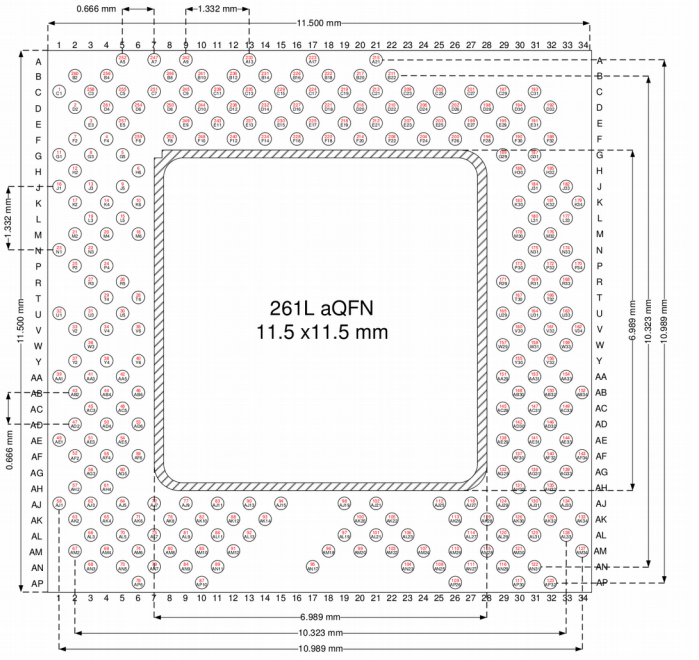
2013 UmTRX 2.3.1

2016 XTRX

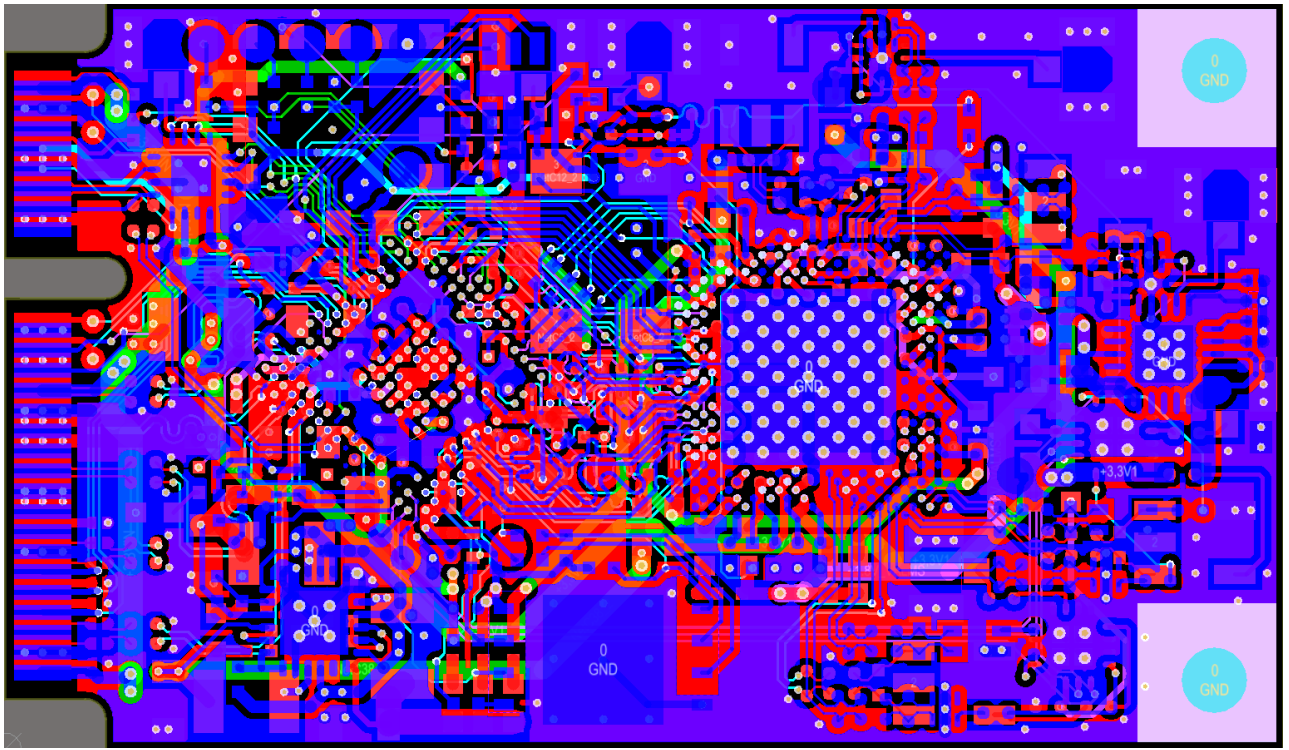


PCB Design challenges

- Small form factor, small component sizes
  - FPGA BGA 10x10mm 0.5mm pitch
  - LMS7 aQFN 0.666mm 2x3 lines deep
- Blind/buried VIAs is a must here
- 1 design: 8 layers initial (blind for different deep)
  - Quoted more than $15`000 just for couple of PCBs
  - PCB should be producing like a sandwich bumping VIAs layer by layer, slow and expensive
  - Never produced it
- 2 design: still 8 layers, but complete rerouting (it was harder since we had more limitations)
  - Only 2-Deep MicroVIA, no buried (drilling in one step vs. classical)
  - PCB produces in 3 steps
  - 5-6 times cheaper!



Routed PCB



PCI Express

- Pros:
  - Low latency (ideal for TDD and CSMA radios and using FPGA for DSP acceleration)
  - High bandwidth (4Gbps per lane)
  - Wide availability (all x86 and some ARMs)
- Cons:
  - Requires OS kernel specific driver for DMA handling
  - Hard to impossible to write a clean cross platform driver
