

Experience Feedback on HLS Implementation of LWE PQC on FPGA

Arnaud TISSERAND

CNRS, Lab-STICC

Nov. 2021 — GDR Sécu & SoC2

Agenda

- ▶ Context, motivation and experimental environment
- ▶ HLS implementation of modular arithmetic operations
- ▶ HLS implementation and parallelization of (M/R)LWE
- ▶ HLS implementation of SCA countermeasures
- ▶ Remarks and future prospects

Work Context

Mainly the work of Timo ZIJLSTRA during his PhD Thesis:

- ▶ Secure Hardware Accelerators for Post-Quantum Cryptography
- ▶ Oct. 2017 – Sep. 2020, PEC grant funded by DGA and Région Bretagne
- ▶ Co-advisor: Karim BIGOU (UBO)
- ▶ Implementation of lattice based solutions for PQC:
 - ▶ modular arithmetic (sequences of operations)
 - ▶ LWE
 - ▶ MLWE
 - ▶ RLWE (with SCA countermeasures)

Feedback from experience on HDL implementations

Notation: URL/links

Motivation (1/2)

We have been implementing **arithmetic accelerators** for cryptography in hardware (ASIC & FPGA) using HDL descriptions and tools for quite some time, **but**

- ▶ hiring PhD students with skills in both arithmetic, crypto and hardware implementation is difficult
- ▶ requires time for PhD students to learn and experiment
- ▶ HDL (Verilog or VHDL) coding is tedious \implies it limits exploration at architecture/algorithm/arithmetic levels

Motivation (2/2)

High-Level Synthesis (HLS) should help us to:

- ▶ reduce design time
- ▶ explore more advanced arithmetic algorithms, representations of numbers and architectures
- ▶ use advanced optimizations (compared to manually optimized solutions)
- ▶ quickly compare various solutions using the same environment
- ▶ other benefits?

Experimental Environment

- ▶ Vivado HLS tools (v. 2017.3 & 2018.1) from Xilinx
- ▶ Implementations and simulations for several FPGAs: Artix-7, Kintex-7, Virtex-7, Zynq UltraScale+, Kintex UltraScale, Virtex UltraScale from Xilinx
- ▶ Verifications on a ZedBoard card (w. XC7Z020 FPGA)
- ▶ Codes written in C (available as open source)
- ▶ Intensive comparisons to values obtained from SageMath
- ▶ Thanks to Xilinx for donations through the XUP

Related Publications

- ▶ [DZBT19] Compas 2019: Comparaison d'algorithmes de réduction modulaire en HLS sur FPGA, par L. Djath, T. Zijlstra, and K. Bigou et A. Tisserand
- ▶ [ZBT19] IndoCrypt 2019: FPGA Implementation and Comparison of Protections against SCAs for RLWE, by T. Zijlstra, K. Bigou, and A. Tisserand
- ▶ [ZBT21] IEEE Trans. Computers 2021: Lattice-based Cryptosystems on FPGA: Parallelization and Comparison using HLS, by T. Zijlstra, K. Bigou, and A. Tisserand
- ▶ C Codes: LWE crypto in HLS on FPGA
- ▶ [GT19] IEEE Trans. Computers 2019. Generation of Finely-Pipelined GF(P) Multipliers for Flexible Curve based Cryptography on FPGAs, by G. Gallin and A. Tisserand.

HLS Implementation of Modular Arithmetic

Need for efficient modular arithmetic on small values in:

- ▶ lattice based solutions for PQC (e.g. 13 – 25 bits field elements)
- ▶ ECC in RNS (vectors of residues on 16 – 64 bits)

Experiment/optimize various **algorithms** and **architectures** for:

- ▶ modular arithmetic (sequences of) **operations**
 - ▶ $a \pm b \bmod m$
 - ▶ $a \times b \bmod m$
 - ▶ $\sum_{i=0}^{N-1} a_i \bmod m$
 - ▶ $\sum_{i=0}^{N-1} a_i \times b_i \bmod m$
- ▶ various **widths** $w \in \{10, \dots, 64\}$ bits
- ▶ various **forms** of moduli
 - ▶ generic m
 - ▶ sparse m for PQC (e.g. a few non-zero digits among w)
 - ▶ RNS friendly $m = 2^w - c$ (with c small)

