Why not build an oscilloscope?

…HOW HARD CAN IT REALLY BE?

THOMAS OLDBURY

Background

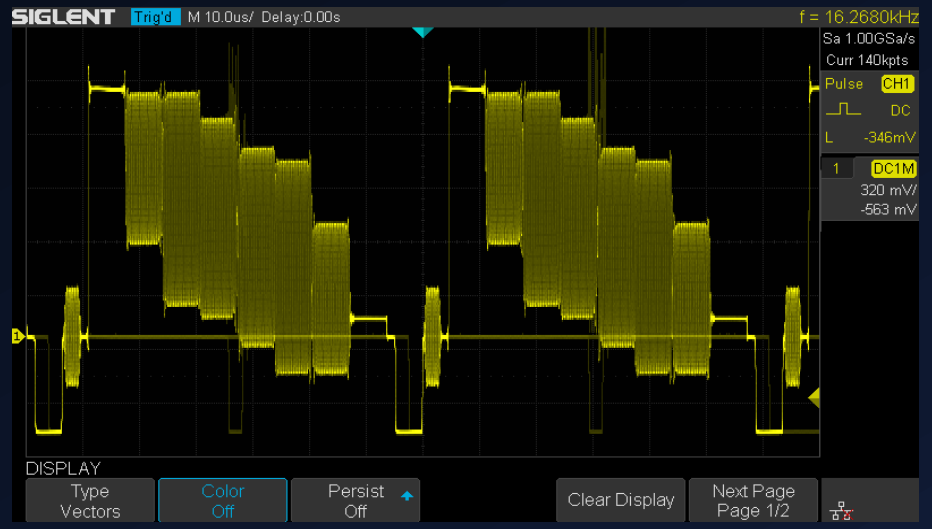
- Open source hardware and software
  - Everything is available – schematic, PCB design, FPGA firmware…
- *Incomplete* project – but demonstrated capability
- Performance roughly equivalent to a £400-500 digital oscilloscope
  - Rigol DS1104Z / Siglent SDS1104X-E / Pico Tech. 3000 Series

Motivation

- At the time of development, there were very few open source oscilloscope projects available
- The situation is improving with ScopeFun / ThunderScope but there are still no stand-alone digital oscilloscopes that are fully open source *(maybe this will change!)*
- It sounded like an interesting challenge (it definitely turned out to be one)
- I wanted to learn more about FPGA design and “full stack hardware design”.

Goals

- 4 channel oscilloscope with 100MHz analog bandwidth
- 1GSa/s (one billion per second) sampling rate
  - Multiplexed between all enabled channels (common in entry-level oscilloscopes)
- Intensity graded waveform support, requiring high waveform capture and render rate
- Decent capture memory (~100MB+)



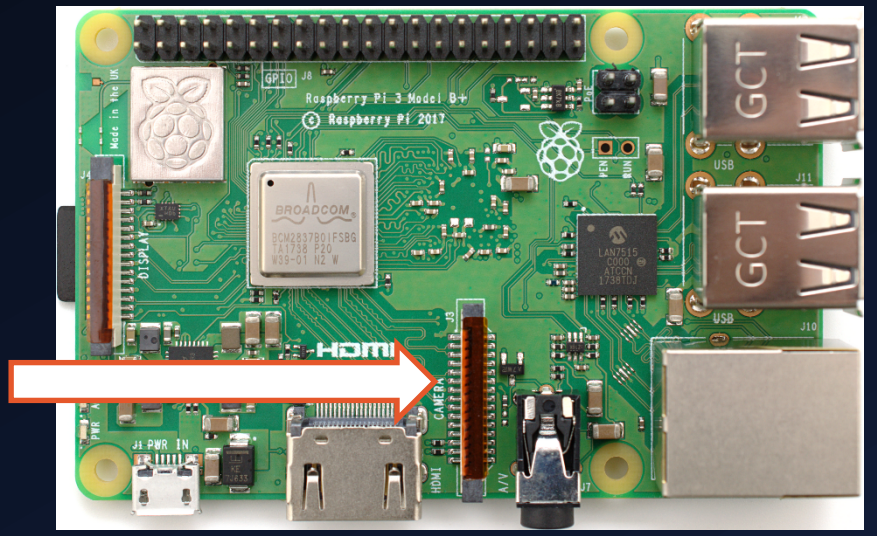
© Siglent Technologies (fair use example of intensity grading on commercial instrument)

