CrypTech

2015.07.23

Praha

Origins

- This effort was started at the suggestion of Russ Housley, Jari Arkko, and Stephen Farrell of the IETF, to **meet the assurance needs of supporting IETF protocols in an open and transparent manner.**

- **But this is NOT an IETF, ISOC, ... project**, though both contribute. As the saying goes, "We work for the Internet."

Goals

- An open-source reference design for HSMs, not a manufactured product
- Scalable, first cut in an FPGA and CPU, plan higher speed (ASIC)options later
- Composable, e.g. "Give me a key store and signer suitable for DNSsec"
- Reasonable assurance by being open, diverse design team, and an increasingly assured tool-chain

CrypTech Project

- An Open Design, not a Product
- Open – everything (docs, design, code)
- BSD, CC license for all we develop
- Diverse engineers and review
- Support for transparency, testing, …
- Multiple contributors: IETF, Comcast, Google, .SE, SUNET, PIR, ISOC, Afilias, RIPE, IANA, Cisco, etc.

Diverse Engineering

Verilog Göteborg & Moscow

Hardware Adaption Layer (HAL) in Boston

Software, PKCS#11, … from Boston

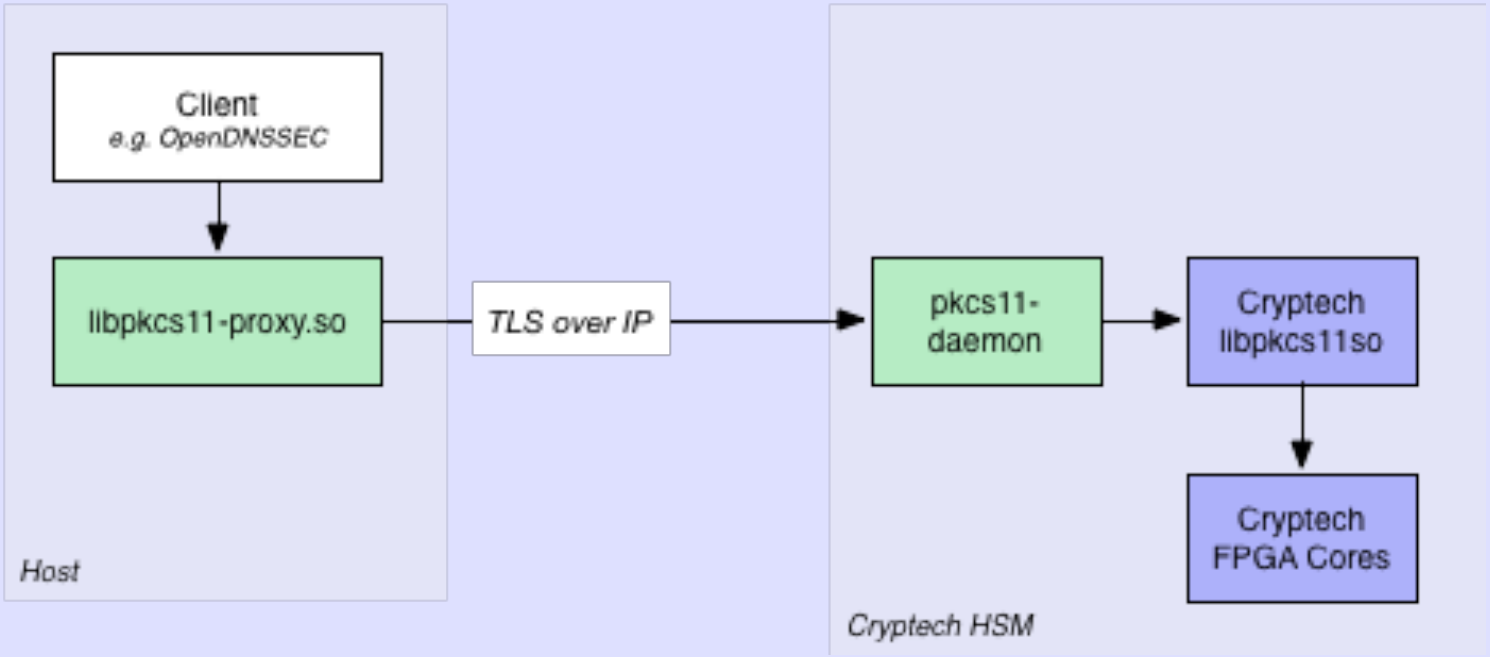
TRNG advice from Germany and States

Hardware Design & Build from Stockholm

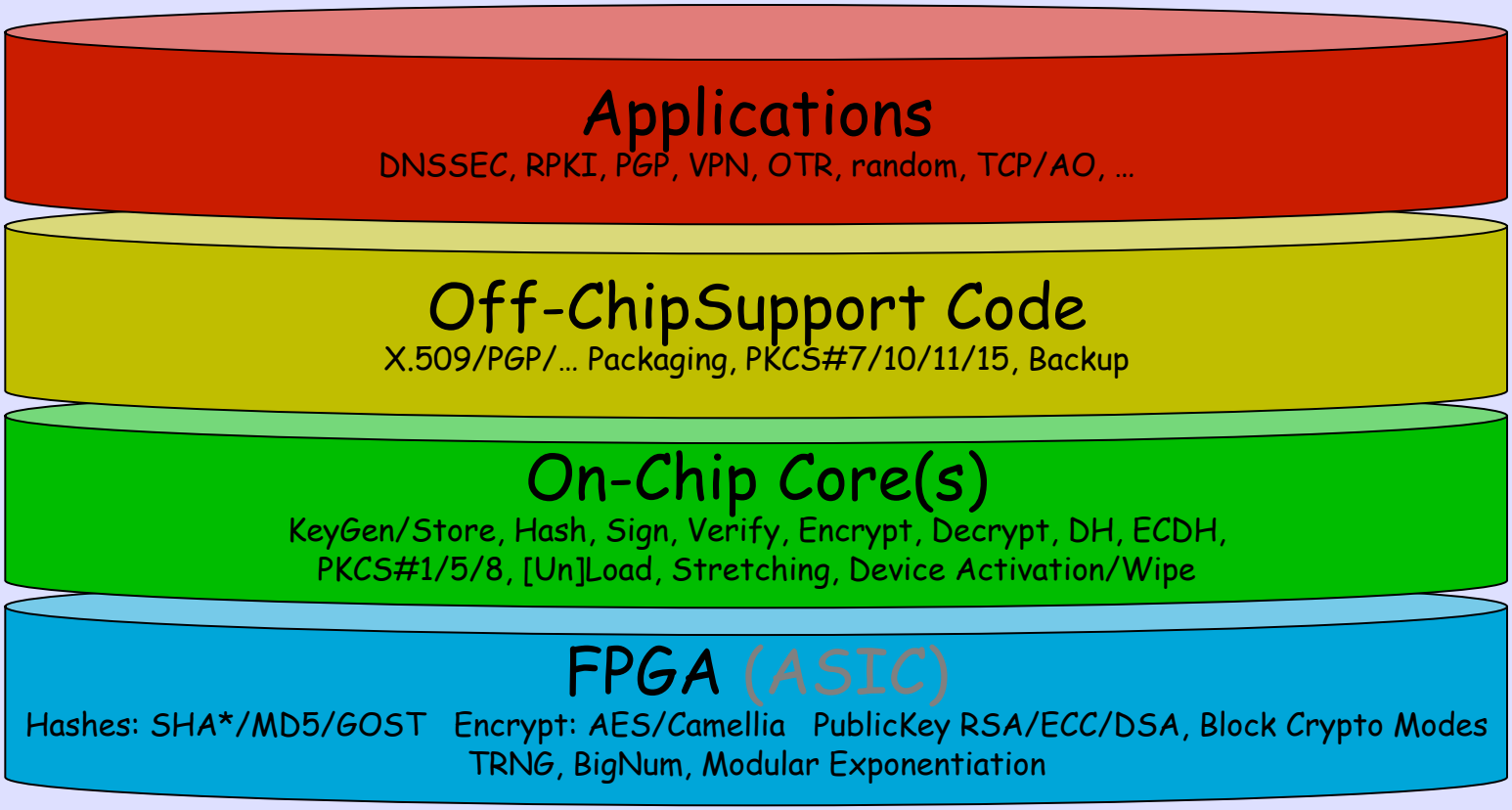
DNSsec from Göteborg & Stockholm

Engineering coordination from Tokyo

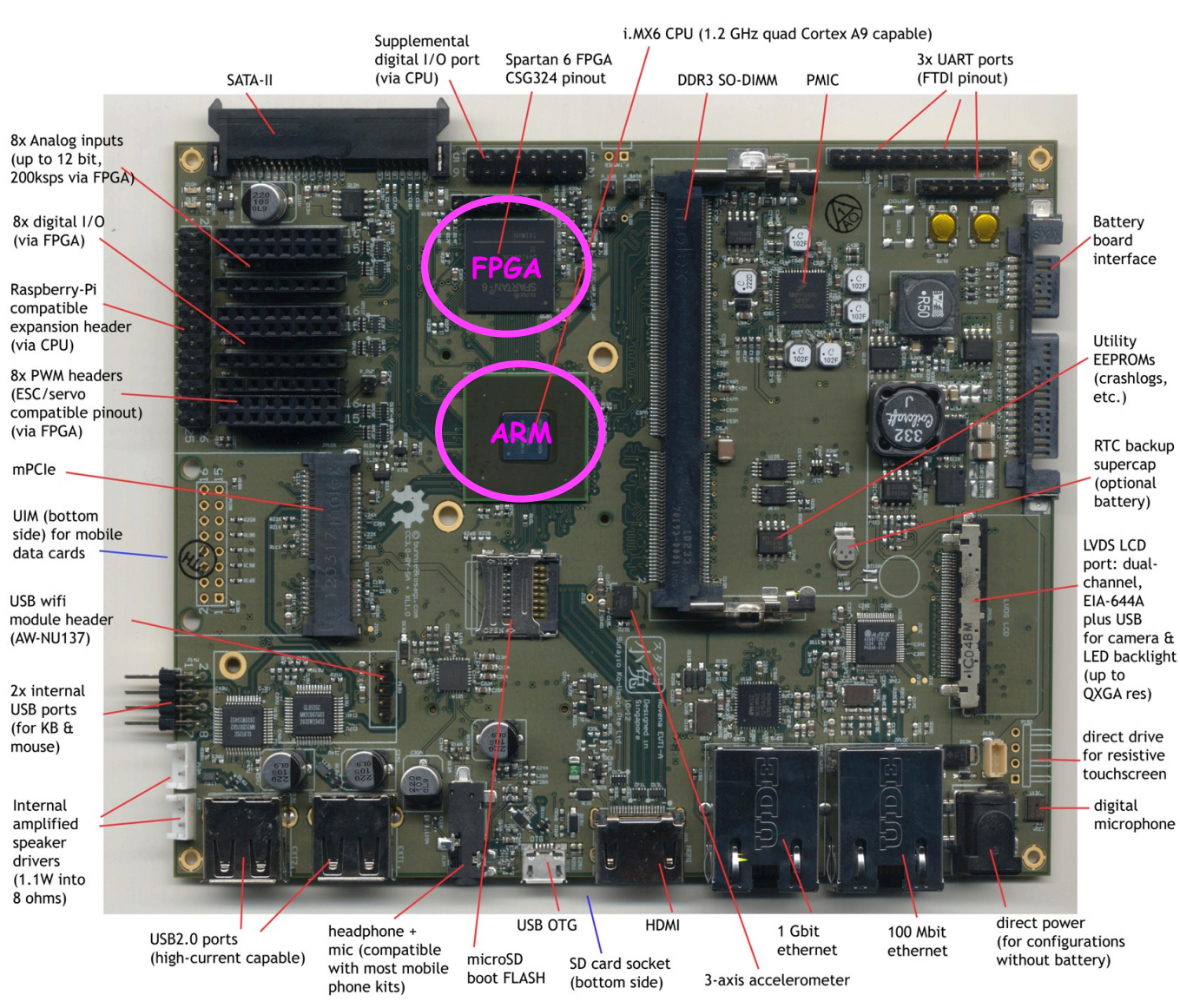
Novena Development Board Setup



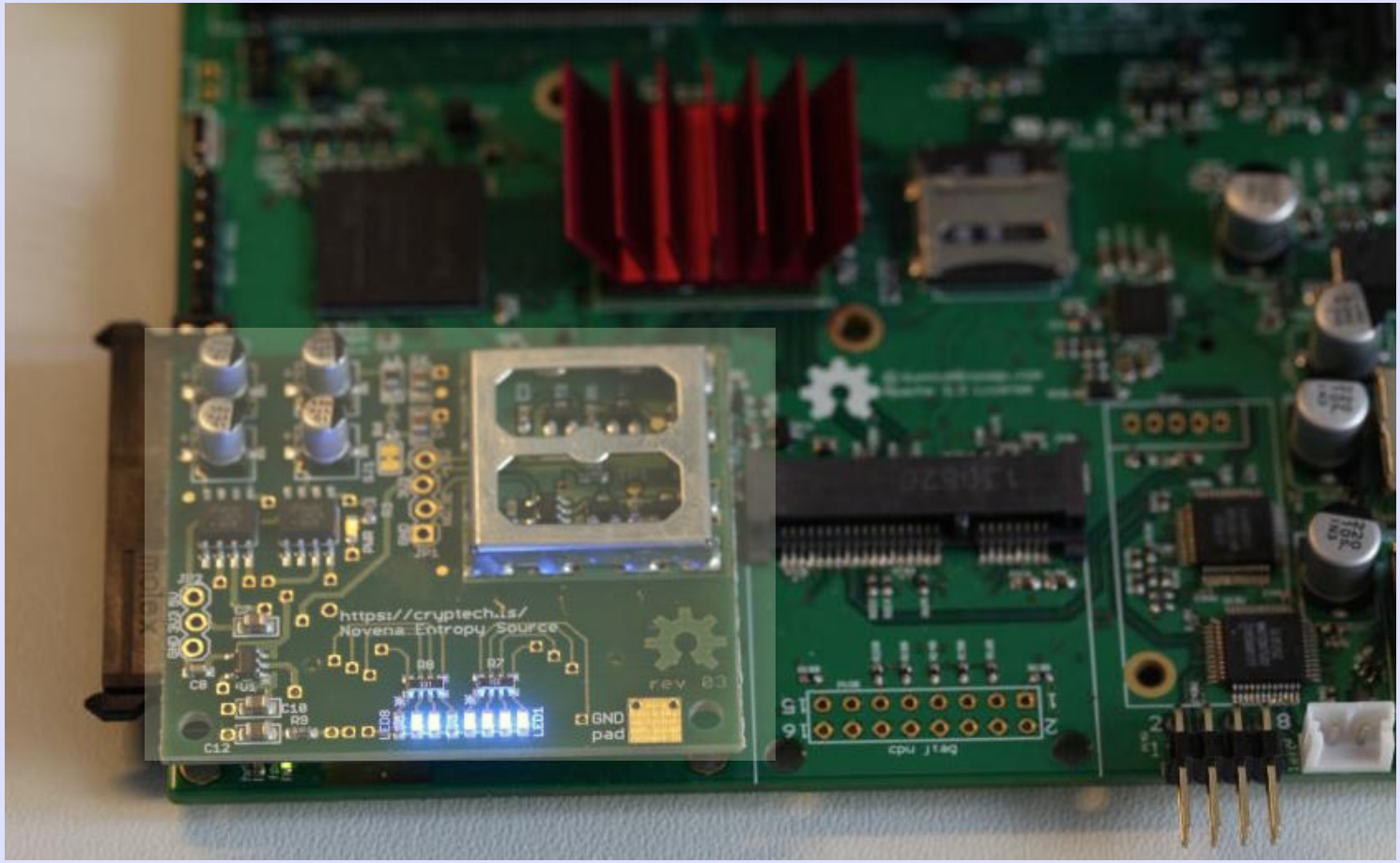
Layer Cake Model



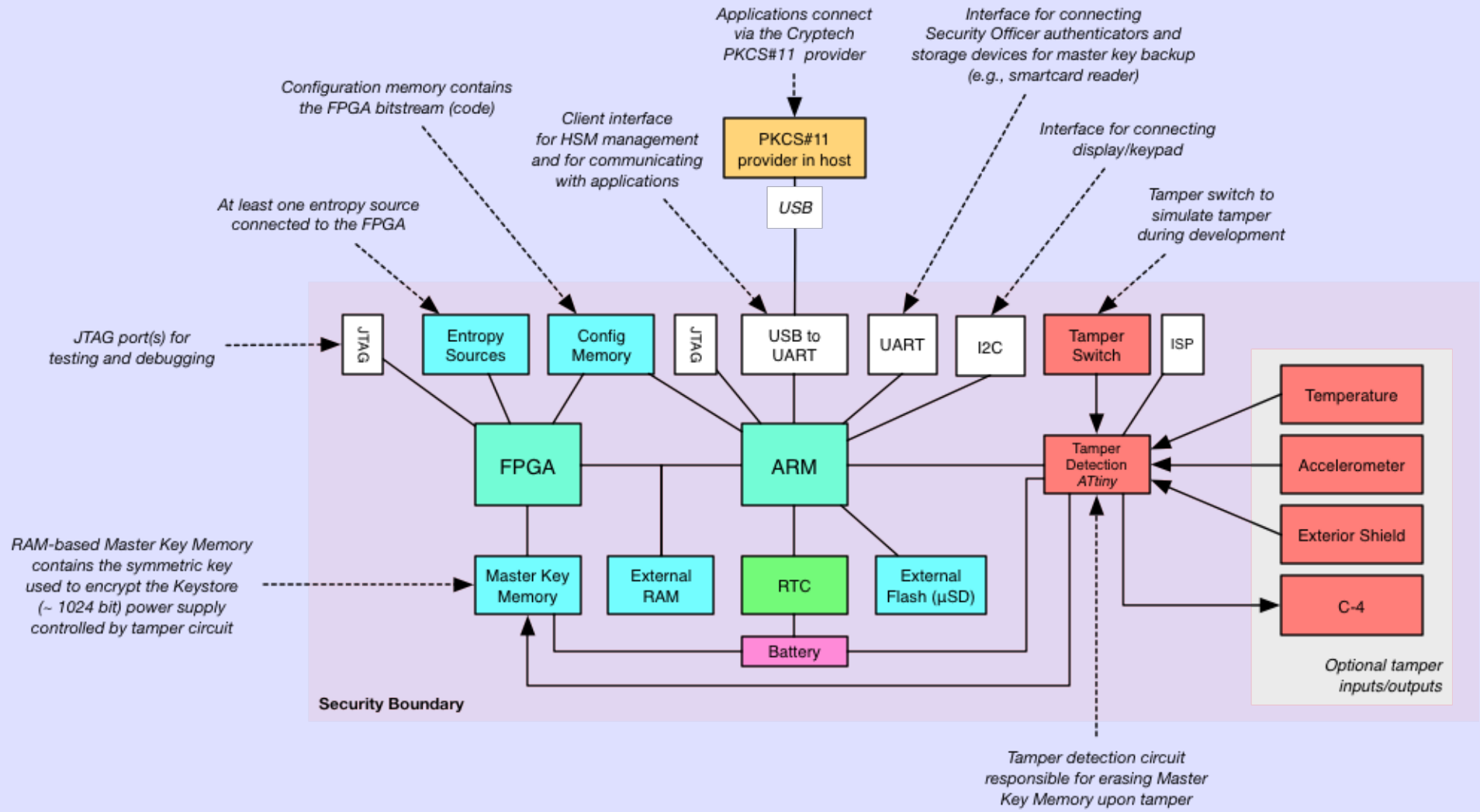
Novena Spartan 'Laptop'



