Crystal Kyber implementation using High Level Synthesis

by Pakin Panawattanakul

7/6/2025

Recap our group

Before this presentation selecting and understanding Post Quantum Cryptography

- **Ninny** : Hash algorithm
- **Tony** : module learning with errors/ polynomial ring
- **Previous presentation** : Kyber PKE & Kyber KEM algorithms

Researching & understanding theory → Start to find implementations methods

Literature reviews

7th Workshop on Communication Networks and Power Systems (WCNPS 2022)

Quantum-resistant Cryptography in FPGA

Renata C. Policarpo *, Alexandre S. Nery *, Robson de O. Albuquerque *
Professional Post-Graduate Program in Electrical Engineering,
Department of Electrical Engineering*, University of Brasília, Brasília 70910-900
Email: renata.policarpo@aluno.unb.br; alexandre.nery@redes.unb.br; robson@redes.unb.br

MAIN OBJECTIVE

Article

- To design and implement FPGA co-processor for Crystals Kyber using High Level Synthesis tools.
- Focusing on optimizing polynomial multiplications

This presentations

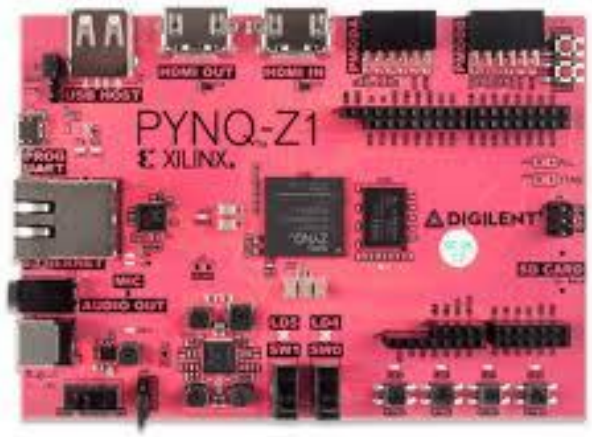
- Review implement Kyber KEM on FPGAs
  - Hardware?
  - which part of algorithms?
  - Method?
- Compare with our project plans

	Compiler	**High Level Synthesis tools (HLS)**
input	High-level code (e.g., C/C++).	High-level code (e.g., C/C++).
transformation	Translates to machine code for a CPU.	Translates to Hardware Description Language (HDL) for FPGAs/ASICs.
output	Executable software	Custom hardware design (RTL).
core purpose	Create software that *runs on* hardware	Create the hardware *itself*

FPGA Board : XC7Z020-1CLG400C

- Programmable logic equivalent to Artix-7 FPGA **(FPGA chip)**
- **CPU** : Arm cortex A9
- **communication modules**
  - ethernet, USB, SDIO
  - ps7-axi-periph, axi_mem_intercon (IP core)
- **Memory**
  - 512 MB DRAM (552 MHz)
  - 16 MB flash memory

https://digilent.com/shop/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/



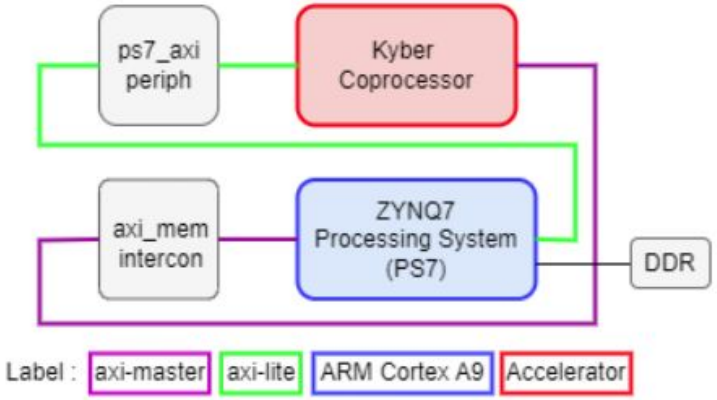
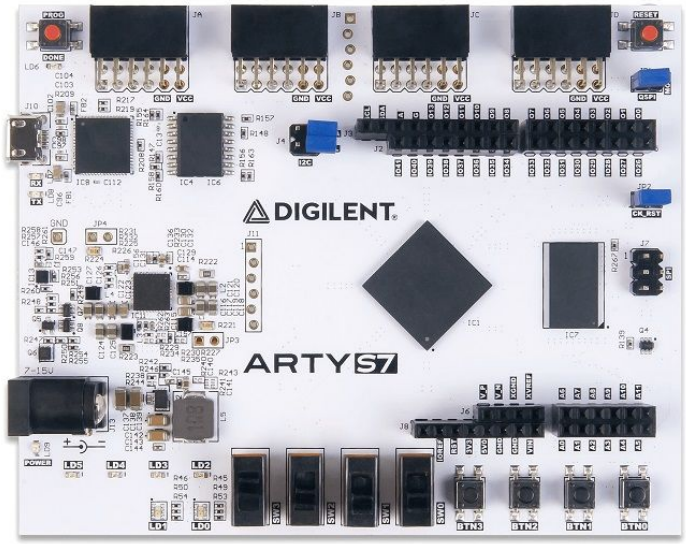