User Code for $(\sum_{i=0}^{N-1} a_i \times b_i) \bmod m$

```
1  #include "parameters.h"
2  #include "arithmod.h"
3
4  word m2_rsf(word A[N], word B[N])
5  {
6      sumdword res=0;
7      acc: for(counter i=0; i<N; i++)
8          res += DW(A[i]) * DW(B[i]);
9      return barrett(res);
10 }
```

word, dword, sumdword, ... are *typedefs* for w , $2w$, $2w + \lceil \log_2 N \rceil$ bits values

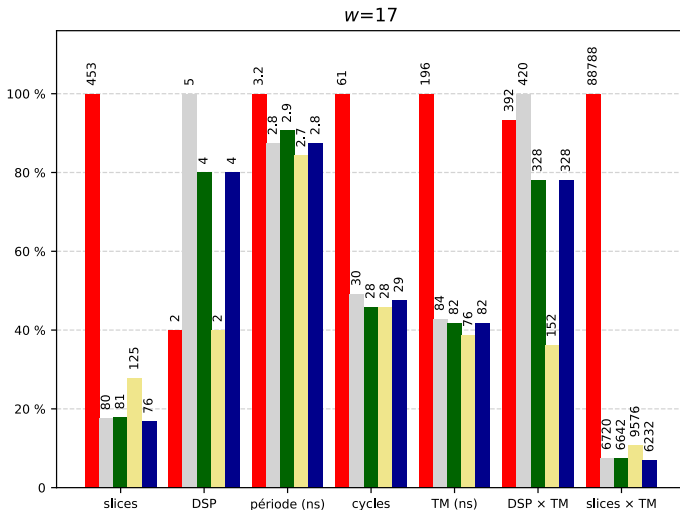
W(), DW(), SUM_DW(), ... are cast *macros* for these types

HLS tools require to **label** loops, function calls, operations, ... to apply **directives** (unroll, pipeline, memory, unit mapping...)

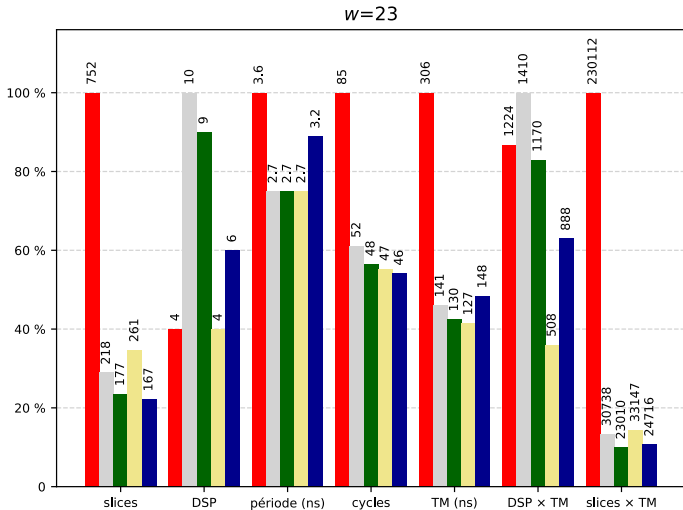
Library Code Example for Barrett Reduction

```
1  #include "parameters.h"
2  #include "arithmod_internal.h"
3
4  word barrett(sumdword x)
5  {
6      sumword x1 = SUM_W(x >> width);
7      sumword q = SUM_W((RSW(x1) * RSW(R_const)) >> (shift - width));
8      word x0 = W(x);
9      counter c = 0;
10     if (x0 > M) c = 2;
11     else if (x0 != 0) c = 1;
12     q = q + c;
13     sumdword z = SUM_DW(q) * SUM_DW(m);
14     signword res = x - z;
15     if (res < 0) res = res + M;
16     if (res < 0) res = res + M;
17     return W(res);
18 }
```