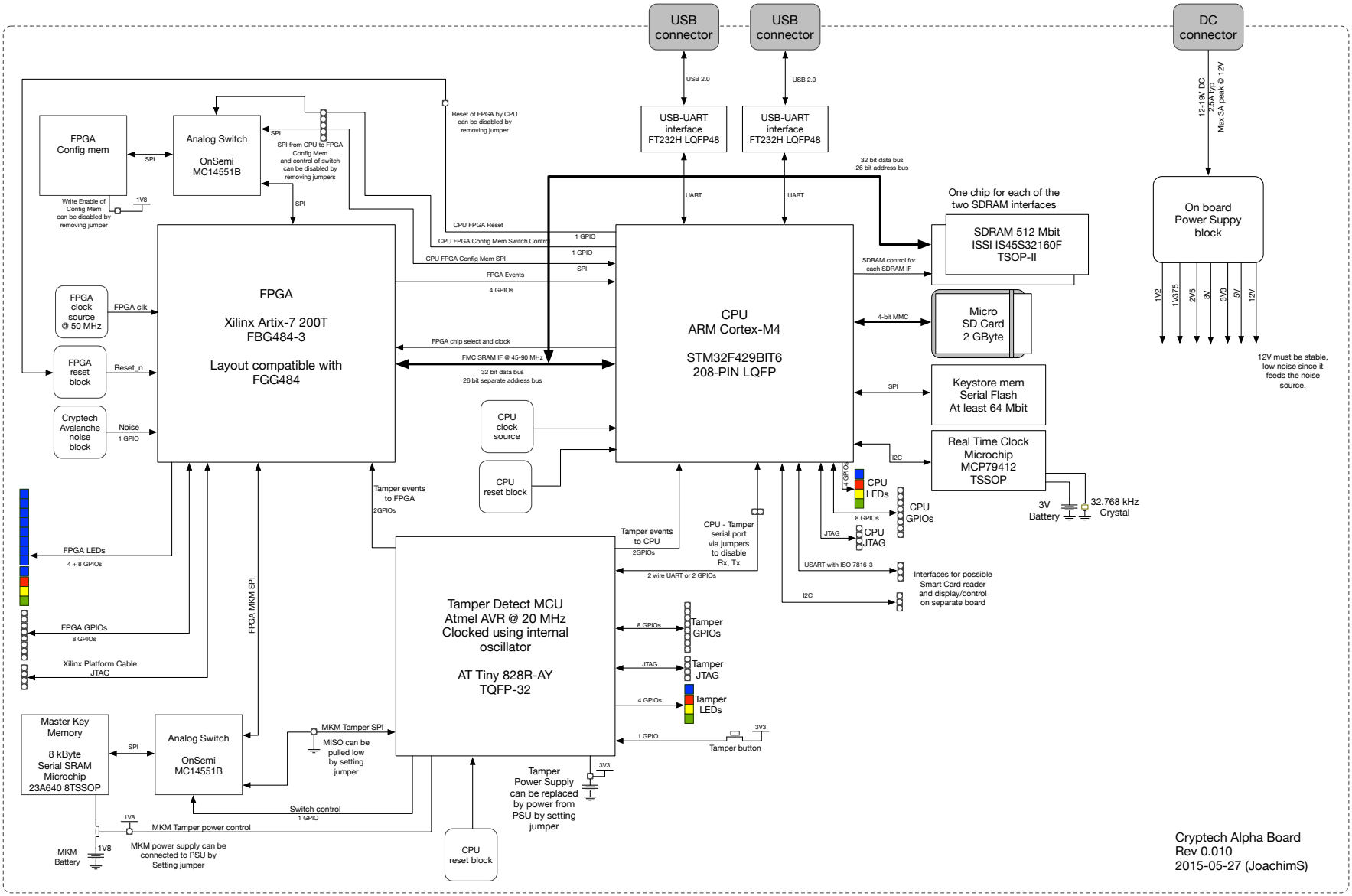
Entropy Noise Board



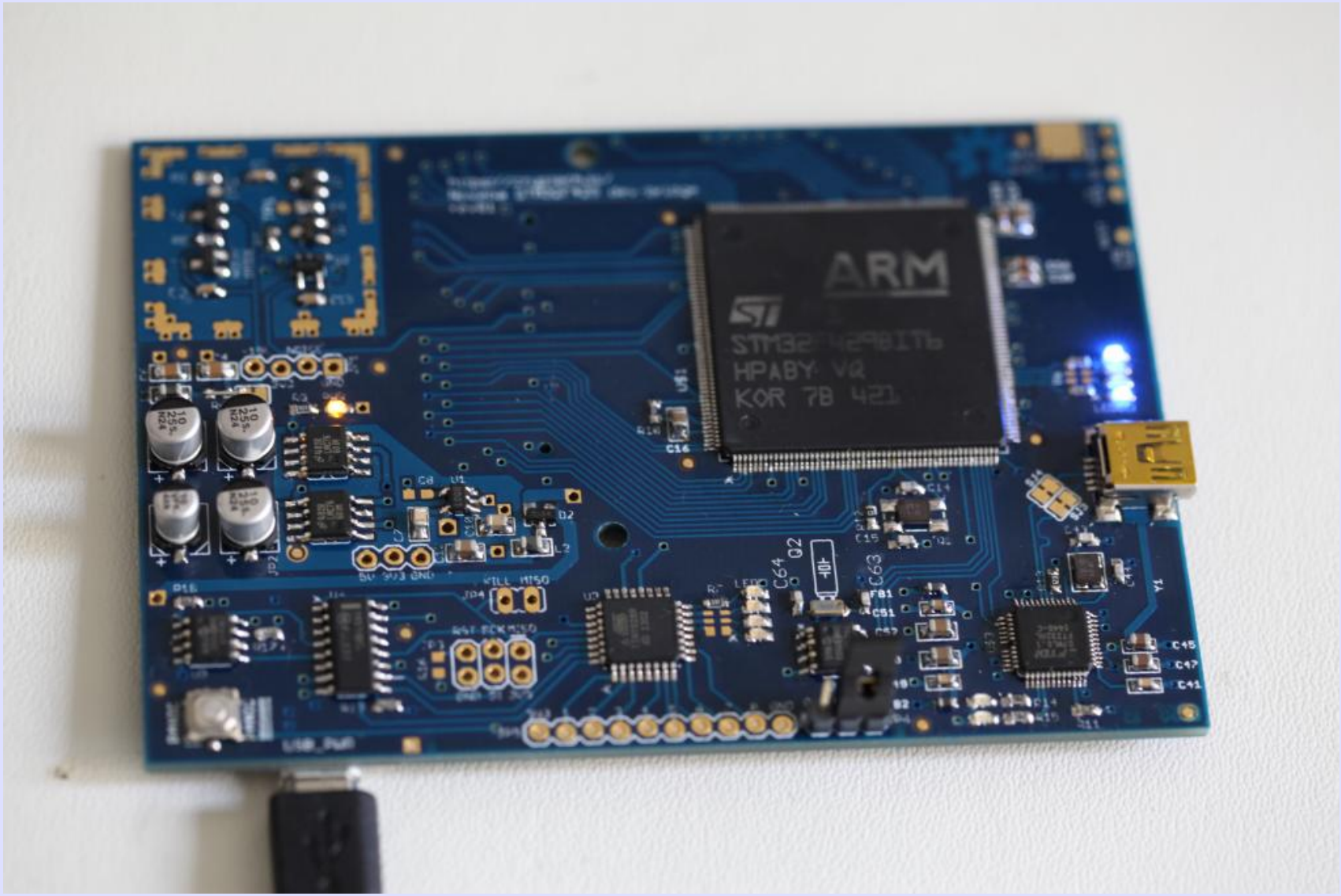
Alpha Board Blocks

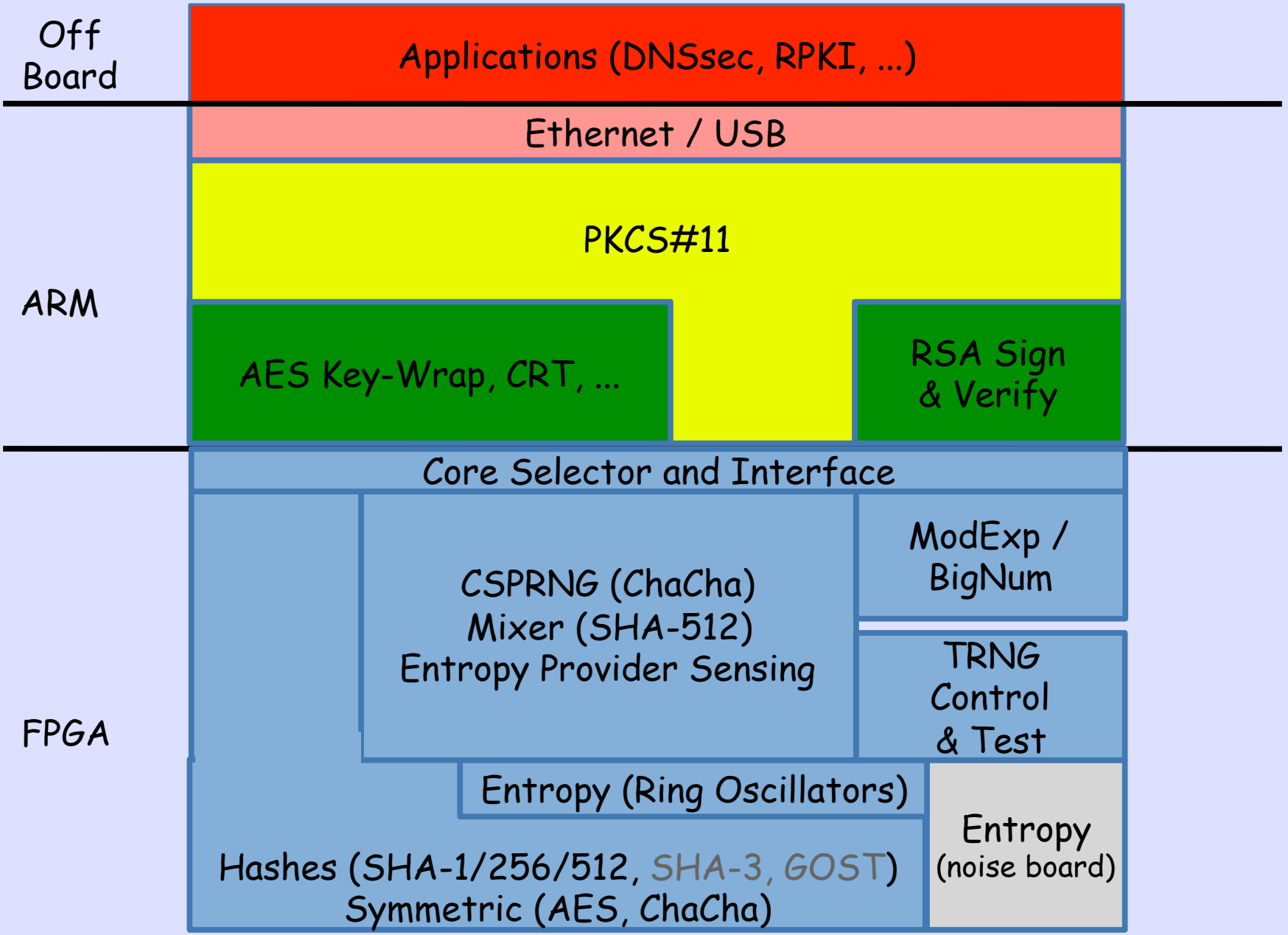


Alpha Board EOY 2015



Bridge Board

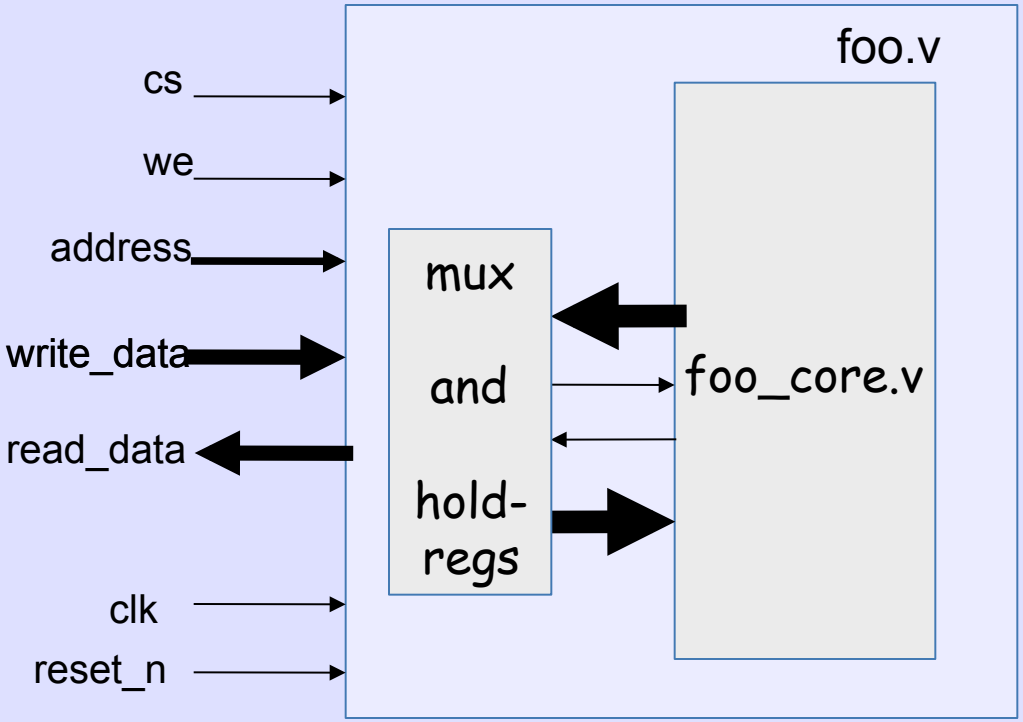




General Core Design

- Plain Verilog 2001 compliant RTL code
- FPGA vendor and FPGA/ASIC agnostic design
  - No explicitly instantiated technology specific macros
- All cores are independent co-processors
  - Cores do not share resources
  - Load data and configure, start core and wait for ready signal
- 32-bit memory like interface
  - Implemented by core wrapper
  - API structured similarly for all cores
- The real functionality is in _core.v and its sub modules