Figure 2: Kyber Architecture and Interface

Our FPGA board: Arty S7

- Only **FPGA chip** : Artix-7
- **No onboard CPU** → Use FPGA as coprocessor and connect to PC/Laptop via UART
- **Memory**
  - 256 MB DRAM (325 MHz)
  - 128 Mbits flash memory



https://digilent.com/shop/arty-s7-spartan-7-fpga-development-board/

1. Polynomial Multiplication

2. Barett reduction

(coefficient reduction)

```
//copy data into the accelerator
for (i = 0 ; i < KYBER_K_hls ; i++) {
   for (j = 0 ; j < KYBER_N_hls ; j++) {
 #pragma HLS PIPELINE
 a_hls.vec[i].coeffs[j] = a->vec[i].coeffs[j];
 b_hls.vec[i].coeffs[j] = b->vec[i].coeffs[j];
   }
}

/********** polynomial multiplication **********/

poly_basemul_montgomery_hls(&r_hls, &a_hls.vec[0],
      &b_hls.vec[0]);

for(i=1;i<KYBER_K_hls;i++) {
   poly_basemul_montgomery_hls(&t, &a_hls.vec[i],
         &b_hls.vec[i]);
   poly_add_hls(&r_hls, &r_hls, &t);
}

//Barrett reduction
poly_reduce_hls(&r_hls);
```

- Polynomial ring : $R_q = \mathbb{Z}_q[x]/(x^n + 1)$
  - q , n
  - Coefficient in $\mathbb{Z}_q \in \{0,1,2,\dots,q-1\}$
  - Degree = $n - 1$ → $a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$
  - After multiplying apply modular reduction $/(x^n + 1)$

e.g. q = 17, n = 4

f(x) = $2 + 16x + 3x^2 + 5x^3$

g(x) = $9 + x + 14x^3$

$$f(x)g(x) = 18 + 146x + 43x^2 + 76x^3 + 229x^4 + 42x^5 + 70x^6$$

Polynomial multiplication

coef. mod q

$$= 1 + 10x + 9x^2 + 8x^3 + 8x^4 + 8x^5 + 2x^6$$

Barrett reduction

modular reduction $/(x^n + 1)$

$$= 10 + 2x + 7x^2 + 8x^3$$

```
/********** polynomial multiplication **********/

poly_basemul_montgomery_hls(&r_hls, &a_hls.vec[0],
        &b_hls.vec[0]);

for(i=1;i<KYBER_K_hls;i++) {
    poly_basemul_montgomery_hls(&t, &a_hls.vec[i],
            &b_hls.vec[i]);
    poly_add_hls(&r_hls, &r_hls, &t);
}
```

K = 3
for Kyber-768

b_hls

a_hls

$$\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 2 & 5 & 1 \\ 6 & 7 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 15 & 20 & 4 \end{bmatrix}$$

r_hls

Official Kyber reference code written in C

```c
void polyvec_basemul_acc_montgomery(poly *r, const polyvec *a, const polyvec *b)
{
  unsigned int i;
  poly tmp;

  poly_basemul_montgomery(r,&a->vec[0],&b->vec[0]);
  for(i=1;i<KYBER_K;i++) {
    poly_basemul_montgomery(&tmp,&a->vec[i],&b->vec[i]);
    poly_add(r,r,&tmp);
  }
}
```

```c
void polyvec_reduce(polyvec *r)
{
  unsigned int i;
  for(i=0;i<KYBER_K;i++)
    poly_reduce(&r->vec[i]);
}
```