  - Non-trivial debugging
  - No good external hot-plug interface like USB (except Thunderbolt3)

Thunderbolt3

- Chips are easily available (DSL6540 $12.5 on Verical)

  BUT
- Virtually no developer information is publicly available. Only marketing & PR
- Thunderbolt3 is not directly compatible with Thunderbolt2 (like USB3 / USB2 does)
- Need to sign a Thunderbolt license agreement with Intel
- Write a business (!) proposal to let Intel approve it (!!)

  Good News
- There are PCIe to Thunderbolt3 adapters on the market (e.g. T3-HDK $280)
- From OS level it's yet another PCI bus, *no special driver is needed*
  - Caveat: Linux doesn't support all TB3 non-secured devices, must be off in UEFI (to prevent DMA attack). Patches exist, but we did get them to work
- Dell XPS3 with Thunderbolt3 works great in Linux!

T3-HDK PCIe to Thunderbolt3

Goal:

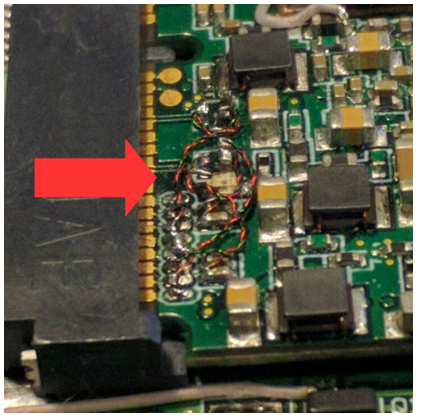
160 MSPS @ 2x2 MIMO, 12 bit = 7680 Mbit/s

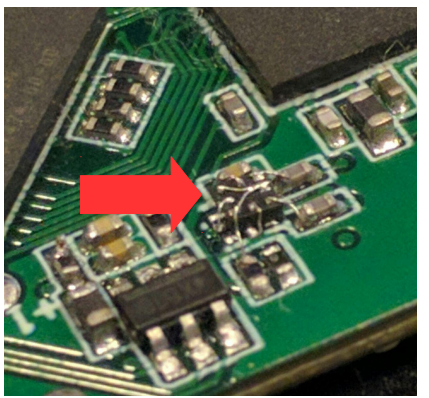
Our first result:

Doesn’t work at all

Deadbugs and other creatures approved by NASA

- FPGA GTP and PCIe bus lanes numbering is reversed
  - FPGA lane 0 -> PCIe lane 1
  - FPGA lane 1 -> PCIe lane 0
- Deadbug
  https://workmanship.nasa.gov/lib/insp/2%20books/links/sections/303_deadbugs.html





Goal:

160 MSPS @ 2x2 MIMO, 12 bit = 7680 Mbit/s

Our second attempt:

Kind of works

FPGA debugging

- Peripherals started to work almost immediately
- DMA did not!
  - No C/C++ style breakpoints, so you have to use testbenches
  - Write testbench for each module, write testbench for integration of modules
  - Simulate all unusual sequences you can think of

FPGA debugging (Cont)

- Good test is your best friend (out of order multiple CplD, delayed completion).
  For each detected problem I added specific test.
- Xilinx ILA, etc. is nice, but can't be used to detect problems in a real-time running system
- Status readback registers are super helpful. Print them in software traces can to understanging a problem:
  - Filling DMA Buffer no
  - Filled DMA Buffer no (i.e. ready to "play")
  - Firing out DMA Buffer no
  - Consumed DMA Buffer no
  - TAG mask are used at the moment (transaction mask "in fly")
  - Counter of delayed DMA bursts
  - State of Frontend, DMA requester block, CplD parser block
- When we see 5'h1f in TAG usage mask all the time => we saturated bus, and data can't be moved fast enough
- When we see Filling DMA bufno == Firing out DMA bufno => we can't provide data on host fast enough

Goal:

160 MSPS @ 2x2 MIMO, 12 bit = 7680 Mbit/s

Naive implementation:

2 MSPS @ 2x2 MIMO, 16 bit = 128 Mbit/s

(60x slower)

Linux system optimization

- Software was not sending/receivind data fast enough
- Use real-time priority
- Fix software bugs - DMA buffer stayed busy longer than it should

Possible driver structures

- Full kernel control

  All data movement & control is done in kernel. Driver exports read()/write()/ioctl()

- Kernel control & “zero-copy” interface

  All control is done in kernel using ioctl() or read()/write() and using mmap() to export device buffers

- “Full” userspace control

  Kernel driver allocates persistent DMA buffers exports via mmap() and also exports control BAR via mmap().

Userspace device control

- No kernel/userspace switches
- Lower latency
- Higher bandwidth

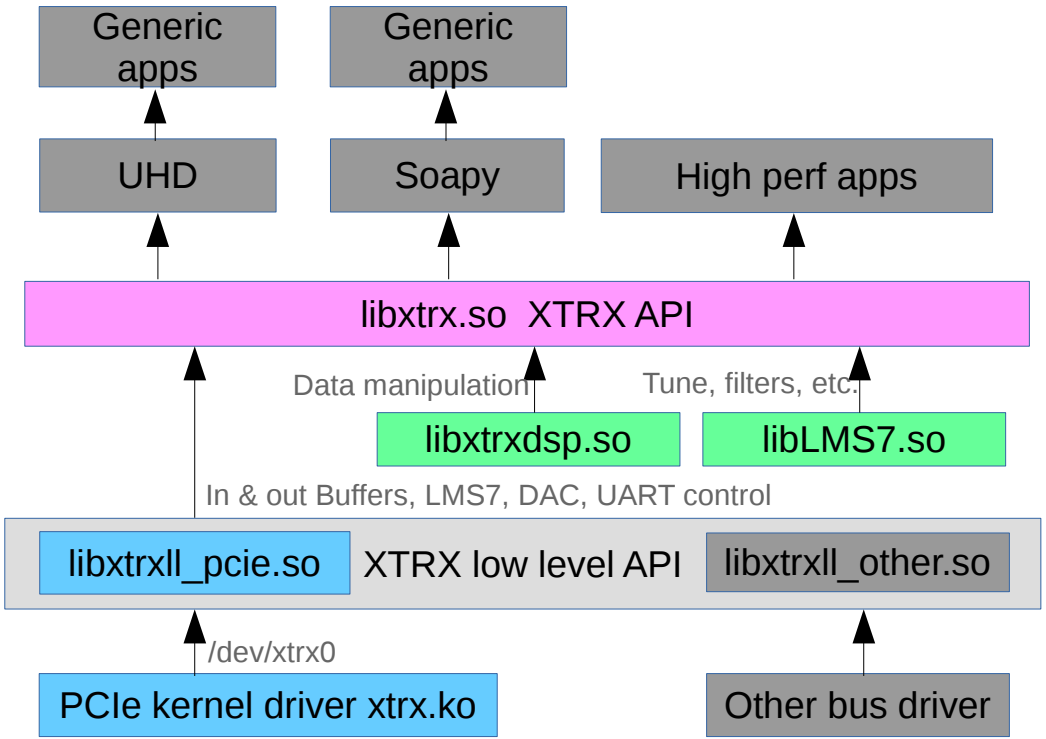
- No ability to get notification when resource is available
- Potentially vulnerable
- Need cache flushing in DMA non-coherent memory
- Hard to share among multiple programs

