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 → it limits exploration at architecture/algorithms/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


- [ZBT19] IndoCrypt 2019: FPGA Implementation and Comparison of Protections against SCAs for RLWE, by T. Zijlstra, K. Bigou, and A. Tisserand


- C Codes: LWE crypto in HLS on FPGA

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
  - \( a \pm b \mod m \)
  - \( a \times b \mod m \)
  - \( \sum_{i=0}^{N-1} a_i \mod m \)
  - \( \sum_{i=0}^{N-1} a_i \times b_i \mod m \)
- various \textit{widths} \( w \in \{10, \ldots, 64\} \) bits
- various \textit{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 \((\sum_{i=0}^{N-1} a_i \times b_i) \mod m\)

```c
#include "parameters.h"
#include "arithmod.h"

word m2_rsf(word A[N], word B[N])
{
    sumdword res=0;
    acc: for (counter i=0; i<N; i++)
        res += DW(A[i]) * DW(B[i]);
    return barrett(res);
}
```

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

\(W(), DW(), SUM_DW(), \ldots\) are cast macros for these types

HLS tools require to label loops, function calls, operations, ... to apply directives (unroll, pipeline, memory, unit mapping...))
```c
#include "parameters.h"
#include "arithmod_internal.h"

word barrett(sumdword x) {
    sumword x1 = SUM_W(x >> width);
    sumword q = SUM_W((RSW(x1) * RSW(R_const)) >> (shift - width));
    word x0 = W(x);
    counter c = 0;
    if (x0 > M) c = 2;
    else if (x0 != 0) c = 1;
    q = q + c;
    sumdword z = SUM_DW(q) * SUM_DW(m);
    signword res = x - z;
    if (res < 0) res = res + M;
    if (res < 0) res = res + M;
    return W(res);
}
```
Implementation Results for \( (\sum_{i=0}^{N-1} a_i \times b_i) \mod m \) (1/3)
Implementation Results for $(\sum_{i=0}^{N-1} a_i \times b_i) \mod m$ (2/3)
Implementation Results for \((\sum_{i=0}^{N-1} a_i \times b_i) \mod m\) (3/3)
Base architecture for multiplication of $8 \times 640$ 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);
        }
    }
    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
Parallel implementation with unrolling factor 2:

- **BRAM 13 = 1.00**
- **Enc. time (μs):**
  - 1201 = 0.55
  - 2181
- **Slice 1588 = 1.10**
- **LUT 4053 = 1.04**
- **DSP 9 = 1.8**
- **LUT 4053 = 1.04**
- **Slice 1588 = 1.10**
Parallel implementation with unrolling factor 4:

- BRAM 17 = 1.30
- Enc. time (s) 698 = 0.32
- Slice 1917 = 1.33
- LUT 5000 = 1.28
- DSP 17 = 3.4
- Enc. time (µs) 698 = 0.32
HLS Implementation of LWE (5/6)

Parallel implementation with unrolling factor 8:

```
BRAM 25 = 1.92×13
Enc. time (s) 554 = 0.25 × 2181

LUT 6683 = 1.72×3884
Slice 2583 = 1.79×1437
```

```
Parallel implementation with unrolling factor 16:

- BRAM 41 = 3.15 \times 13
- Enc. time (\mu s) = 0.23 \times 2181
- Slice 3914 = 2.72 \times 1437
- LUT 9296 = 2.39 \times 3884
- DSP 65 = 13.0 \times 5
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)

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

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

<table>
<thead>
<tr>
<th>FPGA family</th>
<th>Freq. MHz</th>
<th>Time $\mu s$</th>
<th>Area DSP, BRAM, Slices, LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artix-7</td>
<td>200</td>
<td>99/110</td>
<td>11, 16, 11322, 35607</td>
</tr>
<tr>
<td>Kintex-7</td>
<td>286</td>
<td>70/77</td>
<td>9, 16, 12066, 34175</td>
</tr>
<tr>
<td>Virtex-7</td>
<td>286</td>
<td>70/77</td>
<td>9, 16, 12508, 35718</td>
</tr>
<tr>
<td>Zynq UltraScale+</td>
<td>417</td>
<td>48/53</td>
<td>9, 16, 9314, 44964</td>
</tr>
<tr>
<td>Kintex UltraScale</td>
<td>333</td>
<td>61/68</td>
<td>9, 16, 7238, 43101</td>
</tr>
<tr>
<td>Virtex UltraScale</td>
<td>286</td>
<td>69/77</td>
<td>9, 16, 6474, 33979</td>
</tr>
</tbody>
</table>
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 \( \mathbb{Z}/q\mathbb{Z} \) for \( r \) random

See [ZBT19] for details...
FPGA results for RLWE with various countermeasures and \((q, n) = (7681, 256)\) (timings for decryption only):

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

???
“HDL” Experience (1/2)

Example: Hyperthreaded modular Montgomery multiplier (HTMM) for (H)ECC accelerators with high frequency AND reduced area using HDL codes generated by homemade generators (see [GT19])
“HDL” Experience (2/2)

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

- **S44B**
  - D 9: 1.0
  - B 2
  - S 287: 0.9
  - L 523: 0.9
  - F 683: 0.9
  - f 481: 0.8

- **F44B**
  - D 9: 1.0
  - B 2
  - S 325: 1.1
  - L 545: 0.9
  - F 725: 1.0
  - f 528: 0.8

- **F44D**
  - D 9: 1.0
  - B 0
  - S 396: 1.0
  - L 600: 1.0
  - F 758: 1.0
  - f 633: 1.0

- **MA16**
  - D 21: 2.3
  - B 6
  - S 455: 1.5
  - L 1182: 2.0
  - F 1305: 1.7
  - f 350: 0.6
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 results
▶ Countermeasures
▶ Training
Thank you! Questions?

Contact: arnaud.tisserand@cnrs.fr