basemul.S

- polynomial multiplications
- using montgomery technique (future work)

```
#include "consts.h"

.macro schoolbook off
vmovdqa         _16XQINV*2(%rcx),%ymm0
vmovdqa         (64*\off+ 0)*2(%rsi),%ymm1          # a0
vmovdqa         (64*\off+16)*2(%rsi),%ymm2          # b0
vmovdqa         (64*\off+32)*2(%rsi),%ymm3          # a1
vmovdqa         (64*\off+48)*2(%rsi),%ymm4          # b1

vpmullw         %ymm0,%ymm1,%ymm9                   # a0.lo
vpmullw         %ymm0,%ymm2,%ymm10                  # b0.lo
vpmullw         %ymm0,%ymm3,%ymm11                  # a1.lo
vpmullw         %ymm0,%ymm4,%ymm12                  # b1.lo

vmovdqa         (64*\off+ 0)*2(%rdx),%ymm5          # c0
vmovdqa         (64*\off+16)*2(%rdx),%ymm6          # d0

vpmulhw         %ymm5,%ymm1,%ymm13                  # a0c0.hi
vpmulhw         %ymm6,%ymm1,%ymm1                   # a0d0.hi
vpmulhw         %ymm5,%ymm2,%ymm14                  # b0c0.hi
vpmulhw         %ymm6,%ymm2,%ymm2                   # b0d0.hi

vmovdqa         (64*\off+32)*2(%rdx),%ymm7          # c1
vmovdqa         (64*\off+48)*2(%rdx),%ymm8          # d1

vpmulhw         %ymm7,%ymm3,%ymm15                  # a1c1.hi
vpmulhw         %ymm8,%ymm3,%ymm3                   # a1d1.hi
vpmulhw         %ymm7,%ymm4,%ymm0                   # b1c1.hi
vpmulhw         %ymm8,%ymm4,%ymm4                   # b1d1.hi

vmovdqa         %ymm13,(%rsp)

vpmullw         %ymm5,%ymm9,%ymm13                  # a0c0.lo
vpmullw         %ymm6,%ymm9,%ymm9                   # a0d0.lo
vpmullw         %ymm5,%ymm10,%ymm5                  # b0c0.lo
vpmullw         %ymm6,%ymm10,%ymm10                 # b0d0.lo

vpmullw         %ymm7,%ymm11,%ymm6                  # a1c1.lo
vpmullw         %ymm8,%ymm11,%ymm11                 # a1d1.lo
vpmullw         %ymm7,%ymm12,%ymm7                  # b1c1.lo
vpmullw         %ymm8,%ymm12,%ymm12                 # b1d1.lo

vmovdqa         _16XQ*2(%rcx),%ymm8
vpmulhw         %ymm8,%ymm13,%ymm13
vpmulhw         %ymm8,%ymm9,%ymm9
vpmulhw         %ymm8,%ymm5,%ymm5
vpmulhw         %ymm8,%ymm10,%ymm10
vpmulhw         %ymm8,%ymm6,%ymm6
vpmulhw         %ymm8,%ymm11,%ymm11
vpmulhw         %ymm8,%ymm7,%ymm7
vpmulhw         %ymm8,%ymm12,%ymm12

vpsubw          (%rsp),%ymm13,%ymm13                # -a0c0
vpsubw          %ymm9,%ymm1,%ymm9                   # a0d0
vpsubw          %ymm5,%ymm14,%ymm5                  # b0c0
vpsubw          %ymm10,%ymm2,%ymm10                 # b0d0

vpsubw          %ymm6,%ymm15,%ymm6                  # a1c1
vpsubw          %ymm11,%ymm3,%ymm11                 # a1d1
vpsubw          %ymm7,%ymm0,%ymm7                   # b1c1
vpsubw          %ymm12,%ymm4,%ymm12                 # b1d1

vmovdqa         (%r9),%ymm0
vmovdqa         32(%r9),%ymm1
vpmullw         %ymm0,%ymm10,%ymm2
vpmullw         %ymm0,%ymm12,%ymm3
vpmulhw         %ymm1,%ymm10,%ymm10
vpmulhw         %ymm1,%ymm12,%ymm12
vpmulhw         %ymm8,%ymm2,%ymm2
vpmulhw         %ymm8,%ymm3,%ymm3
vpsubw          %ymm2,%ymm10,%ymm10                 # rb0d0
vpsubw          %ymm3,%ymm12,%ymm12                 # rb1d1

vpaddw          %ymm5,%ymm9,%ymm9
vpaddw          %ymm7,%ymm11,%ymm11
vpsubw          %ymm13,%ymm10,%ymm13
vpsubw          %ymm12,%ymm6,%ymm6

vmovdqa         %ymm13,(64*\off+ 0)*2(%rdi)
vmovdqa         %ymm9,(64*\off+16)*2(%rdi)
vmovdqa         %ymm6,(64*\off+32)*2(%rdi)
vmovdqa         %ymm11,(64*\off+48)*2(%rdi)
.endm

.text
.global cdecl(basemul_avx)
cdecl(basemul_avx):
mov             %rsp,%r8
and             $-32,%rsp
sub             $32,%rsp

lea             (_ZETAS_EXP+176)*2(%rcx),%r9
schoolbook      0

add             $32*2,%r9
schoolbook      1

add             $192*2,%r9
schoolbook      2

add             $32*2,%r9
schoolbook      3

mov             %r8,%rsp
ret
```

