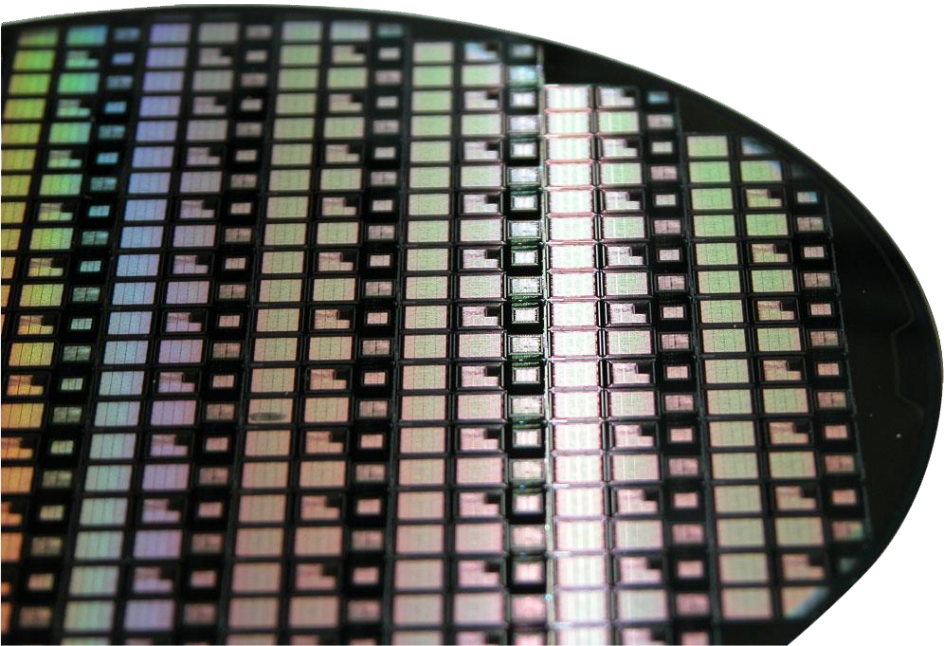
ETH zürich

Accelerating Function Kernels for Elliptic Curve Operations and Mobile Communication Algorithms

Tensilica Day, Hannover

12.02.2016



Michael Gautschi

Prof. Luca Benini

ETH zürich

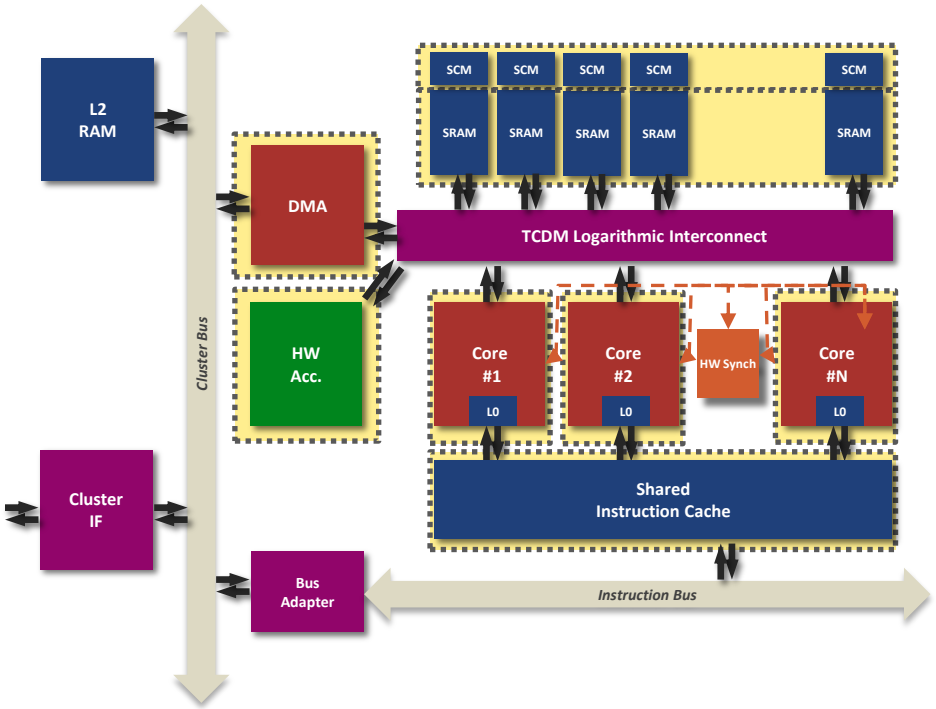
Integrated Systems Laboratory

Our group: Prof. Luca Benini

- ETH Zurich, Integrated Systems Lab (IIS)
  - Digital Circuits and Systems
- Around 40 people
  - IC designers
  - Software developers
- Close collaborations
  - Politecnico di Milano
  - CEA-LETI
  - EPFL
- Industrial support from STMicroelectronics
  - Silicon access to 28nm FDSOI

Our research:
PULP: Parallel Ultra-Low-Power Processor

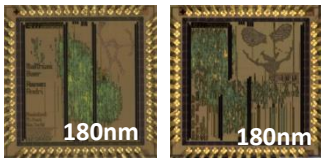
- Exploit parallelism
  - Multiple small cores organized in clusters
  - Share memory within the cluster
  - Multiple clusters per chip
- Simple but efficient processor cores
  - Based on OpenRISC/RISC-V
  - Custom ISA extensions
- Dedicated accelerators
- Multiple Technologies
  - Near-threshold operation