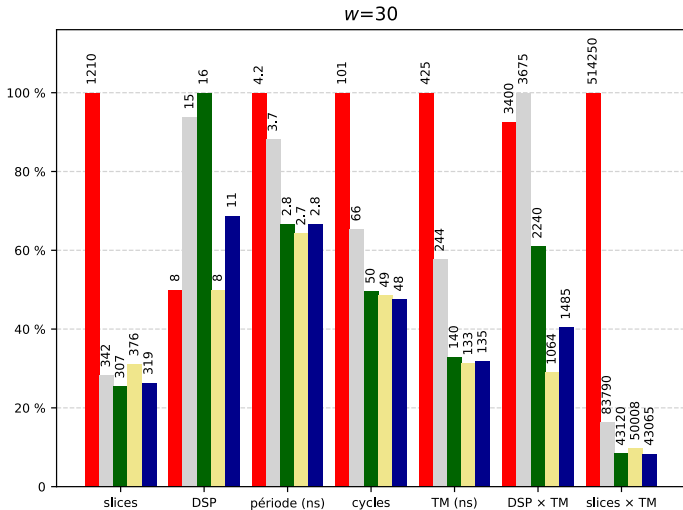
Implementation Results for $(\sum_{i=0}^{N-1} a_i \times b_i) \bmod m$ (1/3)



Implementation Results for $(\sum_{i=0}^{N-1} a_i \times b_i) \bmod m$ (2/3)

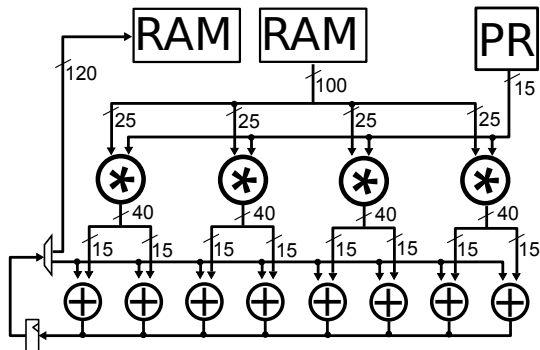


Implementation Results for $(\sum_{i=0}^{N-1} a_i \times b_i) \bmod m$ (3/3)



HLS Implementation of LWE (1/6)

Base architecture for multiplication of 8×640 matrices with 15 bits coefficients:



HLS Implementation of LWE (2/6)

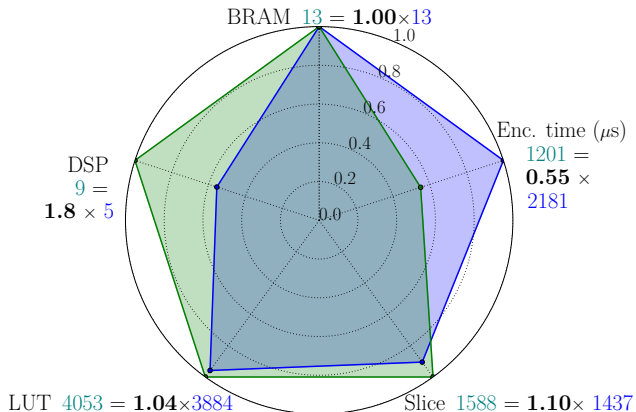
Code for matrix multiplication:

```
col_A: for(i=0; i<k; i++){
    copy1: for(ii=0; ii<8; ii++)
        C1_tmp[ii] = C1[ii][i]; // copy BRAM -> registers
    row_A: for(jj=0; jj<k; jj++){
        sum = 0;
        prng(State_A, &a_coeff); // PK coeff. from PRNG
        row_E: for(j=0; j<4; j++){
            comp_2prods(a_coeff, E1[j][jj], &prod1, &prod2);
            C1_tmp[2*j] = C1_tmp[2*j] + prod1; // update C1
            C1_tmp[2*j+1] = C1_tmp[2*j+1] + prod2;
        }
    }
    copy2 :for(ii=0; ii<8; ii++)
        C1[ii][i] = C1_tmp[ii]; // copy registers -> BRAM
}
```