Early stages (mid 2018)

- I thought basing the oscilloscope on a Raspberry Pi made a lot of sense…
- Why?
  - Fast Linux computer (at the time quad core, 64-bit 1.2GHz + 1GB RAM)
  - Lots of open source examples available
  - Cheap
  - *Nerd factor*
  - Camera interface!

Early stages (mid 2018) – *Camera interface?*

- Every Pi since the first generation model has featured a MIPI CSI-2 camera interface on board
  - This is capable of driving a 1080p camera at 30 frames per second.
- The video interface from the camera is *not compressed*.
  - This is a lot of data!
  - How much? Rough numbers:
    - 1920 pixels x 1080 lines x 3 RGB bytes/pixel x 30 Hz (it can do more)
    - 1.86 Gbit/s – almost **twice** the speed of gigabit Ethernet, faster than Gen1 SATA and 4x that of the onboard USB2.0



Early stages (mid 2018) – How do we use the camera?

- MIPI CSI-2 is a *secret* protocol
  - It's used by smartphone manufacturers to connect camera to SoC – no need for "average engineers" to know how it works, right?
- Registration fees start at $2,000 *per quarter* to gain access to datasheets and specs
- The Pi's drivers for CSI-2 are part hardware, part binary blob – limited opportunity to reverse engineer at that level
- I only had access to a slow oscilloscope (100MHz) – capturing the raw CSI-2 data (~2000MHz) was next to impossible … *or was it?!*

Early stages (mid 2018) – Let's reverse engineer CSI-2!

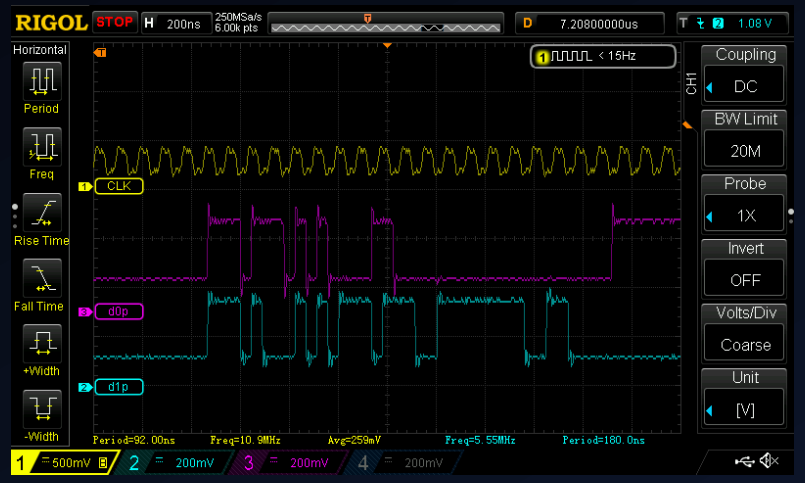
- The Pi camera (Gen 2) is a Sony IMX219 sensor with the MIPI core built in (datasheets are available from the Pi foundation)
- The camera has an external crystal which is used to drive a PLL (phase-lock loop)
  - Generates image sensor sampling clock and MIPI clocks through various dividers
- *What if we were able to slow down the camera?*

Early stages (mid 2018) – Let's reverse engineer CSI-2!