To poll or not to poll?

- Pure polling is good for real high bandwidth (DPDK, Infiniband RDMA, etc)
- Requires a dedicated CPU core
- Not good for lower bandwidth - wastes CPU time, we need notification
- In our design we have DMA bufno filling counter and DMA bufno reading counter, so we can generate IRQ based on their difference
- In driver this IRQ is waking poll() kernel wait queue. (epoll()/poll()/select() user functions)
- **Our choice**: poll for high bandwidth, notification for low bandwidth

Library architecture

- Minimal dependencies
- Plain C implementation
- XTRX Low level library working with device driver. OS specific and Bus specific (PCIe only currently).
- Auxiliary libraries. Mostly cross platform. LMS7 control, data manipulations and preparation.
- XTRX API level library (relies on all above) and user progams may use it



Writing a Linux driver

- Main Issue:

  Classic Linux Device Drivers book (3rd edition, 2005) is already outdated, have to read docs from the mainline kernel

- Decisions:
  - TTY devices for GPS (with DCD 1pps event) & SIM card
  - KPPS/LinuxPPS kernel API to get accuracy for time synchronization
  - Thermal sensor exports kernel's thermal API (works with sensord)
  - IO Control BAR, DMA TX buffers, DMA RX buffers mmaped to userspace via single mmap() with different offsets.
    - pci_alloc_consistent() => remap_pfn_range to VMA
  - Helper ioctl() like to get shared access to LMS7 SPI (for debugging)
  - Poll() to receive notification from interrupts

Goal:
160 MSPS @ 2x2 MIMO, 12 bit = 7680 Mbit/s

Software optimized implementation:
10 MSPS @ 2x2 MIMO, 16 bit = 640 Mbit/s
(12x slower)