Exploration of numerous directives combinations (#pragma HLS ...):
unroll, pipeline, inline, allocation, dependence,
array_partition, array_reshape, array_map

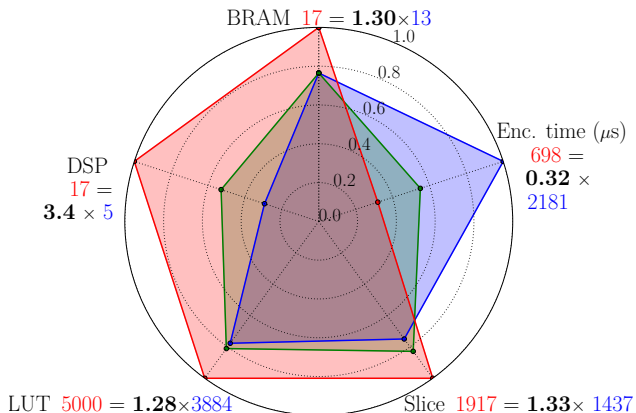
HLS Implementation of LWE (3/6)

Parallel implementation with unrolling factor 2:



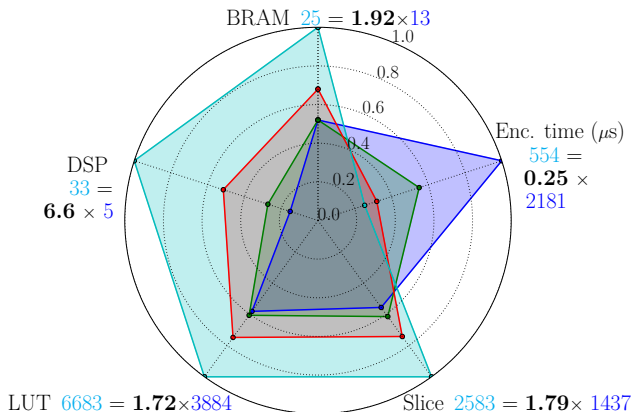
HLS Implementation of LWE (4/6)

Parallel implementation with unrolling factor 4:



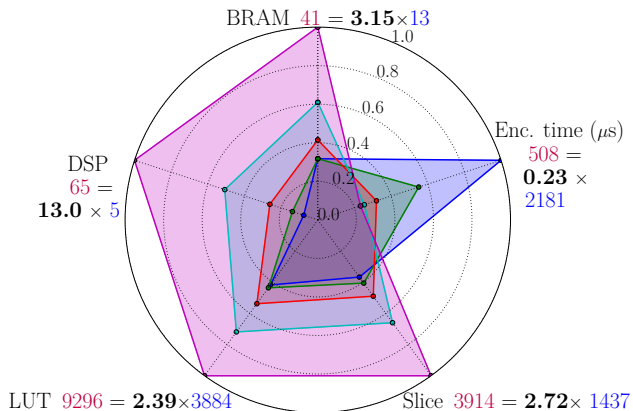
HLS Implementation of LWE (5/6)

Parallel implementation with unrolling factor 8:



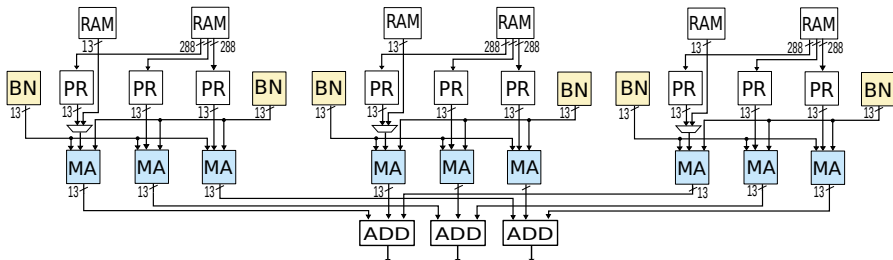
HLS Implementation of LWE (6/6)