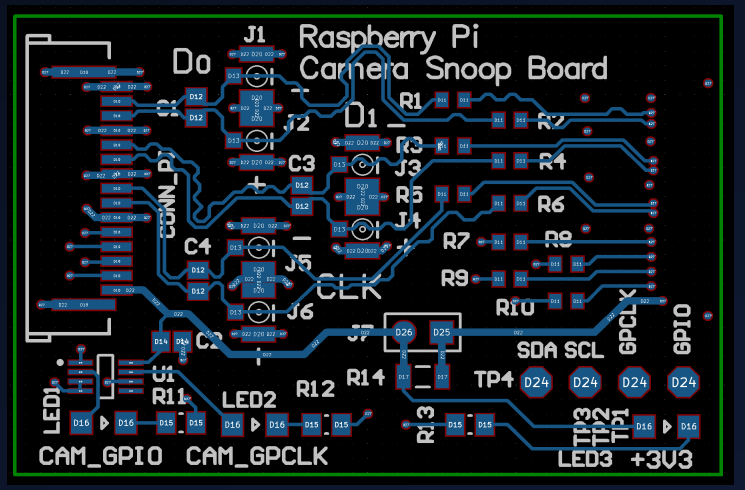
- My first attempt was to remove the oscillator from the camera and inject a low frequency clock
  - Cheap clone cameras were sacrificed for this goal!
- This unfortunately fails – the camera PLL does not achieve lock and the camera will not function at all
  - PLLs regulate their output and indicate to the rest of the system if lock has been achieved – a failure to lock may indicate an unstable PLL
- So I need another way to slow the clock down… Let's intercept the I^2C traffic!

Early stages (mid 2018) – Let's reverse engineer CSI-2!

- By making a board between the Pi and camera I could snoop…
- I could write my own I²C registers to what I wanted…
- I was then able to see the raw protocol at 1/160th its original speed on a normal oscilloscope by changing the PLL divider registers



CSI-2 Header – on a cheap oscilloscope!



Pi Camera CSI-2 "snoop board"

Early stages (mid 2018) – Implementing CSI-2

- I still needed a bit more:
  - Additional documentation from a leaked early draft of CSI-2
  - Agilent Technologies whitepaper on CSI-2 debugging
  - Xilinx XAPP894 “D-PHY Solutions” for implementing CSI-2 with a general purpose FPGA
- My first “proper” FPGA core – huge learning experience
- Implemented it as a “slow output” core initially, matching the slow camera output
- Took about 2 months of nothing at all – the Pi does not give you *any feedback* when the input format is wrong
- A particular headache came that the Pi does not fully implement the CSI-2 specification – *continuous clock is not supported at all*