General Core Structure



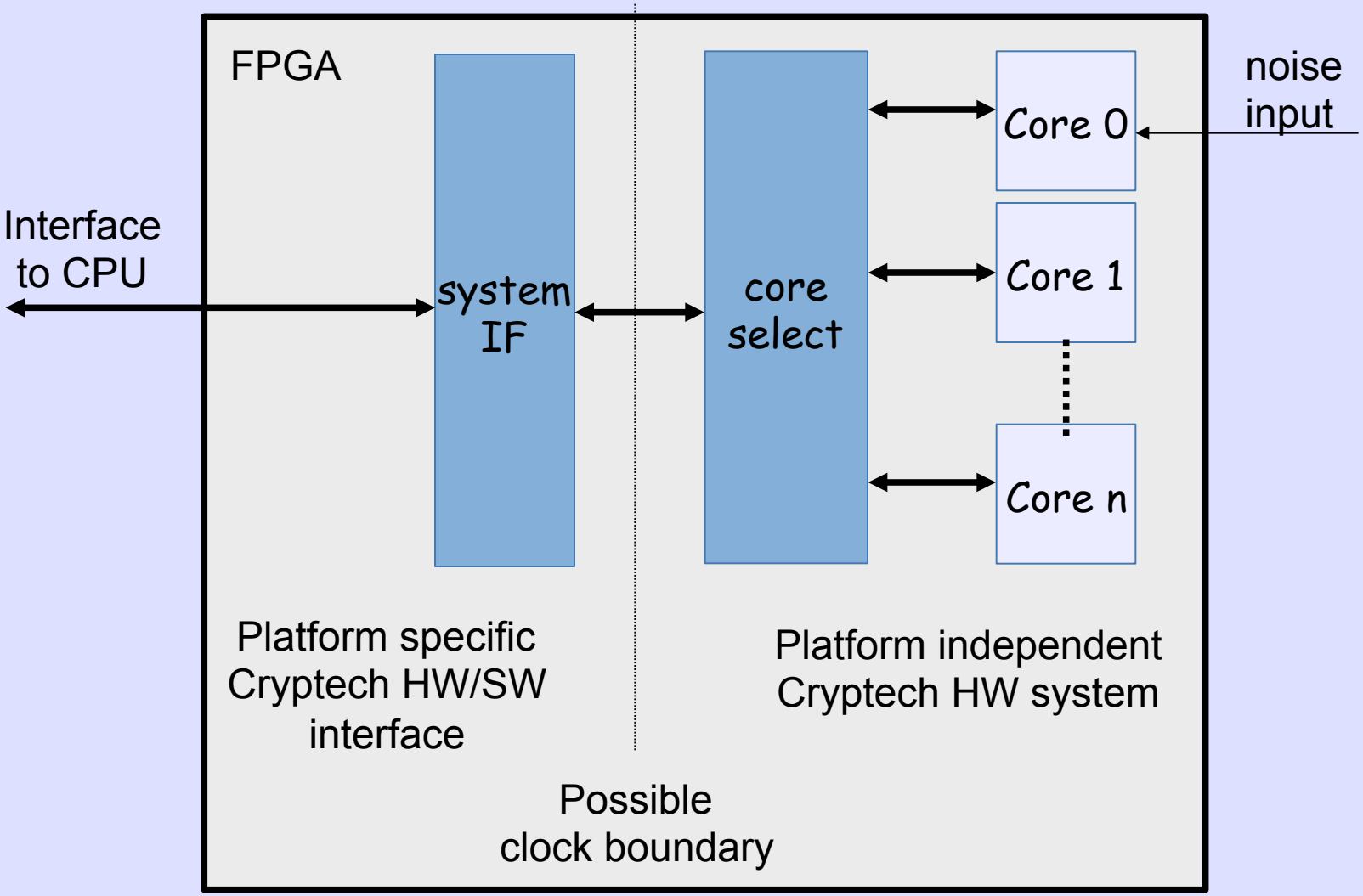
API Example

```
ADDR_NAME0        = 8'h00;
ADDR_NAME1        = 8'h01;
ADDR_VERSION      = 8'h02;

ADDR_CTRL         = 8'h08;
CTRL_INIT_BIT     = 0;
CTRL_NEXT_BIT     = 1;

ADDR_STATUS       = 8'h09;
STATUS_READY_BIT  = 0;
STATUS_VALID_BIT  = 1;

ADDR_BLOCK0       = 8'h10;
...
...
ADDR_BLOCK15      = 8'h1f;

ADDR_DIGEST0      = 8'h20;
...
...
ADDR_DIGEST7      = 8'h27;
```

Core Selector

- Current version hard coded for the use case
- Next version auto generated
  - Generate Verilog based on config
    - Instantiate types and number of instances of cores
  - SW support for discovery of cores in a given FPGA bitstream

Cryptech FPGA system



Core Walk Through

SHA1

- Implements SHA-1 as specified in FIPS 180-2
- Iterative, one cycle/round
  - 82 cycles/block with setup and finish
- Block expansion (W mem) implemented using sliding window with 16 separate 32-bit registers

SHA256

- Implements SHA-256 as specified in FIPS 180-4
- Iterative, one cycle/round
  - 66 cycles/block with setup and finish
- Block expansion (W mem) implemented using sliding window with 16 separate 32-bit registers

SHA512

- Implements SHA-512/x (FIPS 180-4)
  - Including SHA-512/224, SHA-512/256, SHA-512/384 and SHA-512
- Iterative, one cycle/round
  - 82 cycles/block with setup and finish
- Block expansion (W mem) implemented using sliding window with 16 separate 64-bit registers
- Support for work factor processing with up to 2**32-1 iterations/block
- Testbenches for w_mem, core and top level
  - Using NIST test vectors
- Heavily tested with SW on the Novena
- Used in Cryptech as mixer in the TRNG

AES (1)

- As specified in FIPS 197
  - Support for 128 and 256 bit keys
- Iterative, four cycles/round
  - 42 cycles/block with setup and finish for AES-128
  - 58 cycles/block with setup and finish for AES-256

AES (2)

- Key expansion performed before any block processing
  - 10 cycles for 128 bit keys, 14 cycles for 256 bit keys
- Separate encipher and decipher data paths
  - Decipher can be removed for use cases where only encipher is needed (CTR mode etc)
  - Encipher and decipher share key expansion

AES (3)

- Four sbox ROMs
  - Shared between encipher data path and key expansion
- Testbenches for key expansion, data paths, core and and top level
  - Using NIST test vectors and vectors by Sam Trenholme (http://www.samiam.org/key-schedule.html)

AES (4)

- Heavily tested with SW on the Novena
- Used in Cryptech to implement AES Key Wrap (RFC 5649, https://tools.ietf.org/html/rfc5649)