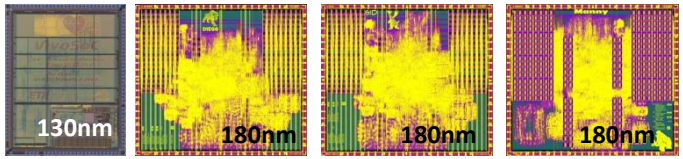


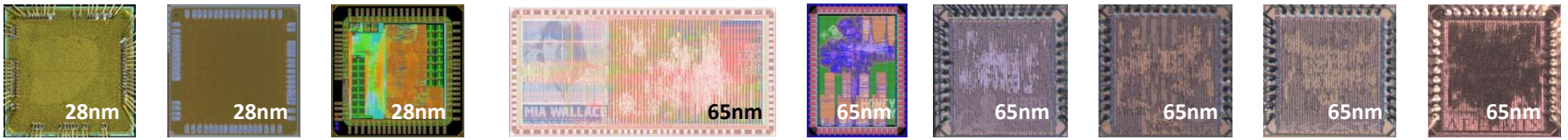
PULP
Parallel Ultra Low Power

PULP related chips

- **Main PULP chips** (ST 28nm FDSOI)
  - PULPv1
  - PULPv2
  - PULPv3 (in production)
  - PULPv4 (in progress)
- **PULP development** (UMC 65nm)
  - Artemis - IEEE 754 FPU
  - Hecate - Shared FPU
  - Selene - Logarithmic Number System FPU
  - Diana - Approximate FPU
  - Mia Wallace – full system
  - Imperio - PULPino chip (Jan 2016)
  - Fulmine – Secure cluster (Jan 2016)
- **RISC-V based systems** (GF 28nm)
  - Honey Bunny
- **Early building blocks** (UMC180)
  - Sir10us
  - Or10n
- **Mixed-signal systems** (SMIC 130nm)
  - VivoSoC
  - EdgeSoC (in planning)
- **IcySoC chips approx. computing platforms** (ALP 180nm)
  - Diego
  - Manny
  - Sid





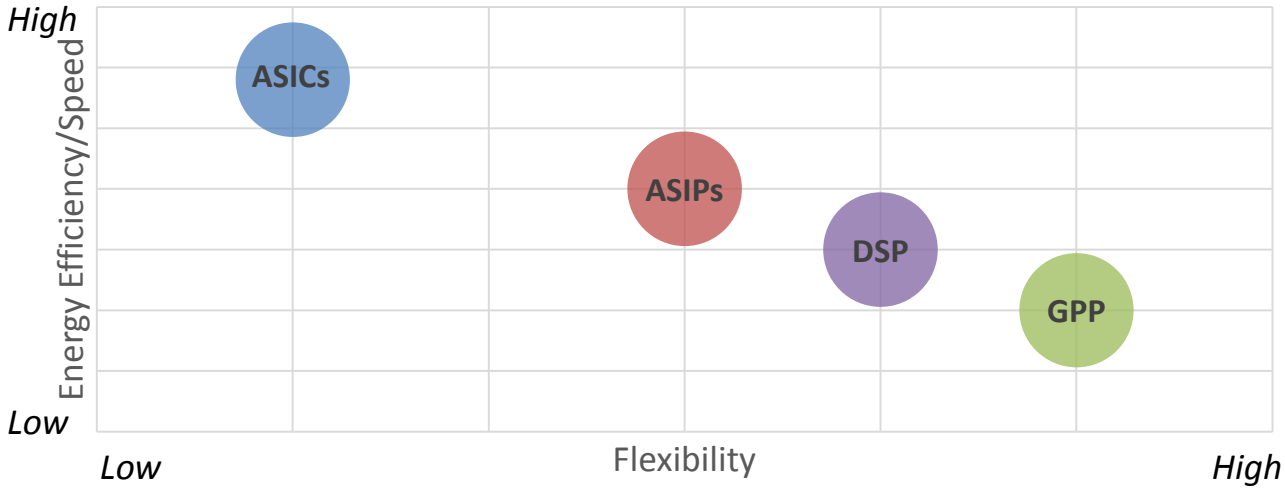


Outline

- Introduction
- Case Study 1: Digital Signature Verification
- Case Study 2: Turbo Decoding in Mobile Communication
- Case Study 3: Random Number Generator
- Conclusion

Application Specific Instruction Set Processors (ASIPs)

Performance – Flexibility Trade-off

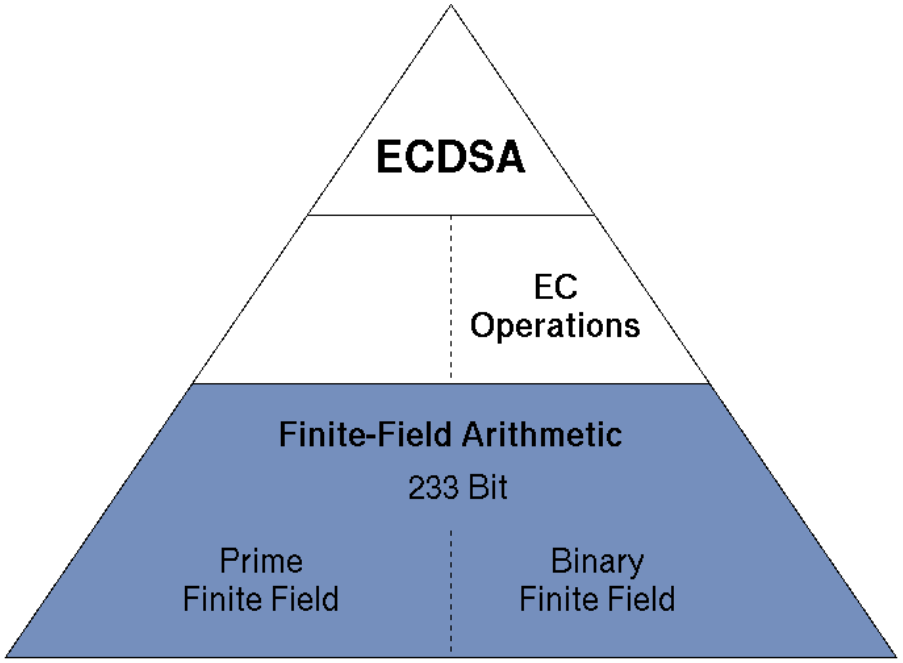


- Xtensa Core Generator allows to build a specific Processor Core

- Used in our Lecture “Advanced System-on-Chip-Design”
  - TIE instructions can be used to speedup applications
  - Students get the opportunity to work with Xtensa Core Generator

Cryptographic Example: ECDSA signature verification

- **E**lliptic **C**urve **D**igital **S**ignature **A**lgorithm (ECDSA)
  - Signature validation with **E**lliptic **C**urves (EC)
- ECDSA requires prime and binary finite field operations
- Add, sub, mult, div, etc. defined in the finite-field arithmetic
  - We used the NIST B-233 curve



- Flexibility allows to support different algorithms and different standards with one hardware!
- Finite field operations are not suitable for general purpose architectures
  => hardware much more efficient!