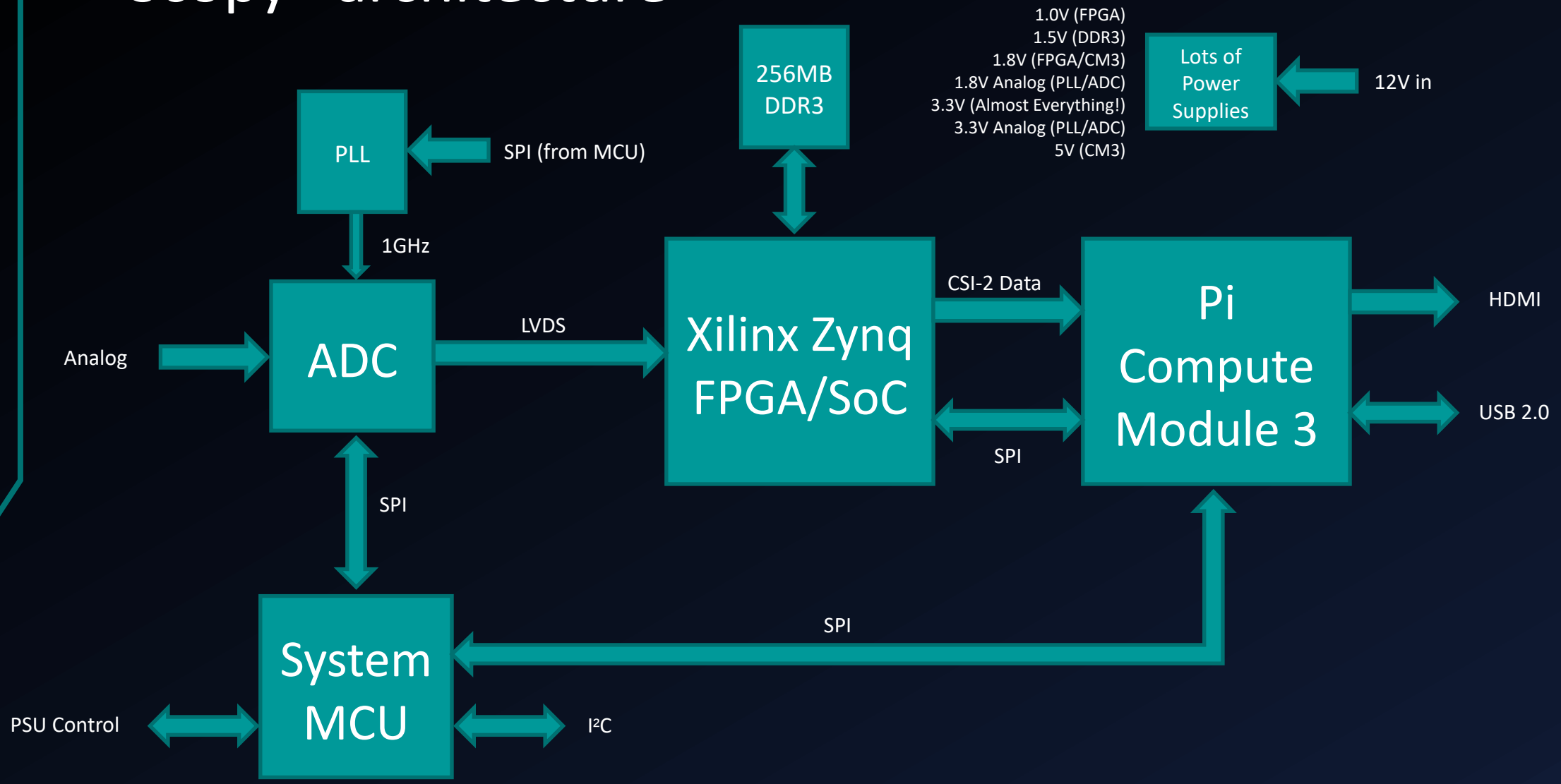
Early stages (mid 2019) – first proper FPGA work

- I was eventually able to get my core to run and output at up to 2Gbit/s on a Zynq development board
  - “Production” build ran at 1.6Gbit/s for stability
- Up to this stage I had been working on a cheap Zynq development kit – but encountering issues with signal integrity
- It was becoming obvious the only way forward was to make … *real hardware*