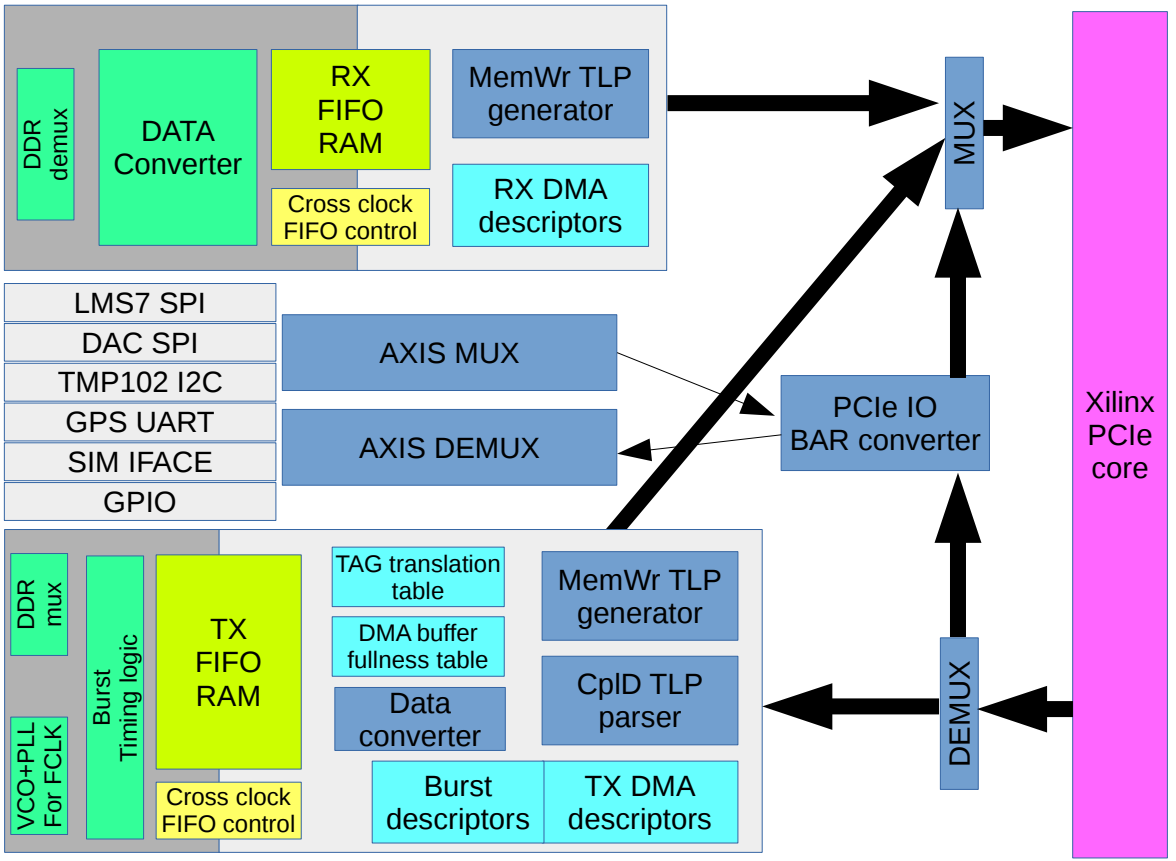
We wrote a slide deck with PCIe gotchas, but removed it, because it’s a couple hours by itself.

PCIe Challenges

- Available (open source) cores are not optimized for real time applications
- Xillybus is nice, but doesn’t perform well in high bandwidth apps
- Xilinx AXI Bus interface to PCIe is interesting but it requires extra FPGA resource for another abstraction layer
- => write your own

FPGA PCIe architecture

(no, we won't go through this now)



PCIe fun

- Byte count
  - Worked well on Atom but completely hang Core i7 (even FS got corrupted)
  - Turned out incorrect remaining Bytes count in Completion - 0 value equals to 4096 bytes (and not 0 bytes) by the standard.
- Completion
  - Not sending a completion to request will lead to device aborting. Due to backward compatibility in x86 architecture no exception is raised to the software layer, but rather any further register access will return 0xffffffff. If you assume that ‘1’ in a register indicated that your device is ok, you’re in trouble. Use ‘0’ or (even better) 2'b01 or 2'b10 to avoid misreadings

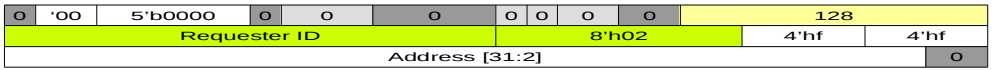
PCIe fun (Cont)

- Unpredictable crushes, hangs that hard to catch initially. Hard to understand what's wrong - may be FPGA is sending crap, may be kernel driver is doing something bad...
- Memory corruption
  - Turned out the low level library was setting incorrect address for DMA buffer for device to write to. Solution: used Xilinx ILA (Integrated Logic Analyzer) to verify that addresses are (in)correct.
  - Exotic platforms like Sparc and IBM have additional address translation table to map bus addresses to physical addresses => naturally prevent this issues. Intel x86 also has this layer, but it's used for virtualization and can't protect from this issue.
- Incorrect order of driver deinitialization may lead to a write to an already freed DMA buffer

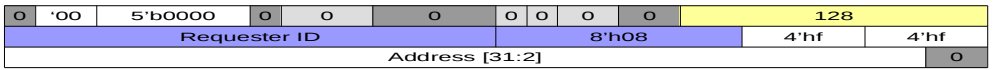
Why DMA is hard

- Device can also initiate MemRd/MemWr transactions, you just need address and size!
- Easy to say, but not to make it
- DMA RX is easy. We need move data from FIFO ram to PCIe using, no “completion” (ack) required.
- DMA TX is complex.
  - Data coming in “completion” transactions may come out of order.
  - High latency, you **must** support multiple transactions in flight.