Cryptographic Example: Base processor configuration

- Processor configuration:
  - 5 stage integer pipeline
  - 32 32bit general purpose registers
  - 16x16 bit multiplier
  - 2 KB I$, D$

- Estimated performance (65nm LP technology):
  - Max speed: 344 MHz (worst case)
  - Area: 83 kGE
  - Power : 20 mW

- ECDSA algorithm ported from a former semester project [1]

[1] Semester Thesis by A. Traber, S. Stucki, 2014, A Unified Multiplier Based Hardware Architecture for Elliptic Curve Cryptography

ECDSA Signature Verification Algorithm

Profiling results using Xtensa Xplorer:

Operation	Total (cycles)	Total (%)	# function calls	Code size (bytes)
ECDSA Verification	46'674'997			6518
$GF(2^{233})$ multiplication	42'063'643	90.1%	2'309	366
16x16 bit binary f. field mult.	34'199'189	73.3%	591'104	157
$GF(2^{233})$ squaring	2'706'852	5.7%	2'472	446
others	1'904'502	21.0%	-	4'930

- Simple squaring operation on a 233 bit binary field:
  - Insert '0' bit between each input bit
    - e.g. '1101' => '01010001'
  - Reduce resulting 466 bit number to 233 bit
- Requires masking and shifting in C -> not efficient at all!
- Hardware architectures can do such operations in one cycle!
  => add custom instructions for 16bit mult, and squaring

Optimizing the multiplication in GF(2^{233})

Original C code

```
// shortened C-code: binary field 16 bit
// multiplication

uint16_t a, b;       // input
uint32_t prod = 0;   // output

for(i = 0; i < 16; i++) {
  if((a&(1<<i))
      prod ^= b << i;
}
```

~60 cycles

New C code

```
// intrinsic function call

prod = BinMul16(a,b);

// a, b, prod are stored
```

Speedup of factor 60!

"TIE" – Tensilica Instruction Extensions

```
// shortened BinMul operation
operation BinMul16 {out AR res, in AR a, in AR b}{}{
   wire [31:0] temp0 = (a & (1 << 0)) ? (b << 0)            : 32'b0
   wire [31:0] temp1 = (a & (1 << 1)) ? (temp0^(b << 1))    : temp0;
   wire [31:0] temp2 = (a & (1 << 2)) ? (temp1^(b << 2))    : temp1;
   // …
   wire [31:0] temp15 = (a & (1 << 15)) ? (temp14^(b << 15)) : temp14;
   assign res = temp15;
}
```

1 cycle only!

Results: Performance

- Profiling results of specialized circuit
- Area requirements: 2.2 kGE

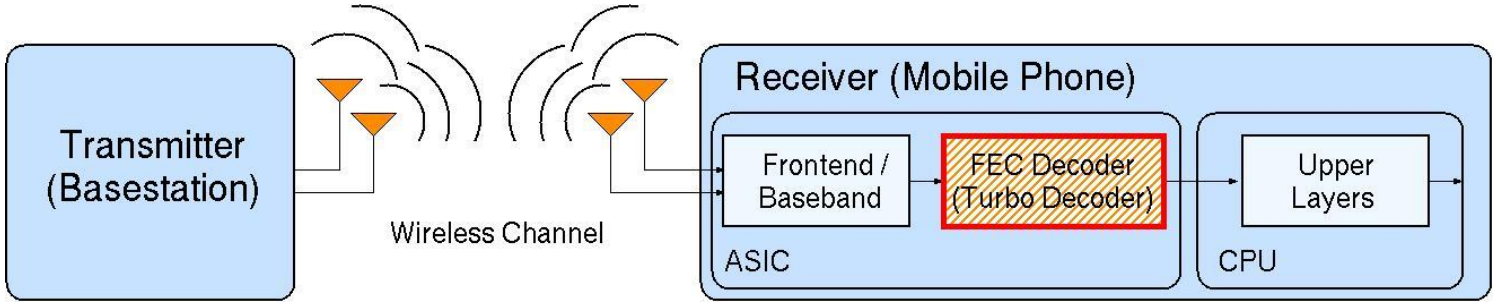
Operation	Total (cycles)	# cycles before	Speedup
ECDSA Verification	**6'181'171**	46'674'997	**7.5x**
$GF(2^{233})$ multiplication	**4'163'127**	42'063'643	**10.1x**
16x16 bit binary f. field multiplication	**591'104**	34'199'189	**57.9x**
$GF(2^{233})$ squaring	**177'998**	2'706'852	**15.2x**
others	**1'840'046**	1'904'502	

- Comparison to HW-architecture:[2]
  - Coprocessor with 16 bit datapath requires ~1'850'000 cycles (12 kGE)
  - Factor 3.3 slower

[2] M. Gautschi, M. Mühlberghuber et.al. , SIRIOUS: A tightly coupled ECC Coprocessor for the OpenRISC

ASIP Implementation of a BCJR Decoder

Mobile Communications – System Model

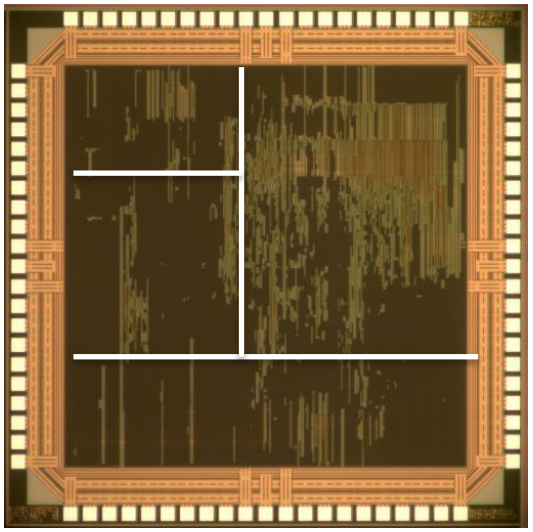


- Decoder for forward error correcting codes one of the main bottlenecks for the receiver
- Typically: processor for higher layers, ASIC for decoder and baseband processing
- Throughput, area and power consumption critical
- Convolutional codes (CCs) and concatenated CCs used in mobile communications

- This project: decoder for a convolutional code (BCJR algorithm)

ASIC vs. Processor

- State of the art BCJR ASIC [3]
  - **1 bit/cycle**
  - **40kGE**
  - **500Mbps in 180nm CMOS**
  - **Inflexible, difficult to change**



BCJR Decoder ASIC:
4,8,16,32 states [3]

- Processor
  - **1e-4 bit/cycle**
  - **100kGE**
  - **Very flexible, multi-standard BCJR**
  - **Can be used for other tasks as well (rate matching, higher OSI layers)**

[3] Studer et al., "Implementation Trade-Offs of Soft-Input Soft-Output MAP Decoders for Convolutional Codes", 2012

ASIP: The best of both worlds?