A bit about oscilloscope architecture

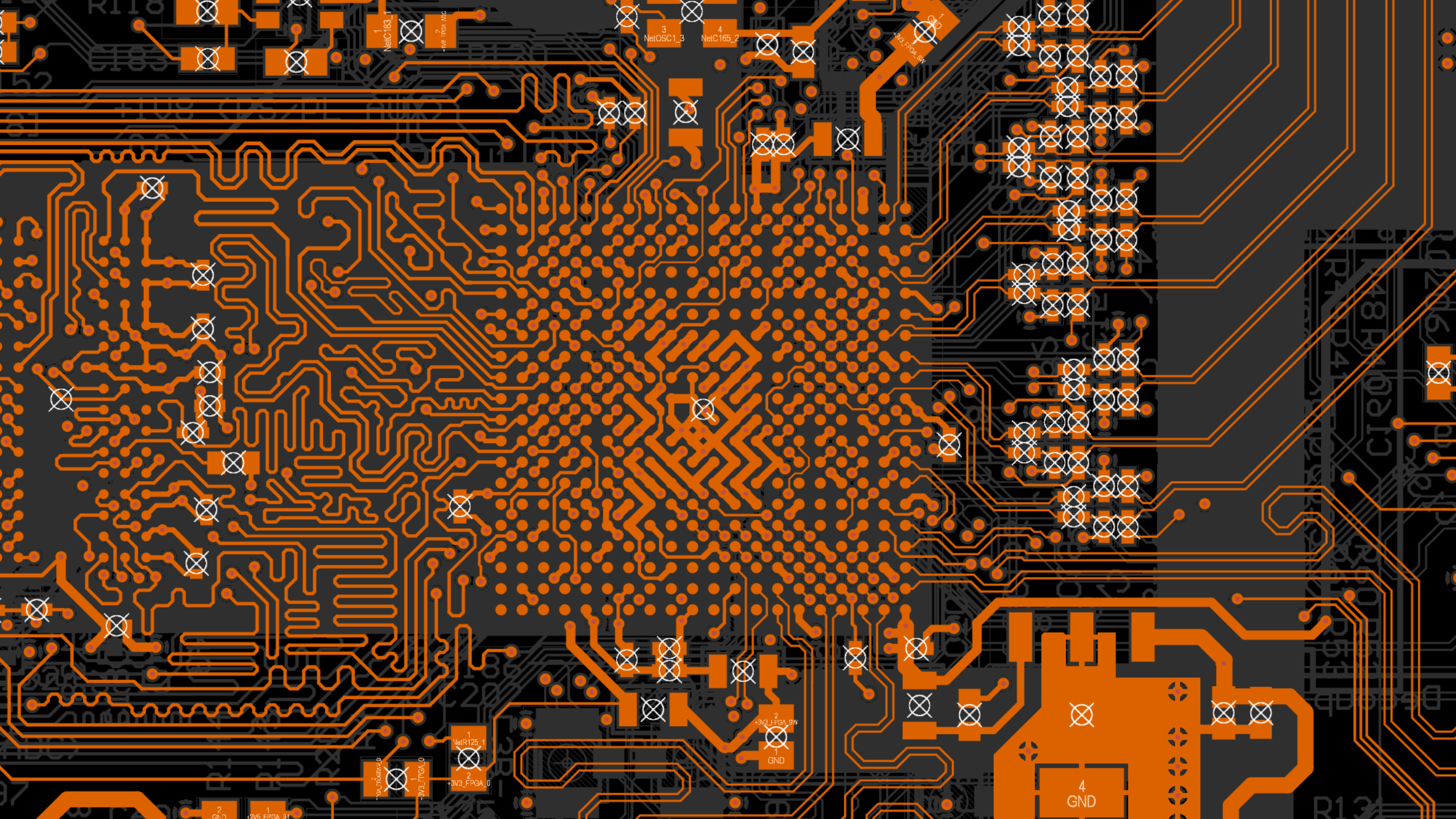
- Most oscilloscopes have an FPGA and a main processor
  - Sometimes the same device, sometimes the FPGA is an ASIC
- Acquisition memory needs to be fast
  - Consequentially, on many older scopes, it is limited in size – SRAM type or built into FPGA/ASIC
  - But cheap DDR is now available – we can do it all with conventional memory
- All digital oscilloscopes need an ADC and PLL for that ADC
- Analog front end is also important
  - But for now I put that to the side – the digital part was more interesting
- Trigger can be done digitally if your ADC is fast – most scopes do this