fq.S

```
.global cdecl(reduce_avx)
cdecl(reduce_avx):
#consts
vmovdqa         _16XQ*2(%rsi),%ymm0
vmovdqa         _16XV*2(%rsi),%ymm1
call            reduce128_avx
add             $256,%rdi
call            reduce128_avx
ret

tomont128_avx:
#load
vmovdqa         (%rdi),%ymm3
vmovdqa         32(%rdi),%ymm4
vmovdqa         64(%rdi),%ymm5
vmovdqa         96(%rdi),%ymm6
vmovdqa         128(%rdi),%ymm7
vmovdqa         160(%rdi),%ymm8
vmovdqa         192(%rdi),%ymm9
vmovdqa         224(%rdi),%ymm10

fqmulprecomp    1,2,3,11
fqmulprecomp    1,2,4,12
fqmulprecomp    1,2,5,13
fqmulprecomp    1,2,6,14
fqmulprecomp    1,2,7,15
fqmulprecomp    1,2,8,11
fqmulprecomp    1,2,9,12
fqmulprecomp    1,2,10,13

#store
vmovdqa         %ymm3,(%rdi)
vmovdqa         %ymm4,32(%rdi)
vmovdqa         %ymm5,64(%rdi)
vmovdqa         %ymm6,96(%rdi)
vmovdqa         %ymm7,128(%rdi)
vmovdqa         %ymm8,160(%rdi)
vmovdqa         %ymm9,192(%rdi)
vmovdqa         %ymm10,224(%rdi)

ret
```

- The detail implementation written in assembly
  - Architecture specific
  - cannot use with HSL
- Author use this code as reference → rewrite the algorithm in C
- Then put the C code into HLS → Verilog

note : the translated code in github is no longer available

Results

Table I: Results of accelerator performance

RTL	Latency (clock cycles)	Interval (clock cycles)	Total execution time (clock cycles)
Verilog	6983	6983	83804

Table II: FPGA Resources utilization

Resource	Available	Utilization	%
LUT	53200	2200	4.14
LUTRAM	17400	215	1.24
FF	106400	3001	2.82
BRAM	140	3.50	2.50
DSP	220	28	12.73

Table IV: Comparison with previous works

Related Works	Function	HLS	LUT	FF	DSP	BRAM	Freq (MHz)
[6]	Enc / Dec	No	110260	-	292	202	155
[7]	KeyGen / Enc / Dec	No	7412	4644	2	3	161
[8]	KeyGen / Enc / Dec	No	16000	6000	9	16	115
[9]	NTT	No	801	717	4	2	222
[12][2]	Enc / Dec	Yes	1977896	194126	-	-	-
[13]	NTT / PM[1]	No	9508	-	16	35	172
[14]	NTT / PM[1]	No	5181	4833	16	-	227
Our	PM[1]	Yes	2200	3001	28	3.5	100

[1] Polynomial Multiplication
[2] Kyber512 version

- @100MHz execution time = 0.84 ms (best =0.47ms, worst = 192.89 ms)
- Use DSP slice the highest (special unit for arithmetic operations)
- Don't use much fpga resources
- Comparison table is not meaningful because different number of parts implement on fpga, different CLK fpga

Conclusion & Future work

Topics I found interesting, and will research further

- Montgomery Multiplications → Faster modular multiplications : (A * B) mod q
- Different accelerator architecture : UART TX, RX pin on FPGA
- Reference assembly code from Kyber official github

Next presentation : Verilog implementation on fpga, start from polynomial multiplication

THANK YOU