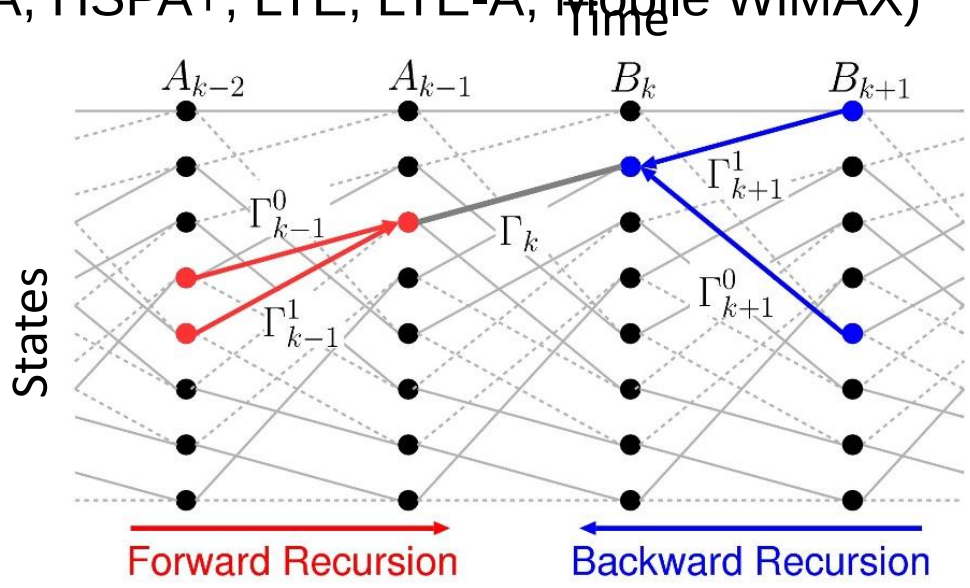
- High flexibility for decoder required:
  - Wide range of code rates, channel conditions, coding schemes, block sizes, …
  - Devices have to support different standards

- CPU advantageous for other blocks
  - Adjacent blocks (rate matching, HARQ memories) better suited for software
  - Processor required for the protocol stack
  - Easier implementation of multi-mode, multi-standard decoders

- Still high performance though custom extensions
  - Is it possible to achieve good performance without a co-processor?

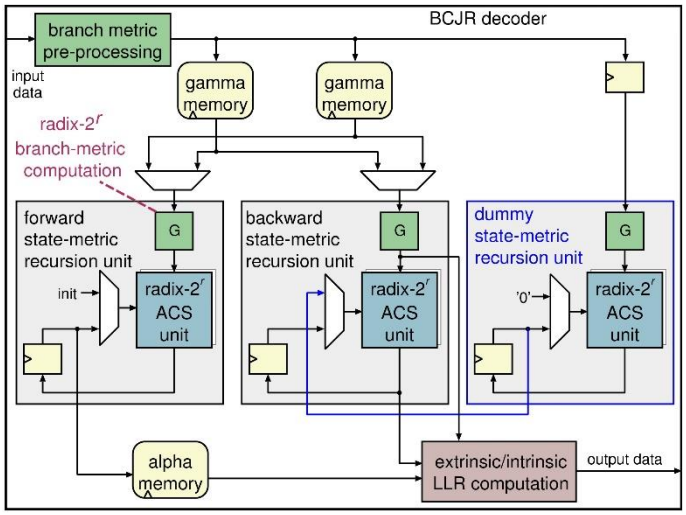
BCJR Algorithm

- Decoder algorithm for convolutional codes by Bahl, Cocke, Jelinek, Raviv
- Used as inner decoders in almost all mobile communications standards
  (e.g. EEDGE, UMTS, HSPA, HSPA+, LTE, LTE-A, Mobile WiMAX)
- Trellis-based algorithm
- Algorithm: uses a forward and a backward recursion
- **Goal: find the most likely sequence of transmitted symbols**