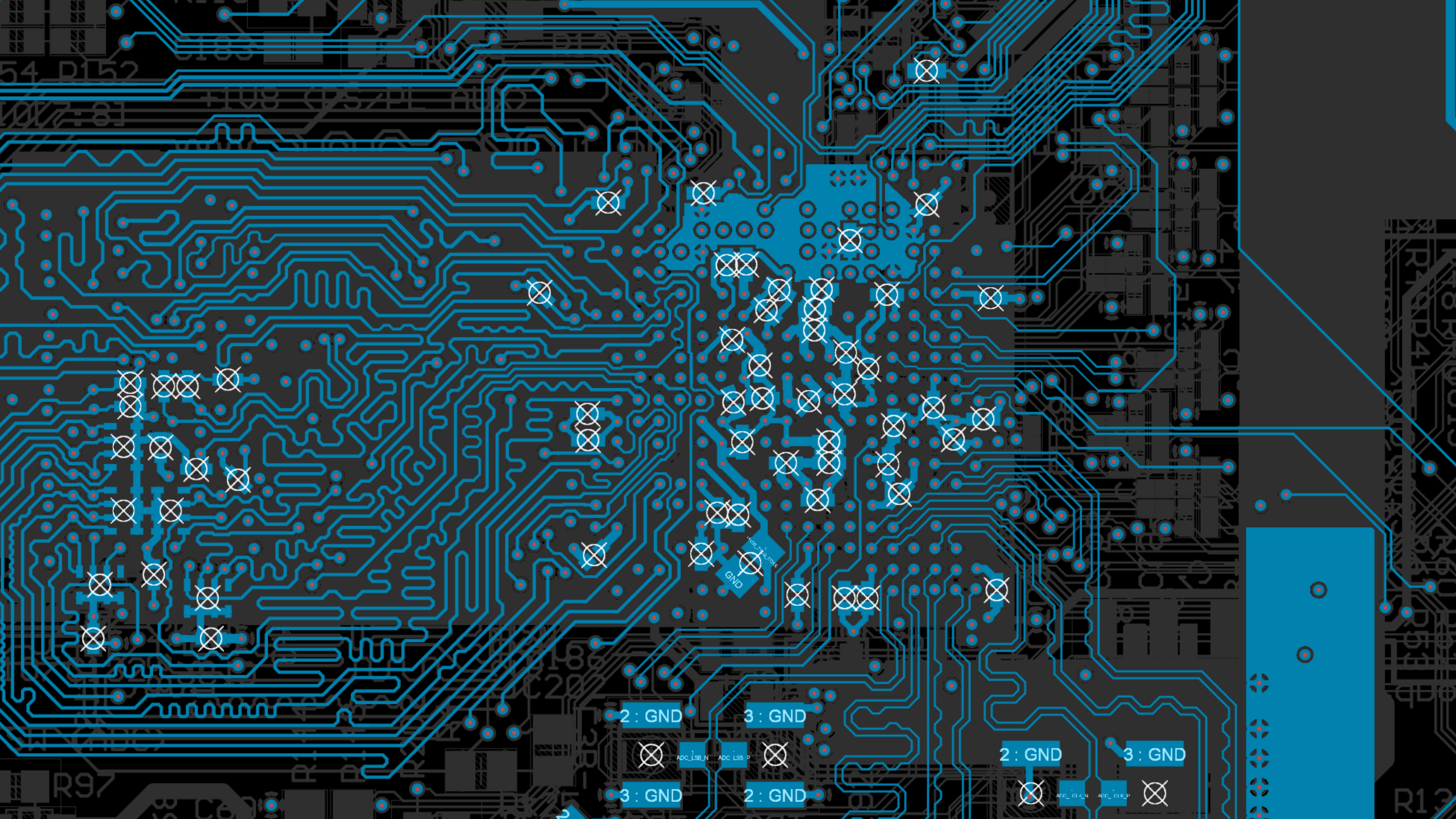
“Scopy” architecture

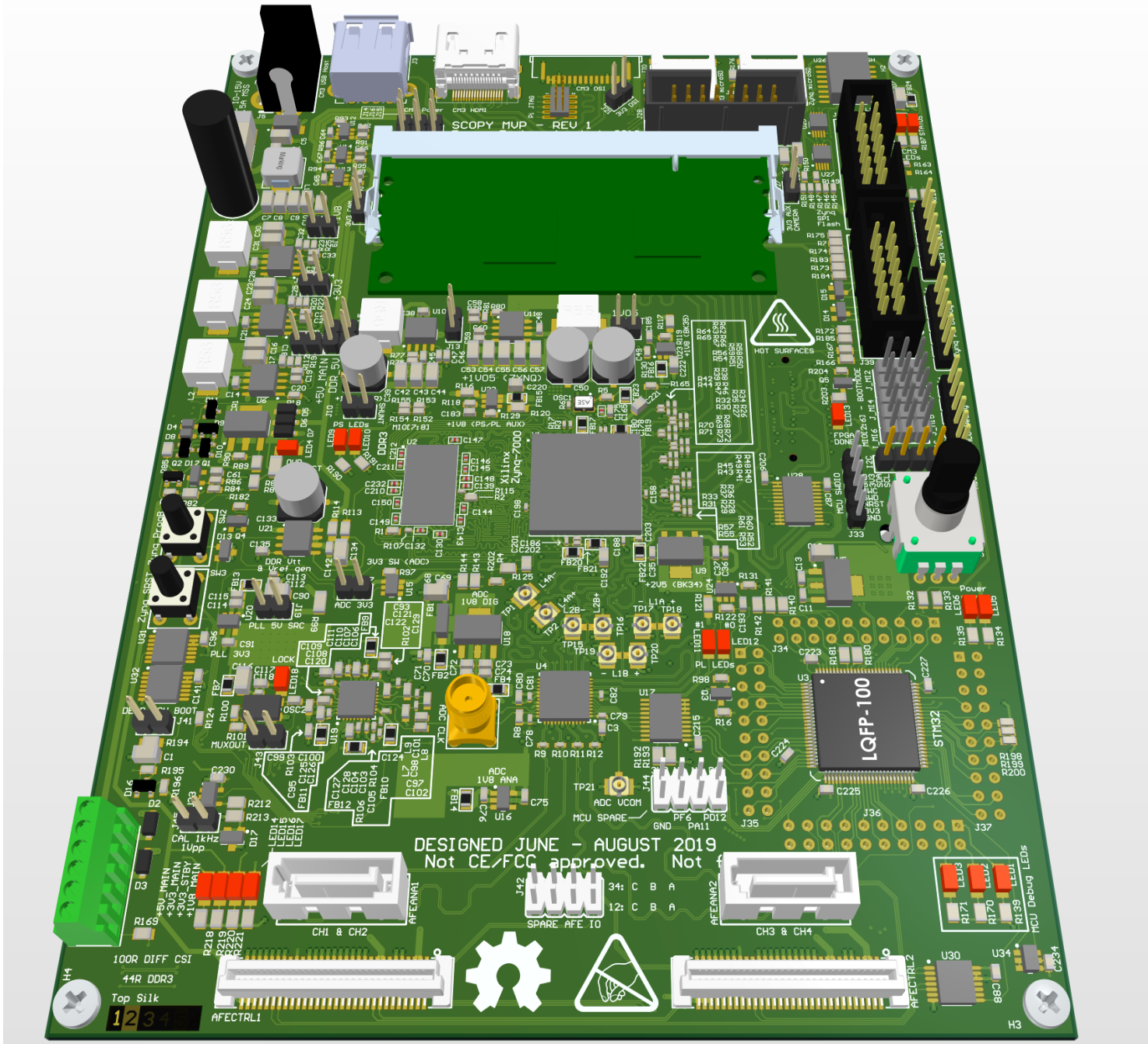


Designing the hardware

- Designed in Altium CircuitMaker (free, but proprietary)
- A huge amount of effort expended on validating the design – right first time was the aim
- To keep costs low, I needed to stick to 6 layers
  - Not as many as you think: two high speed, two power, and two ground planes!
  - Conventional stack up – so no buried/blind vias either…
- Routing DDR3 and length matching by hand (length tuning not included in CircuitMaker!)
- Keeping the layer count low meaning very difficult breakout of the FPGA – lots of pin-swapping