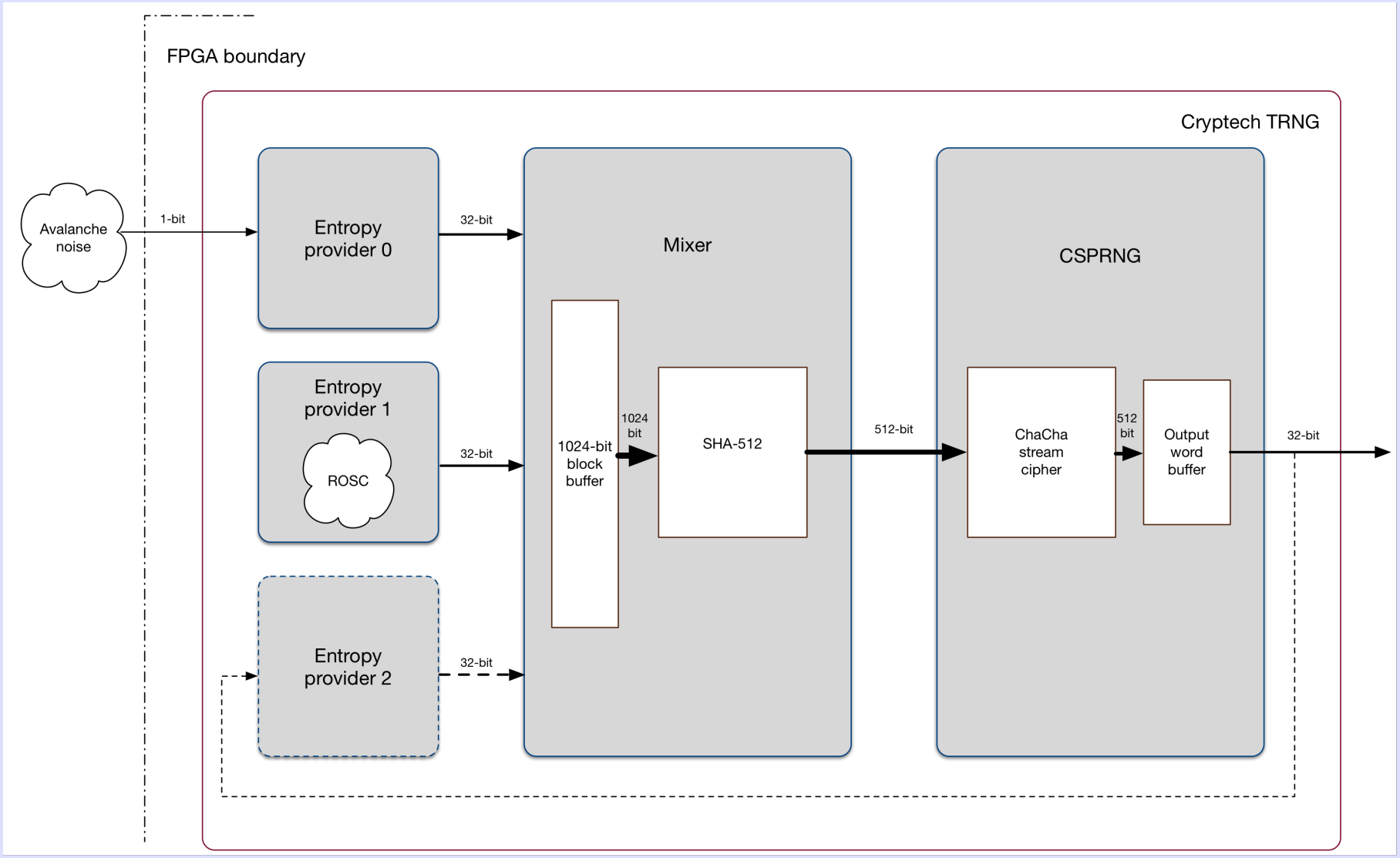
ChaCha (1)

- Implements the ChaCha stream cipher
  - http://cr.yp.to/chacha/chacha-20080128.pdf
  - Support for 128 and 256 bit keys
  - Support for up to 32 rounds
  - Support for settable 64-bit initial counter value
- Iterative, two cycles/double round
  - 42 cycles/block with setup and finish for ChaCha20

ChaCha (2)

- Testbenches for core and top level
  - Using DJB test vectors and generated test vectors for draft https://tools.ietf.org/html/draft-strombergson-chacha-test-vectors-00
- Used in Cryptech as CSPRNG in the TRNG
  - With 256 bit key and 24 rounds
  - Key, block, IV and initial counter as seed

TRNG



TRNG (1)

- Sub system using multiple cores
  - avalanche noise entropy provider core
  - ring oscillator entropy provider core
  - SHA512 core used as entropy mixer
  - ChaCha core used as CSPRNG

TRNG (2)

- Modular architecture
  - Support for adding more entropy sources
  - Support for replacing SHA512 in mixer and ChaCha in CSPRNG with other cores
- Support for observability and testing and of all parts and output
  - Extract raw noise and entropy from the sources
  - Inject test vectors and extract results to allow verification of functionality
  - Planned support for on-line testing and alarms for entropy sources and CSPRNG

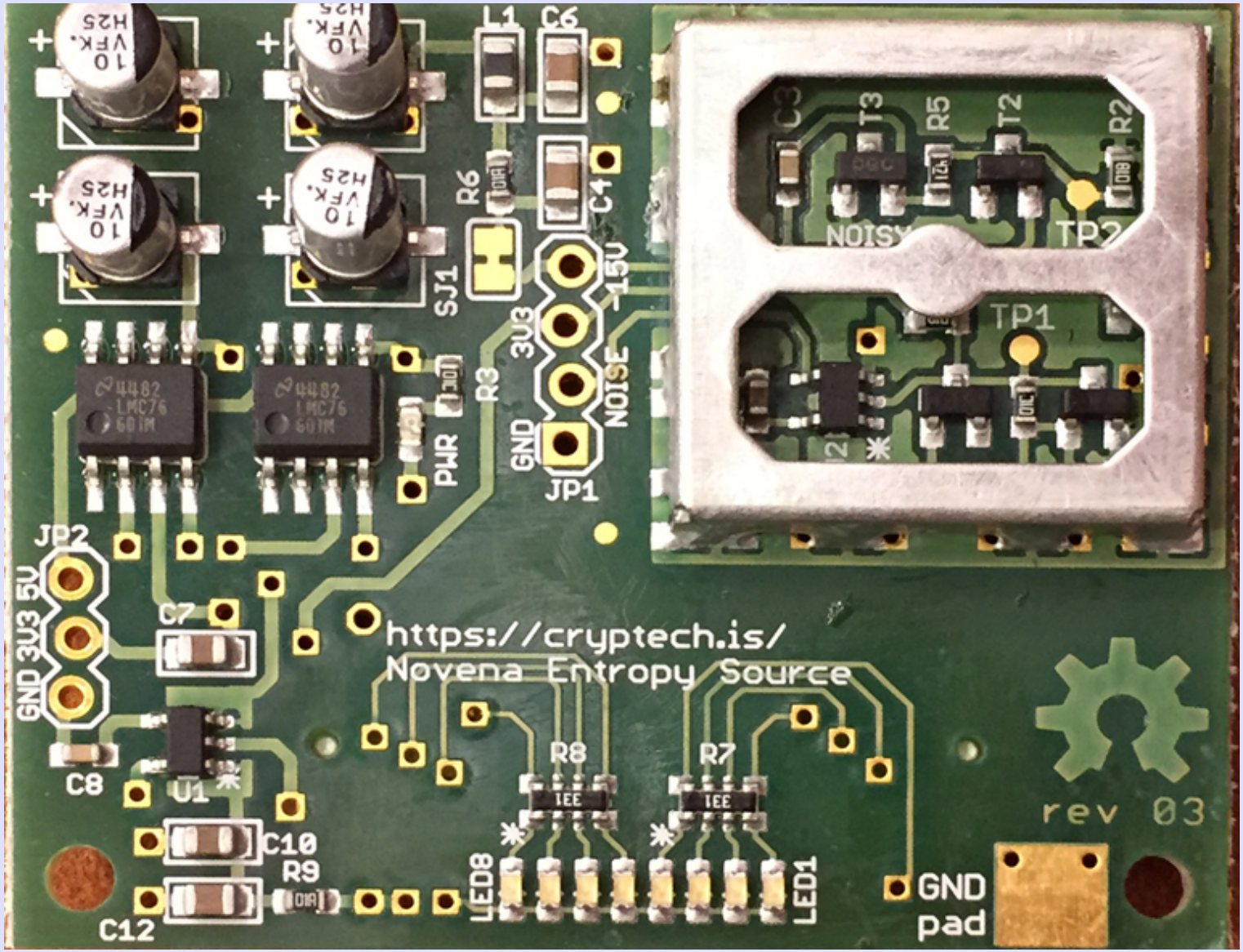
TRNG (3)

- Scalable performance and security
  - Number of rounds (default 24) can be configured via API
  - Reseed frequency settable and can be forced via API
  - Can generate ~500 Mbps @50 MHz clock frequency
  - Can instantiate multiple ChaCha cores (seeded separately) to scale performance to multiple Gbps performance