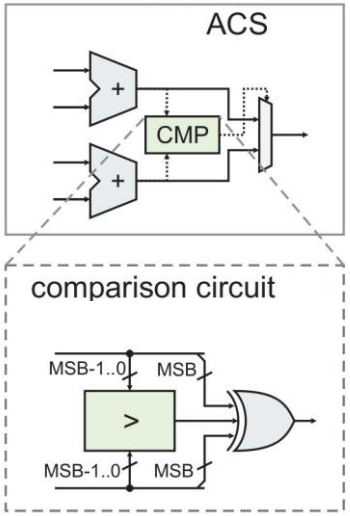


Trellis diagram: forward metrics A, backward metrics B and branch metrics Γ

ASIP Implementation – ACS Extension



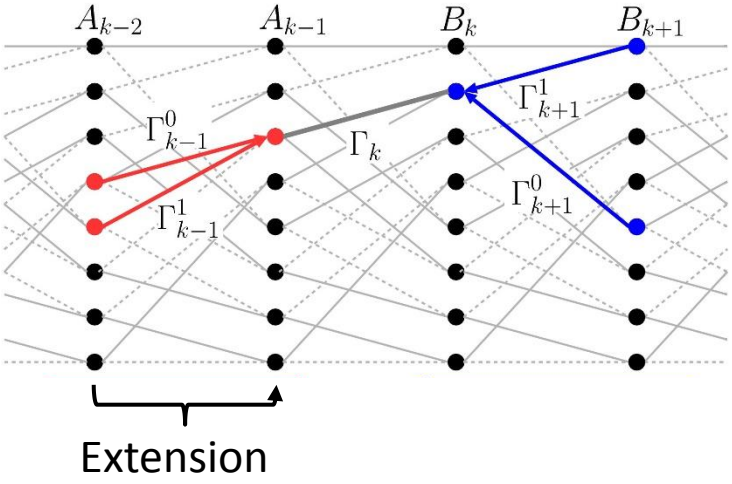
BCJR Architecture

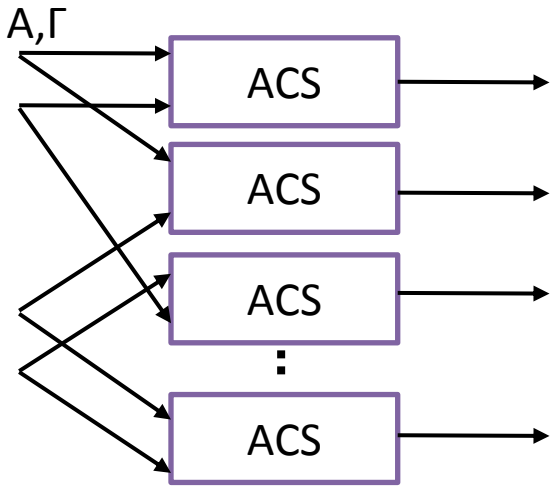


ACS Architecture

- The critical path of a fully pipelined BCJR decoder lies within an add-compare-select (ACS) unit
- No multipliers or other complex circuits needed
- First try: use an extension for the ACS unit
- Reduction minimal, only 5-10% reduction

ASIP Implementation – Alpha Unit





Extension: HW
8 ACS units

- Idea: calculate one complete trellis step in an extension
  - 8+8: inputs (previous alpha and branch metrics)
  - 8 outputs (new alpha metrics)
- Number of cycles reduced by 94%
  - Current bottleneck: memory access and address calculation
- Area overhead of ~8kGE (processor ~100kGE)

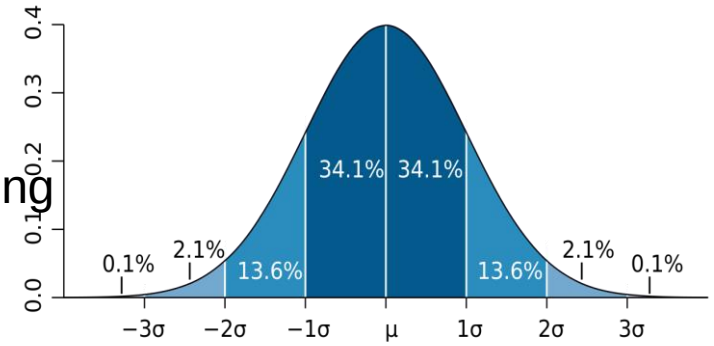
Results

- The number of cycles/bit for a BCJR implementation can be significantly reduced (by ~90%) by adding custom extensions, at a moderate area overhead of 8kGE.
- The resulting ASIP is still 1000x slower than a highly optimized ASIC implementation
- The ASIP solution might be interesting for low-throughput standards (UMTS, EDGE)

- Possible Future Optimizations
  - Quantization (ASIC: 5 to 10bit, ASIP: 32bit)
  - Different memory organization with quantization

Fast Gaussian Random Number Generation

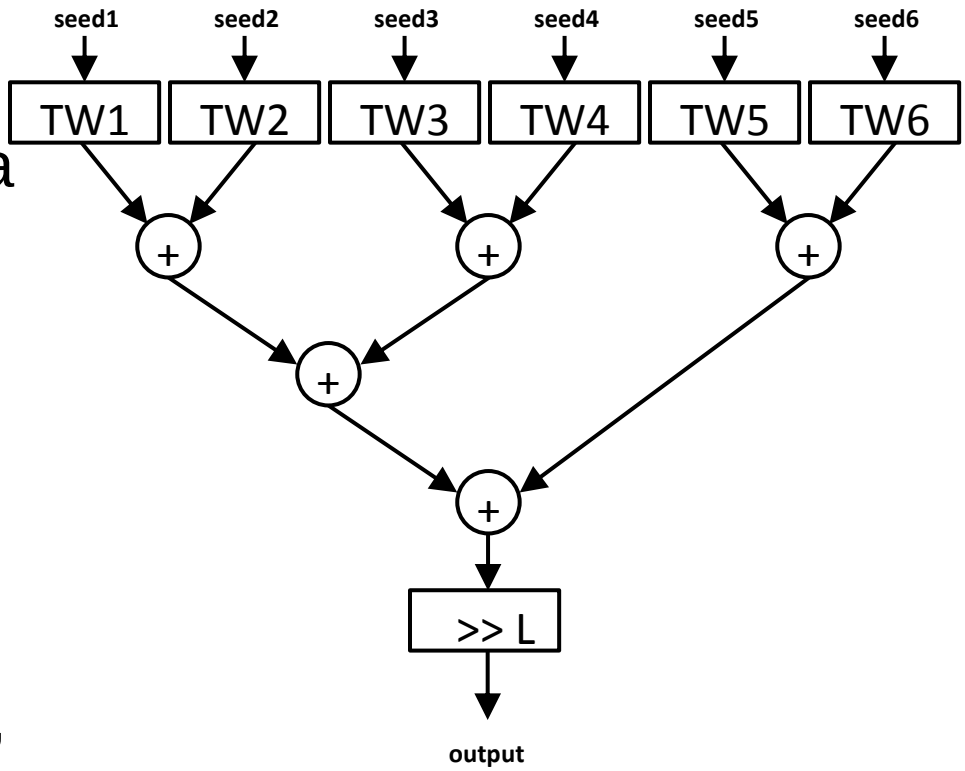
- Gaussian random numbers are used in many fields:
  - Noise generators for communication channels
  - Particle filters
  - Monte Carlo simulations
    - in our case finance
    - Hardware accelerated derivative pricing



- Digital GRNG in a nutshell:
  - Until the late 1900s, mostly software methods are used, based on transformations of Uniformly distributed random numbers (URNs)
  - Todays HW implementations are inherited from their software predecessors
    - e.g. Box muller method (used in Xilinx IP core & CUDA)
    - But they come along with complex FSMs, loops, huge LUTs and thus often result in complex HW designs
  - Recently (2014 IEEE Trans. on VLSI), the Central Limit Theorem (CLT) approach was rediscovered → **This is what is investigated in this project**