Parallel implementation with unrolling factor 16:



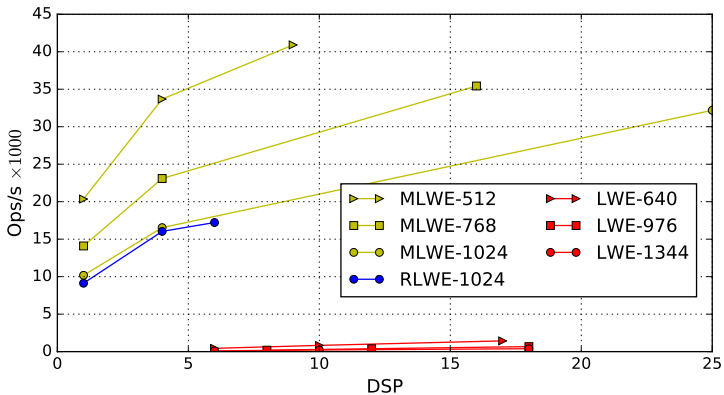
HLS Implementation of MLWE

Example of architecture “easily” implemented and optimized:



Comparison LWE – MLWE – RLWE (1/2)

Throughput (in k-encryptions per second) vs area (in DSPs) trade-offs for various parallelism levels. The most left point of each curve corresponds to a sequential architecture, the middle point embeds parallel NTTs (for RLWE/MLWE) and the most right point is a full parallel architecture.



Comparison LWE – MLWE – RLWE (2/2)

Source	Lvl. of Parallel.	Scheme type-size	Algorithm	PRNG	Type	FPGA family (model)	Freq. MHz	Time μ s	Area DSP, 18kb BRAM, Slices, LUT
[18]	high	LWE-640	Frodo	Hybrid	CCA	Artix-7	171	1212	16, 0, 1692, 5796
[18]	medium	LWE-640	Frodo	Hybrid	CCA	Artix-7	177	2342	8, 0, 1485, 5155
TW	low	LWE-640		Hybrid	CCA	Artix-7 (200)	159	2972	5, 37, 12951, 39077 *
[18]	low	LWE-640	Frodo	Hybrid	CCA	Artix-7	183	4624	4, 0, 1338, 4620
[25]		LWE-640	Frodo	Hybrid	CCA	Artix-7 (35T)	167	19608	1, 11, 1855, 6745
[18]	high	LWE-976	Frodo	Hybrid	CCA	Artix-7	168	2857	16, 0, 1782, 6188
[18]	medium	LWE-976	Frodo	Hybrid	CCA	Artix-7	175	5464	8, 0, 1608, 5562
TW	low	LWE-976		Hybrid	CCA	Artix-7 (200)	167	6317	14, 37, 13468, 41100 *
[18]	low	LWE-976	Frodo	Hybrid	CCA	Artix-7	180	10638	4, 0, 1455, 4996
[25]		LWE-976	Frodo	Hybrid	CCA	Artix-7 (35T)	166	45455	1, 16, 1985, 7209
TW	low	LWE-1344		Hybrid	CCA	Artix-7 200	167	11606	6, 62, 12299, 37342 *
[20]		RLWE-1024	NewHope	SHAKE	K-E	Zynq-7 (20)	131	79	8, 14, n.a., 20826