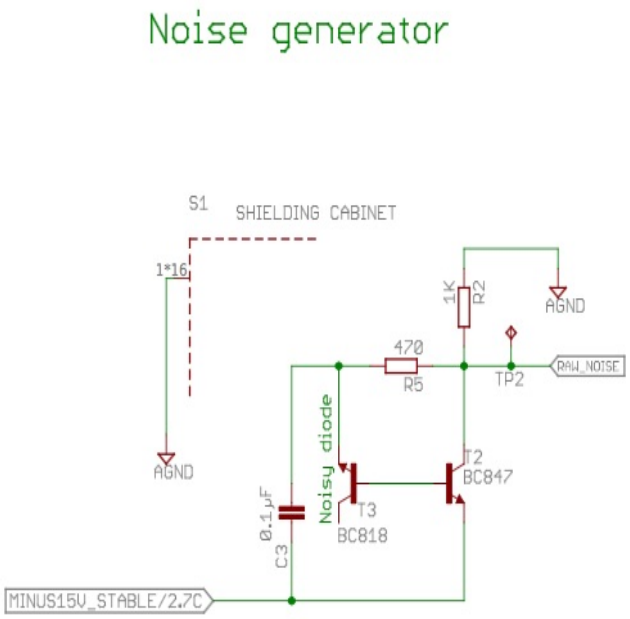
TRNG (4)

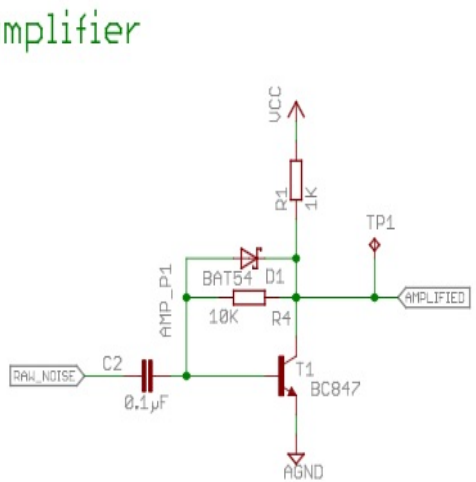
- Tested using ent, diehard, dieharder and several custom tools
  - TBytes of data generated and tested so far
  - Test server that provides public access to continiously generated data being setup