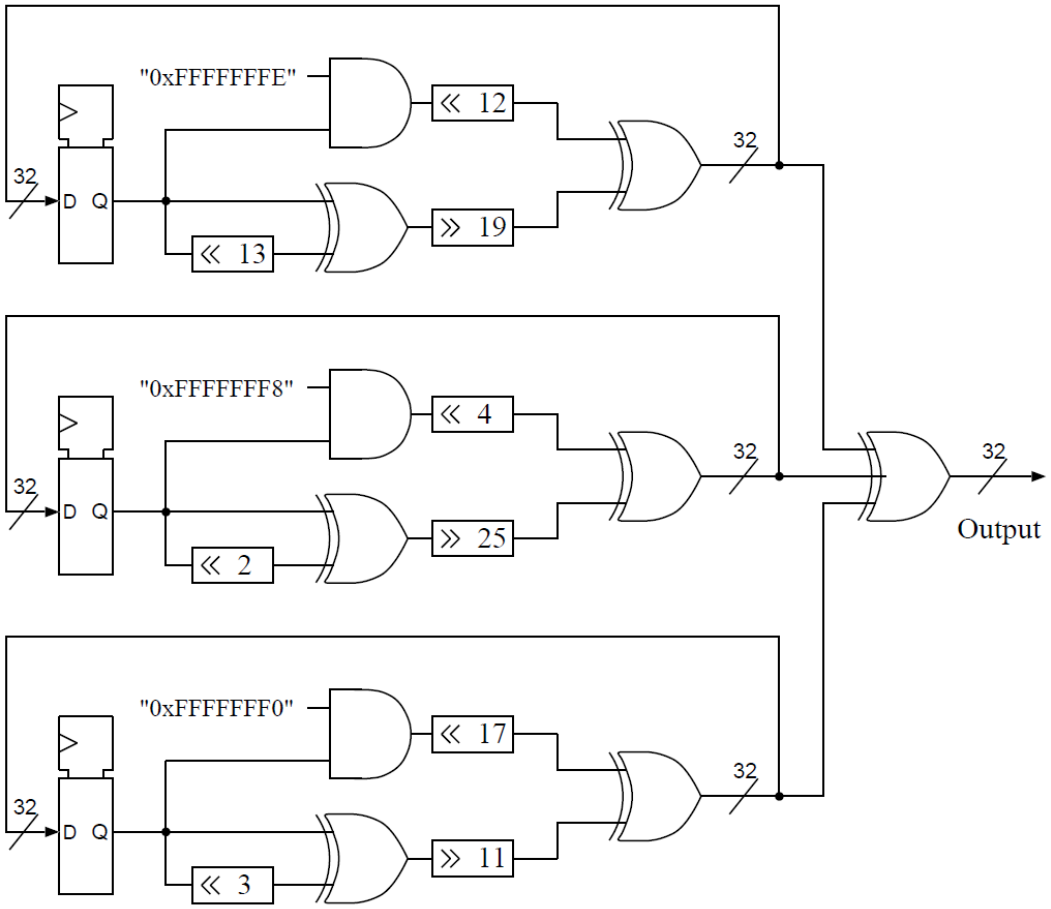
CLT based Gaussian Random Number Generator

- Sum of uniformly distributed random numbers approximates a GRN
- Generated with Tausworthe URNGs
- Seed as input
- Shift operation at output, to get distribution in the desired range



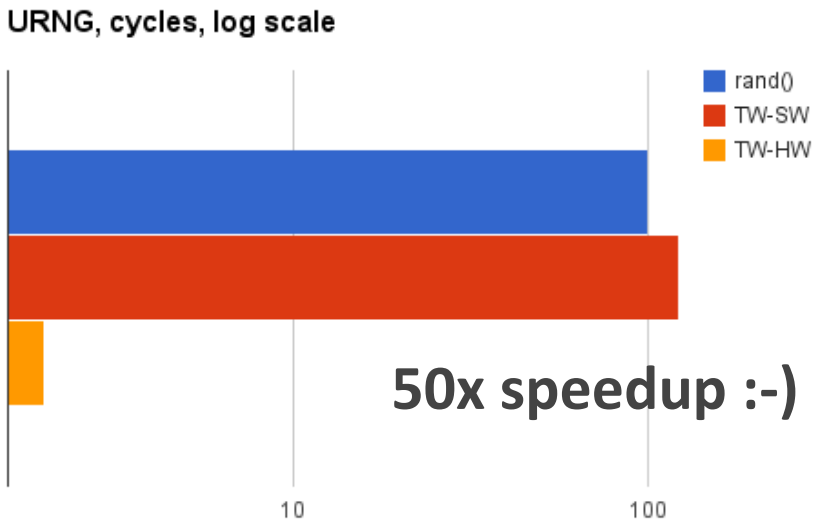
Basic Ingredient of CLT: Tausworthe URNG

- 32b output
- 3 x 32 bit state registers
- One output/cycle
- Implemented in C and HW (Xtensa TIE)