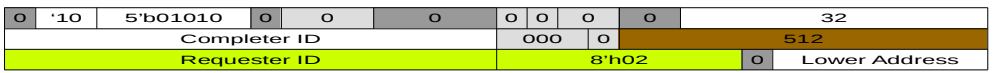
PCIe Read Transactions

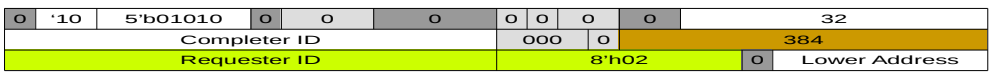


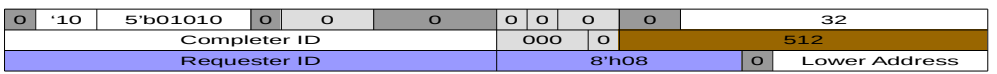
Requested 512 bytes with tag 02

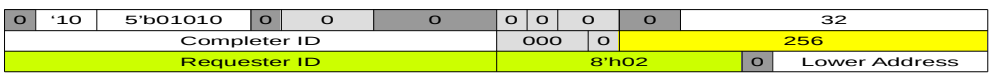


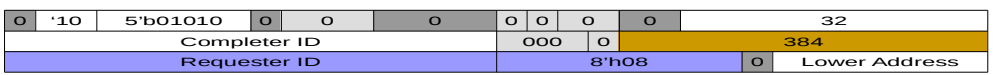
Requested 512 bytes with tag 08

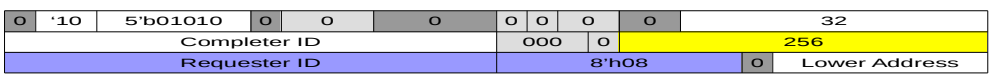


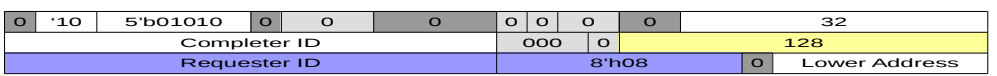


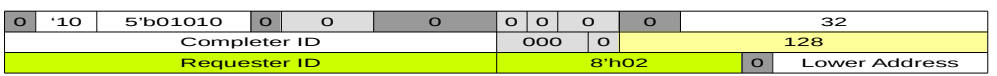












Goal:

160 MSPS @ 2x2 MIMO, 12 bit = 7680 Mbit/s

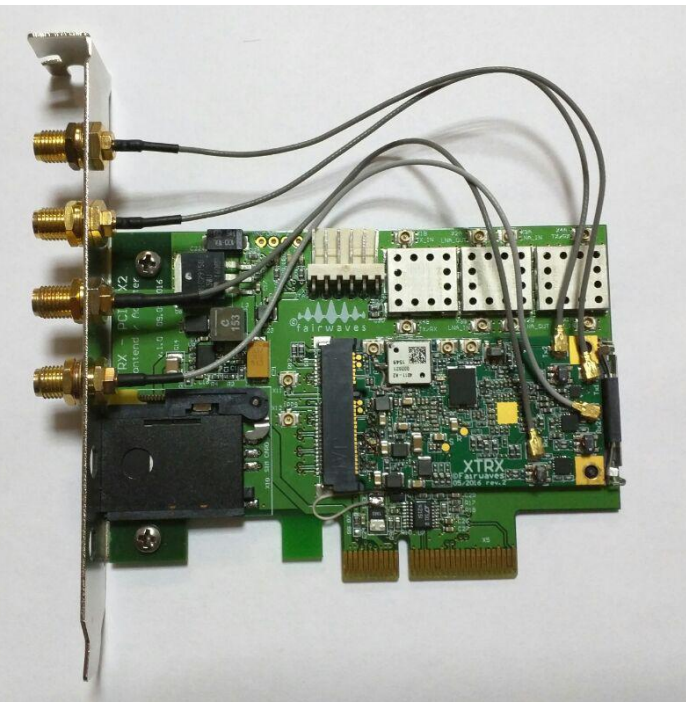
FPGA optimized implementation:

20+ MSPS @ 2x2 MIMO, 16 bit = 1280 Mbit/s

(6x slower)

PCIe lane scalability

- PCIe lanes allows to scale bandwidth
  (x1, x2, x4, x8, x16)
- Issue:
  miniPCIe second PCIe lane is optional
  and rarely implemented in motherboards
- Passive conveter miniPCIe → PCIe x2
  (also allows to add RF amps, LNA,
  expose JTAG, SIM card interface, etc)



Goal:

160 MSPS @ 2x2 MIMO, 12 bit = 7680 Mbit/s

With PCIe x2:

50 MSPS @ 2x2 MIMO, 16 bit = 3200 Mbit/s

(2.4x slower)

Sample size scaling

- Driver is designed to support 8-bit, 12-bit, 16-bit samples.
- Allows to increase MSPS at the same bandwidth
- 12 bit is native for LMS7
- 16 bit is easier for CPU
- Using 8 bit somewhat reduce dynamic range, but in some use cases it’s acceptable

Goal:

160 MSPS @ 2x2 MIMO, 12 bit = 7680 Mbit/s

With 8 bit samples:

50 MSPS @ 2x2 MIMO, 8 bit = 1600 Mbit/s

(4.8x slower)

WTF??

High bus speed design

- FPGA to LMS7 bus gets flaky after 100 MHz (50 MSPS)
- We chose 1.8V bus level to reduce power consumption
- Wrong choice for a high speed bus
- We’re redesigning to increase voltage on the bus (without increasing voltage of other components)

Goal:

160 MSPS @ 2x2 MIMO, 12 bit = 7680 Mbit/s

With proper bus (theory):

50 MSPS @ 2x2 MIMO, 16 bit = 3200 Mbit/s

66 MSPS @ 2x2 MIMO, 12 bit = 3168 Mbit/s

100 MSPS @ 2x2 MIMO, 8 bit = 3200 Mbit/s

(1.2x slower)

Upgrade PCIe Gen1 to Gen2

- PCIe Gen 1.0 (2Gbit/s) → 2.0 (4Gbit/s) → 3.0 (~8Gbit/s)
- Right now only 2.0 is available in low-cost FPGA
- We made a mistake and bought wrong FPGA
  - XC7A35T-**1**CPG236C instead of XC7A35T-**2**CPG236C
  - Supports only PCIe Gen 1 :(
  - Will be fixed in the next prototype

Goal:

160 MSPS @ 2x2 MIMO, 12 bit = 7680 Mbit/s

With PCIe Gen2 (theory):

100 MSPS @ 2x2 MIMO, 16 bit = 6400 Mbit/s

125 MSPS @ 2x2 MIMO, 12 bit = 6336 Mbit/s

200 MSPS @ 2x2 MIMO, 8 bit = 6400 Mbit/s

(1.2x slower)

Practice vs Theory

- PCI bus has overheads due to payload size limits
- Standard specifies 4kB maximum payload
- Intel Atom and Core have 128 byte payload
  - 87% (6960 Mbps) efficiency theoretical maximum
- Intel Xeon has 256 byte payload
  - 92% (7360 Mbps) efficiency theoretical maximum
- Now subtruct protocol overhead..

Platform differences

- Embedded Intel Atom
  - Surprisingly small PCI bus jitter
  - Tested without video subsystem
- Intel Core
  - Performed worse than Atom (sic!)
  - Supposedly due to interference with embedded video card sitting on the same PCIe bus
  - Late PCIe transactions when you move a window
- Intel Xeon
  - Really smooth
  - Got maximum performance (about 5%-7% higher than Atom)
  - Tested with discrete video card

Workshop?

for users & for deeper PCIe dive