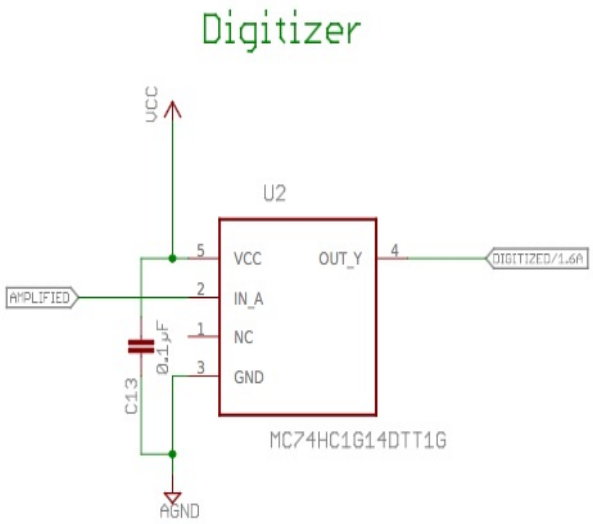
Avalanche Noise Board



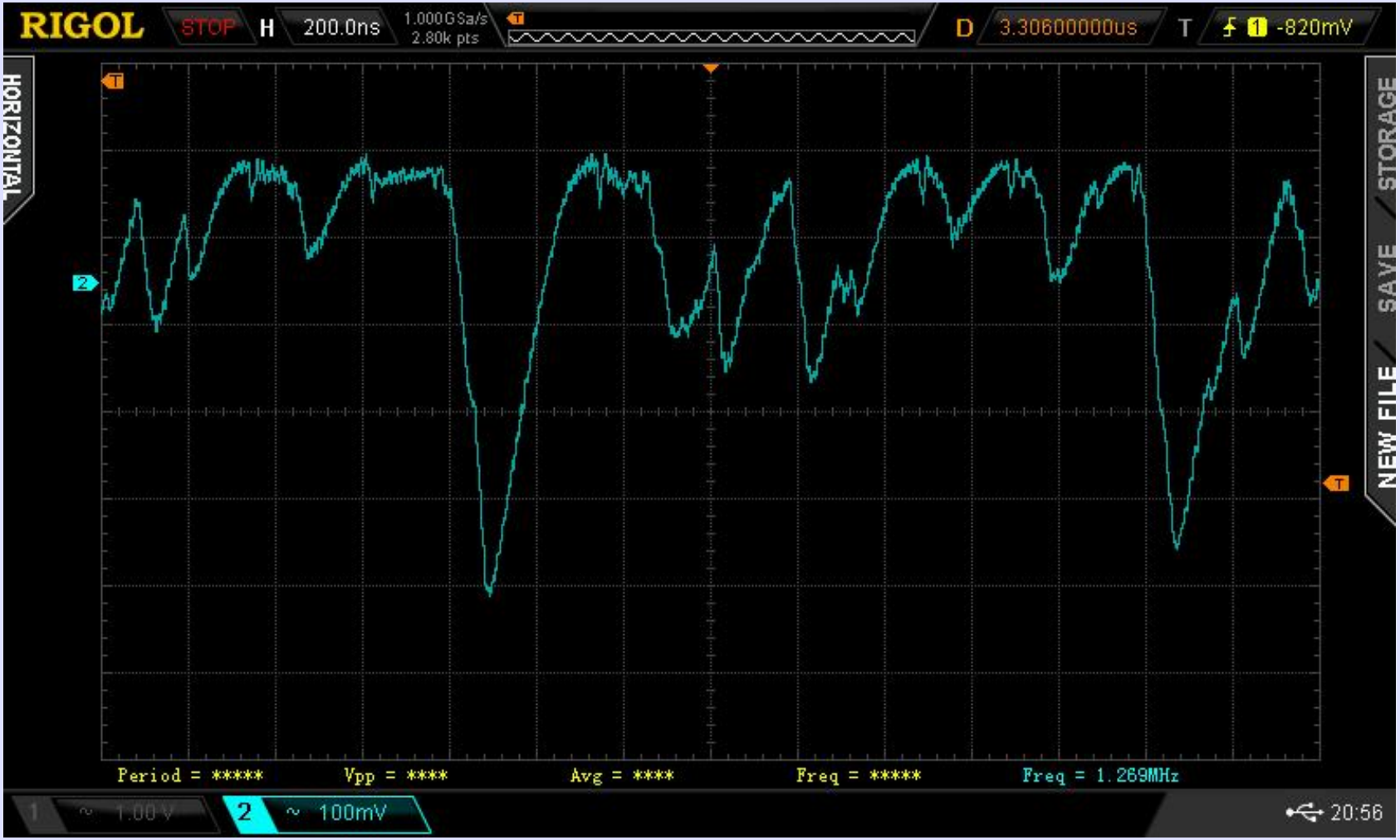
Noise Generation





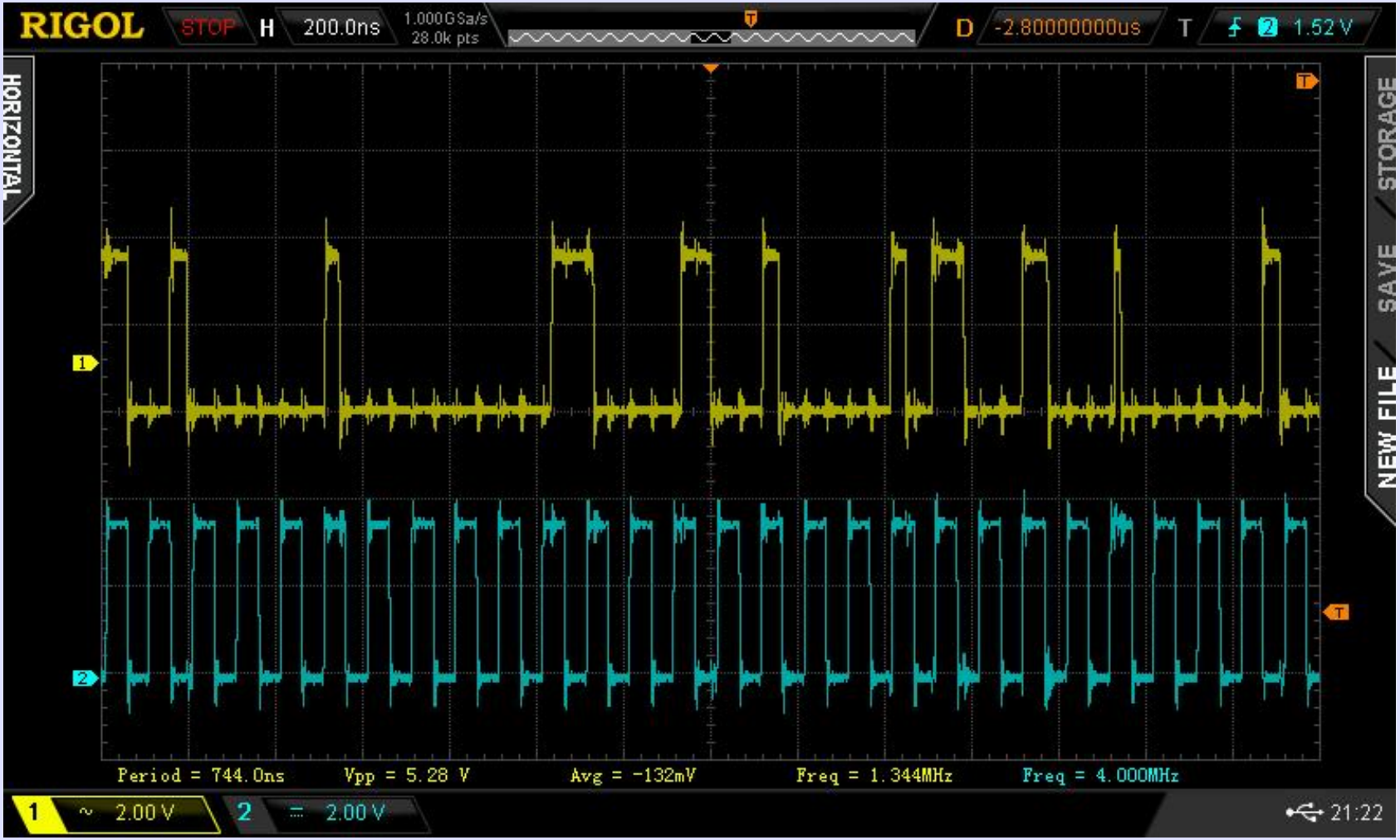


Raw noise



Amplified (yellow)

Digitized (yellow)



Twitterized explanation

- To combat component ageing, measure time between flanks, use LSB of time delta as entropy. Do whitening.

Avalanche Entropy (1)

- Entropy provider using external noise source
  - Used with the Cryptech Avalanche noise source
  - Noise digitized by board using a schmitt-trigger and provided as single bit stream

Avalanche Entropy (2)

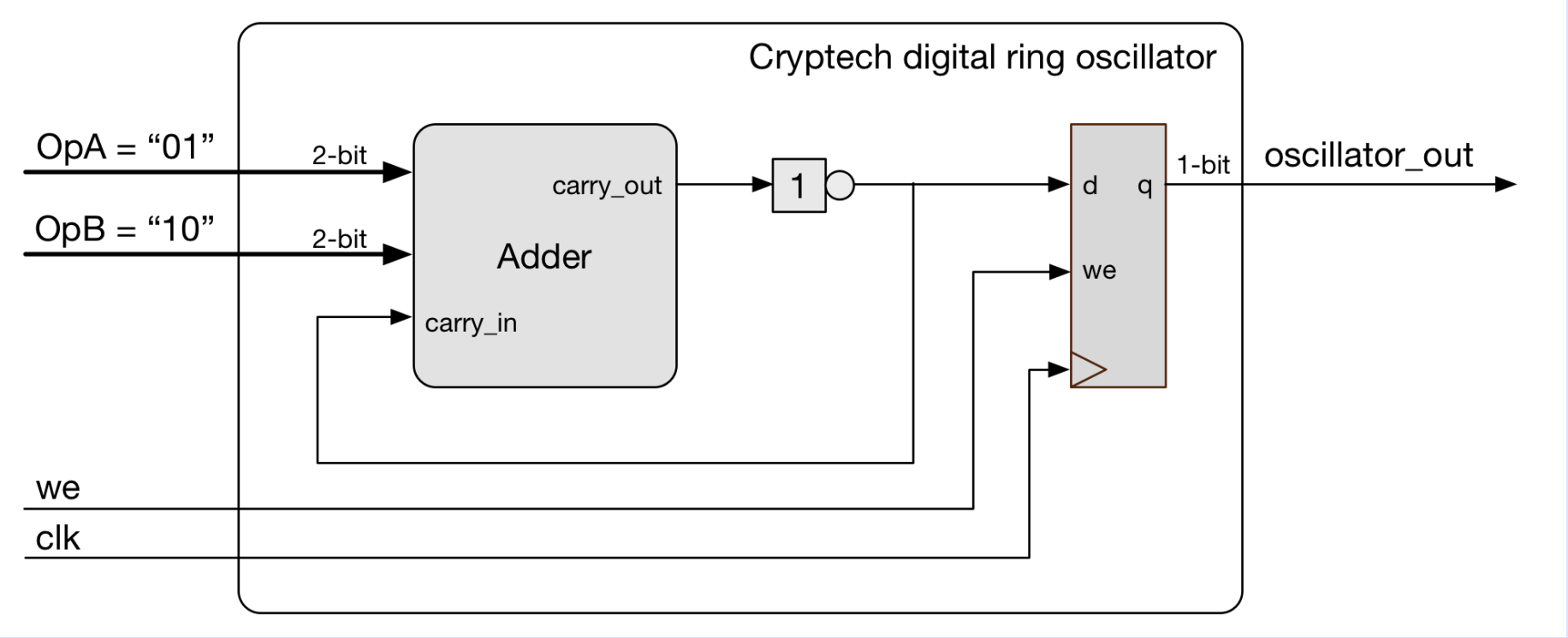
- Measures time (cycles) between positive flanks on noise source
  - LSB from cycle counter used as entropy bit
  - 32 consecutive bits provided as entropy data to consumer (mixer)

Avalanche Entropy (3)

- Heavily tested using ent, several custom tools
  - Good confidence that the entropy provided has good quality
  - Long term stability needs to be evaluated (and being worked on)

- ~10 kbps data rate

Adder based Ring Oscillator



ROSC Entropy (1)

- Entropy provider using internal jitter source
  - Using a novel adder based ring oscillator (ROSC) suitable for FPGA implementation.
  - Designed by Bernd Paysan
  - ~2 kbps data rate

ROSC Entropy (2)

- Generates entropy using jitter between ring oscillators
  - Uses 32 separate ring oscillators (running at 300+ MHz in Spartan-6)
  - Samples the output values from the oscillators every 256 clock cycles
  - The outputs from the oscillators are XOR combined into a single bit value