Results: URNG

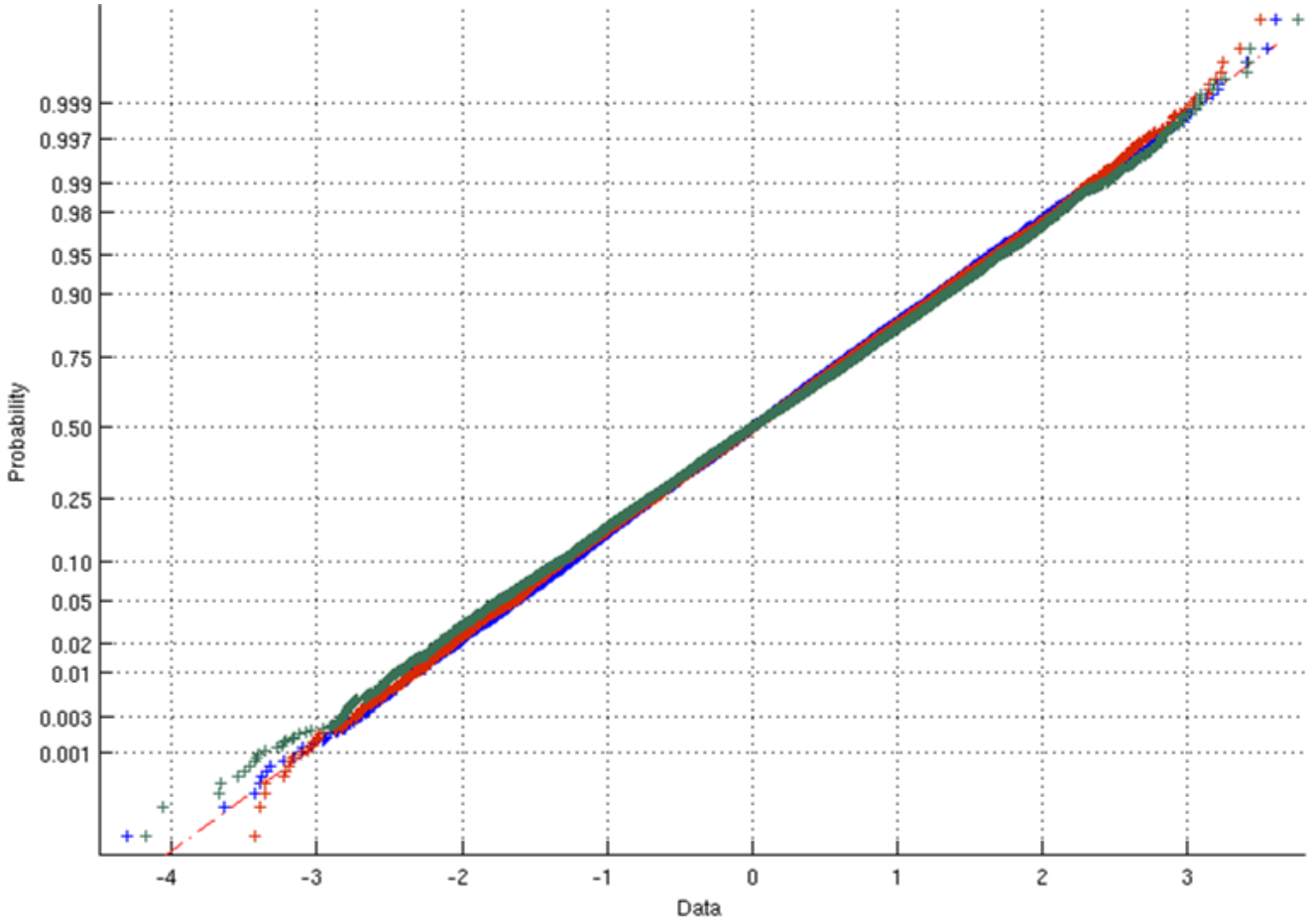
	rand()	TW-SW	TW-HW (TIE)	TW-HW [4], Virtex 4*
cycles/RN	100	121	2	1
accelerator GEs	0	0	2183	1065 to 1704



Uniformly distributed 32b integer random numbers can be generated in **one cycle** by investing **2.2kGE**, while the built-in **rand()** function takes 100 cycles.

[4] Pierre Greisen, Flexible Digital Emulator for a Wireless MIMO Channel, ETHZ IIS MSc Thesis Spring 2007
*15 to 25 GEs per FPGA slice according to Xilinx

Results: GRNG Performance - normplot()



TW-CLT12-HW

RAND-CLT12-SW

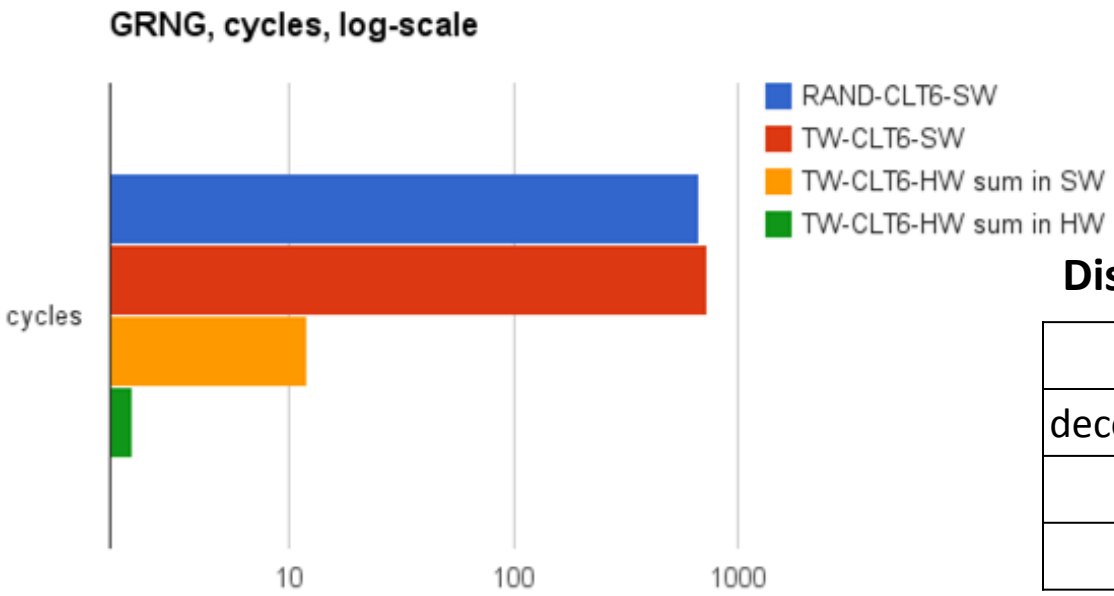
`randn()` Matlab

Observation:

Acceptable results between +/- 3 sigma (99.7% coverage)

Results: GRNG with 6 TW-URNGs

	RAND-CLT6-SW	TW-CLT6-SW	TW-CLT6-HW (TIE) sum in SW	TW-CLT6-HW (TIE)
cycles/RN	673	736	12	2
accelerator GEs	0	0	11014	11113