1
NetC183_1
2
+VIN_FPGA_MSC
3
NetOSC1_3
4
NetC165_2
1
GND
+3V3_FPGA_SW
2
+3V3_FPGA_SW
1
GND
1
NetR125_1
2
+3V3_FPGA_0
4
GND


2 : GND
3 : GND
ADC_LSB_N
ADC_LSB_P
3 : GND
2 : GND
2 : GND
3 : GND
GND
R97


SCOPY MVP - REV 1
DESIGNED JUNE - AUGUST 2019
Not CE/FCC approved. Not f
LQFP-100
STM32
Xilinx Zynq-7000
HOT SURFACES
CH1 & CH2
CH3 & CH4
SPARE AFE IO
MCU SPARE
ADC VCOM
PL LEDs
PS LEDs
MCU Debug LEDs
100R DIFF CSI
44R DDR3
Top Silk
AFECTRL1
AFECTRL2
AFEANA1
AFEANA2

Building the hardware

- PCB's made in China (6 layer, Eurocard size)
  - Cost around £200 to have five boards made
  - Excellent quality by JLCPCB
- Assembled in Sweden! 
  - Too complex to hand assemble, machine assembly the way to go.
  - Thanks to Fredrik at Svensk Elektronikproduktion for giving a discount on the build because he appreciated the idea!
  - Excellent quality and rapid turn-around
  - A generous friend (RB) part-funded the project but in all I spent about £1,200 on the project – but it was well worth it

The hardware arrived…

- It was a bit like Christmas… waiting for the FedEx van to arrive
- I spent the next week bringing up the hardware and validating it
  - Writing small test scripts to exercise the hardware
- A few mod wires required, but board booted and ran correctly!
- So time to start developing the FPGA HDL and software…

HDL/Software stack (Zynq/FPGA)

- Written in standard Verilog
- Built using a mix of AXI DMA blocks and a software controller to handle memory copies
- High CPU overhead on the Zynq cores serving interrupts
  - … but we sort-of don’t care, since nothing else runs on it?
- Not using embedded Linux – single core application on baremetal
- Zynq gives advantage of hard DDR3 core (less timing constraints, faster build) – and no need to add softcore CPU/memory block

Software stack (Pi)

- Written in Python 3
  - Ultimately, I think this was a mistake… C++/Rust would be better?
  - Had to separate high speed and UI stuff … headache
  - Used ‘multiprocessing’ to have many threads without GIL … headache
  - SPI interface from Pi to Zynq also a major headache – maybe USB2.0 for control would have been better?
- Using MMAL drivers with a custom Python library (contributed by a friend) to capture the camera framebuffer
- A highly optimised piece of C called “armwave” which renders the waveform
  - Future goal would be to render the waveform on the FPGA
  - One thought was to render on the Pi’s GPU, but performance was an issue

Achievements

- A scope able to capture 240MB of data at 1GSa/s and display it, continuously
- Edge trigger function implemented in the fully digital domain
- Pre- and post-trigger function (capturing data “before” the trigger happens)
- Multiple timebases
- Sending this data to the Pi and displaying it
- Intensity graded waveform on the display
  - >20,000 waveforms per second captured and rendered (in some conditions!)

Conclusions…

- A great deal of effort was spent on getting the Pi 3 CM3 to work with the camera interface
  - … only for the Pi foundation to release the Pi 4 and related CM4 with PCI Express support about a year later
  - So the camera stuff is a bit redundant (so much better/easier/faster to do via PCI Express)
  - But it was still an interesting problem to solve!
- FPGA design:
  - *I didn't really know what I was doing at the time* – but I did teach myself FPGA design in the process
  - Too much time debugging in hardware, not enough time spent building test benches and models

Thanks for listening!

toldbury@gmail.com
github.com/tom66

Demo to follow (hopefully!)