  - 32 consequtive bits provided as entropy data to consumer (mixer)

ROSC Entropy (3)

- Heavily tested using ent, several custom tools
  - Fairly good confidence that the entropy provides sufficient quality
  - ROSC feedback path routing critical to clock frequency. Should preferrably be locked down using Place & Route constraints
  - rosc_entropy core should be requalified when moved to a new technology (for example a new FPGA family)

Mixer (1)

- Combines entropy from providers to create seeds for the CSPRNG
  - Strict round robin extraction from a set of entropy providers
- Decouples the entropy collection from the random number generation by the CSPRNG

Mixer (2)

- Make it hard to predict seed when trying to control an entropy source

- Make it hard (infeasible) to guess mixer state and entropy state based on guess of bits in seed

Mixer (3)

- Seeds are intermediate digests for an arbitrarily long message
  - Unless full restart is forced
- With SHA-512 as mixer primitive, 1024 bits of entropy is needed to generate 512 bits of seed
  - With the current Cryptech CSPRNG, two 512-bit seed words are needed. In total 2048 bits of entropy is needed to be able to reseed the CSPRNG

CSPRNG

- Using the ChaCha stream cipher as primitive
  - 24 rounds by default

896 bits in total

- Cipher initialized by
  - 256 bit key
  - 512 bit *message block*
  - 64 bit IV
  - 64 bit initial counter value

CSPRNG (2)

- Blocks of 512 bits of stream data extracted via a FIFO as 32-bit random words by consumers
- Decouples data generation from consumption

Funding From

A Few Private Donations