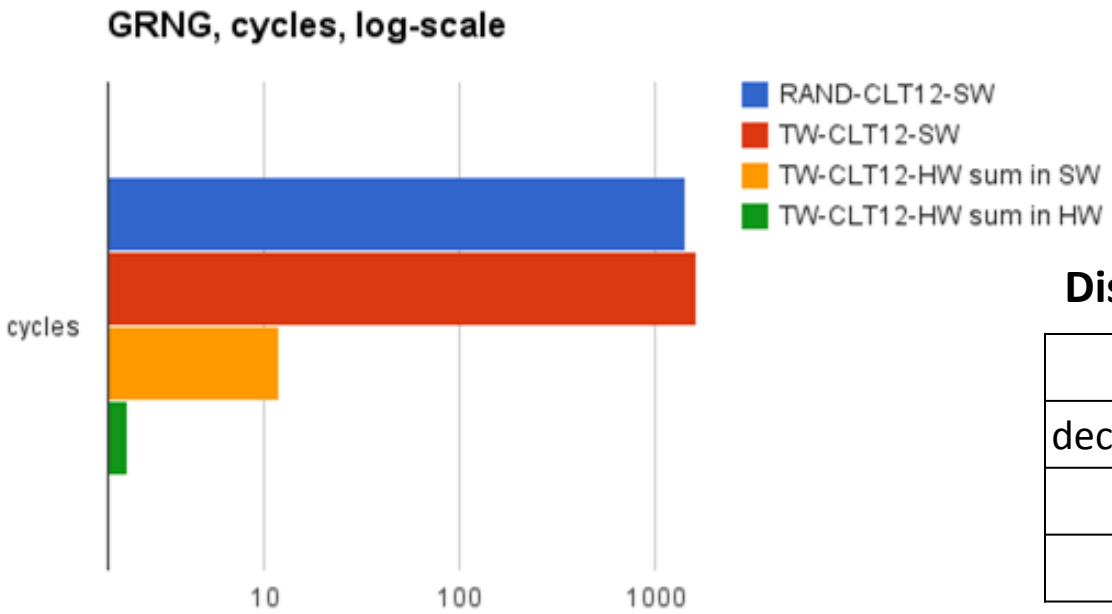


Distribution of accelerator gates:

TW-CLT6-HW sum in HW	
decode/multiplexing	20%
operations	14%
states	66%

Results: GRNG with 12 TW-URNGs

	RAND-CLT12-SW	TW-CLT12-SW	TW-CLT12-HW (TIE) sum in SW	TW-CLT12-HW (TIE)
cycles/RN	1440	1603	24	2
accelerator GEs	0	0	21628	21805



Distribution of accelerator gates:

TW-CLT12-HW sum in HW	
decode/multiplexing	18%
operations	15%
states	67%

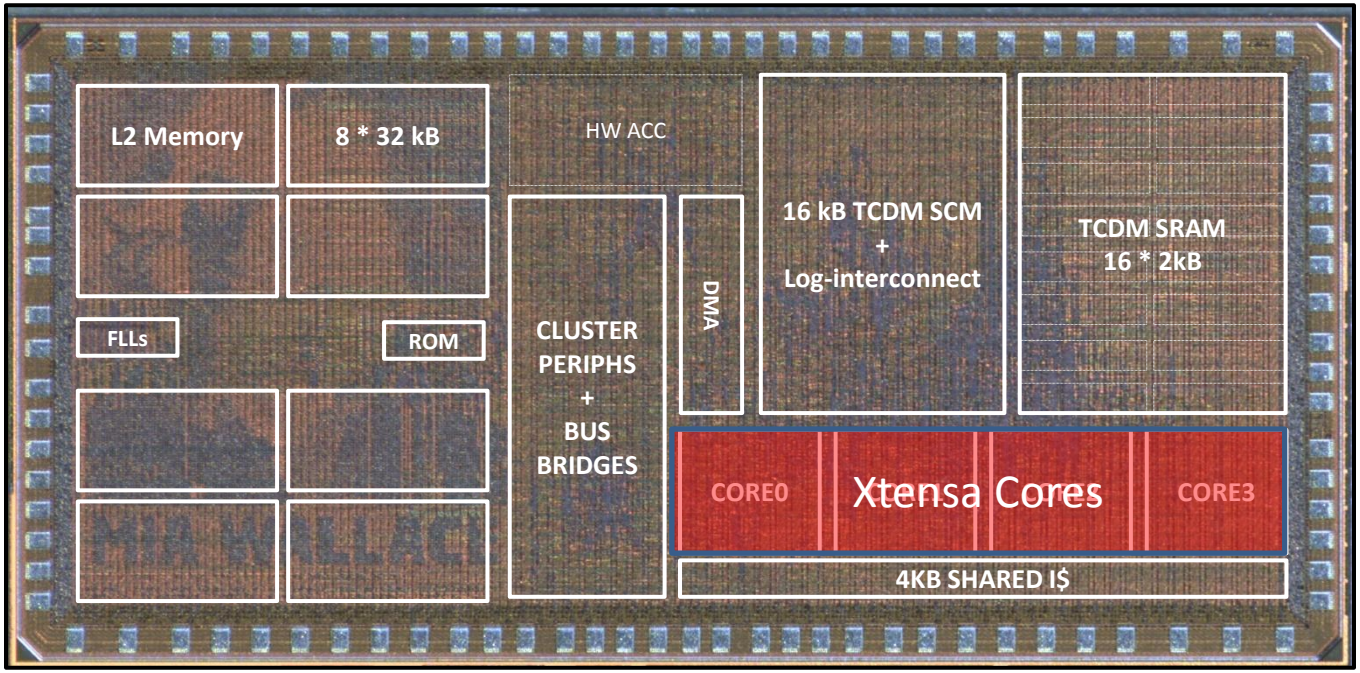
Normally distributed random numbers can be generated in **two cycles** with an **N-stage CLT**-based GRNG, which requires **N x 1.8kGE**.

Conclusion

- With the Xtensa CoreGen development framework it is possible to implement and evaluate “small” hardware projects in a **couple of days**.
- Speedups up to 50x can be achieved by replacing critical functions with hardware TIE.
- Since we have a lot of experience in hardware design and designed a lot of chips, we would be very interested in doing tapeouts with the Xtensa Cores.
  - Optimize an Xtensa core for near threshold operation
  - Ultimate goal: Heterogeneous PULP architecture with clusters of Xtensa cores specialized for computer vision

Questions

PULP cluster:



Acknowledgements:
Sven Stucki, Sandro Belfanti, Harald Kröll