[19]		RLWE-1024	NewHope	SHAKE	K-E	Artix-7 (35T)	117	1532	2, 4, n.a., 4498
TW	medium	RLWE-512		Hybrid	CPA	Artix-7 (200)	211	50	5, 12, 7797, 16338 *
[34]		RLWE-1024	NewHope	SHAKE	CPA	Zynq-7 (20)	200	62	2, 8, n.a., 6781 *
TW	medium	RLWE-1024		Trivium	CPA	Artix-7 (200)	259	63	4, 10, 3701, 10112 *
[31]		RLWR-1018	Round5	SHAKE	CPA	Cyclone V (5csea5)	133	1000	4116 ALM, 10753 bytes *
TW	medium	RLWE-1024		Hybrid	CCA	Artix-7 200	167	137	9, 17, 14026, 42062 *
[31]		RLWR-1170	Round5	SHAKE	CCA	Cyclone V (5csea5)	130	1350	6337 ALM, 11765 bytes *
[36]		MLWR-512	Saber	SHAKE	CPA	UltraScale+	100	5.2	85, 12, n.a., 34886
[36]		MLWR-768	Saber	SHAKE	CPA	UltraScale+	100	11.6	85, 12, n.a., 34886
[36]		MLWR-1024	Saber	SHAKE	CPA	UltraScale+	100	21.0	85, 12, n.a., 34886
[35]		MLWE-512	Kyber	SHAKE	CCA	Artix-7 (12T)	161	30.5	2, 6, 2126, 7412
TW	medium	MLWE-512		Hybrid	CCA	Artix-7 (200)	170	60	11, 16, 11028, 34206 *
[37]	high	MLWR-768	Saber	SHAKE	CCA	UltraScale+ (9eg)	250	26	0, 2, n.a., 23600 *
[35]		MLWE-768	Kyber	SHAKE	CCA	Artix-7 (12T)	161	47.6	2, 6, 2126, 7412
TW	medium	MLWE-768		Hybrid	CCA	Artix-7 (200)	167	88	11, 16, 11890, 34145 *
[38]		MLWR-768	Saber	SHAKE	CCA	Zynq-7 (20)	125	4147	28, 4, n.a., 7400 *
TW	medium	MLWE-1024		Hybrid	CCA	UltraScale+ (9eg)	417	48	9, 16, 9314, 44964 *
[35]		MLWE-1024	Kyber	SHAKE	CCA	Artix-7 (12T)	161	67.9	2, 6, 2126, 7412
TW	medium	MLWE-1024		Hybrid	CCA	Artix-7 (200)	170	116	11, 16, 11567, 33707 *
[39]		MLWE-1024	Kyber	SHAKE	CCA	Artix-7 (35T)	60	6900	4, 34, n.a., 1738

FPGA “Scalability”

Implementation results for CCA-secure MWLE-1024 using SHAKE256 for error sampling on different FPGA families using Vivado 2018.3:

FPGA family	Freq. MHz	Time μs enc/dec	Area DSP, BRAM, Slices, LUT
Artix-7	200	99/110	11, 16, 11322, 35607
Kintex-7	286	70/77	9, 16, 12066, 34175
Virtex-7	286	70/77	9, 16, 12508, 35718
Zynq UltraScale+	417	48/53	9, 16, 9314, 44964
Kintex UltraScale	333	61/68	9, 16, 7238, 43101
Virtex UltraScale	286	69/77	9, 16, 6474, 33979

HLS Implementation of SCA Countermeasures (1/2)

Implementation of various countermeasures from state of art:

- ▶ masking
- ▶ shifting
- ▶ blinding

Implementation of our own countermeasures:

- ▶ masking with deterministic decoder
- ▶ shuffling (LSFR and permutation network)
- ▶ randomized redundant representations
 $\mathbb{Z}/(2^r q)\mathbb{Z}$ instead of $\mathbb{Z}/q\mathbb{Z}$ with random multiples of q

See [ZBT19] for details. . .

HLS Implementation of SCA Countermeasures (2/2)

FPGA results for RLWE with various countermeasures and $(q, n) = (7681, 256)$ (timings for decryption only):

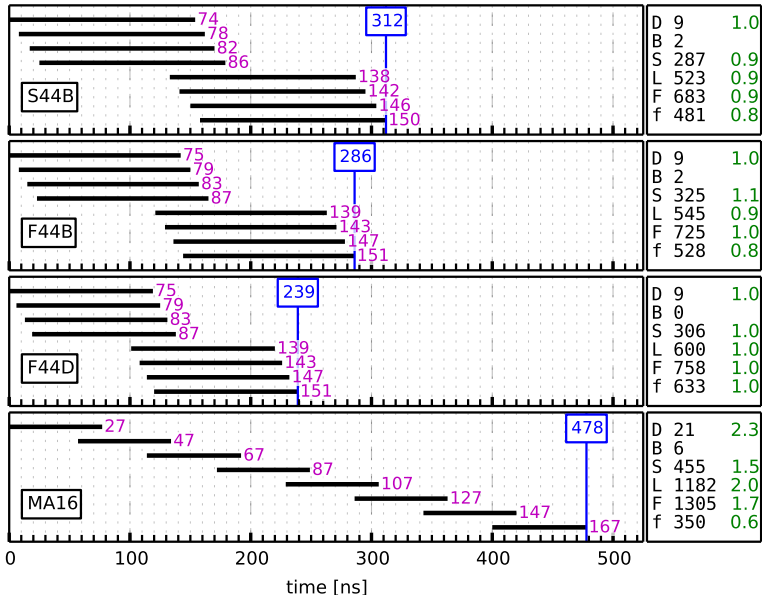
Counter-measure	Entropy added (bits)	Src.	Impl.	FPGA	Lat.	Clk. (ns)	Time (μs)	Slice, LUT, DSP, BRAM
None	0	-	25	V2	2800	8.3	23.5	-, 1713, 1, -
Masking	3328	25	25		7500	10	75.2	-, 2014, 1, -
None	0	-	this work	A7	2357	3.3	7.8	483, 1163, 2, 3
Blinding	16	27			2768	3.8	10.6	941, 2284, 3, 4
Shifting	16	27			3138	4.7	14.8	832, 2150, 3, 4
Shift + Blind	32	27			3183	4.6	14.7	1063, 2781, 3, 4
Masking	3328	25			2517	4.0	10.1	2187, 5500, 5, 6
Our Mask.	3328				2510	4.0	10.1	1722, 4269, 5, 6
Permutation	1280	this work			2521	4.5	11.4	3183, 7385, 2, 4
LFSR ctr.	71				2846	3.6	10.3	1069, 2861, 2, 3
$r = 1$	256				2272	3.7	8.5	629, 1599, 2, 3
$r = 2$	512				2273	3.6	8.2	611, 1664, 2, 3
$r = 3$	768				2333	3.8	8.9	807, 2067, 2, 3
$r = 4$	1024				2338	3.6	8.5	872, 2285, 2, 3
$r = 5$	1280				2352	3.8	9.0	990, 2677, 2, 6
$r = 6$	1536				2394	3.9	9.4	1254, 3466, 3, 6
$r = 7$	1792	2410	3.9	9.4	1713, 5017, 3, 6			
$r = 8$	2048	2426	3.9	9.5	2544, 7837, 3, 6			

HLS vs HDL

???

"HDL" Experience (2/2)

Generated HDL results for 128b HTMM on Virtex-7:



Remarks

- ▶ HLS is interesting for arithmetic and algorithmic exploration
- ▶ Provides good results **after** important code optimizations (/rewrite)
- ▶ Requires some experience on FPGA implementation
- ▶ Still room for improvement
 - ▶ frequency & pipeline
 - ▶ memory directives
 - ▶ parallel descriptions
 - ▶ ...
- ▶ Is C a good language for HLS?
- ▶ In software, why do we still write parts in assembly?
 - ▶ are we “better” than compilers? NO!
 - ▶ we don't know how to write specific behavior in C
 - ▶ some architecture features are not yet supported

Future Prospects

We plan to work on:

- ▶ An arithmetic library for asymmetric crypto in HLS with various representations of numbers and advanced algorithms
- ▶ “Calibration” of our library components from implementation **S** results **S**
- ▶ Countermeasures
- ▶ Training

Thank you! Questions?

Contact: arnaud.tisserand@cnrs.fr