Sabancı University Program for Undergraduate Research (PURE) Spring 2019-2020

Hardware Design for Post-Quantum Cryptography

Hüseyin Kaya kayahuseyin@sabanciuniv.edu *Electronics Engineering/FENS, 2018*

Doğan Can Hasanoğlu hasanogludogan@sabanciuniv.edu *Computer Science & Engineering /FENS, 2018*

Bora Mert Karal boramert@sabanciuniv.edu *Computer Science & Engineering /FENS, 2017*

Supervisor: **Erdinç Öztürk -** Electronics Engineering

Abstract: In the world of Cryptography most of the scientists and engineers are working to come up with a better solution to secure the data. There is an on-going competition for Post Quantum Cryptography Standardization and hardware performance will be one of the determining factors for this competition. In this project it is aimed to implement the Kyber cryptographic scheme posted on NIST second round submissions for post-quantum cryptography on an FPGA. Multiplying two polynomials with enormous coefficients is expensive. This is why it is necessary to use an efficient method, number theoretic transform. We are in need of further development in the hardware part of the solutions in order to keep up with the quantum world which current software security may not be enough hereby we were able to have a better understanding of the hardware process that is being implemented

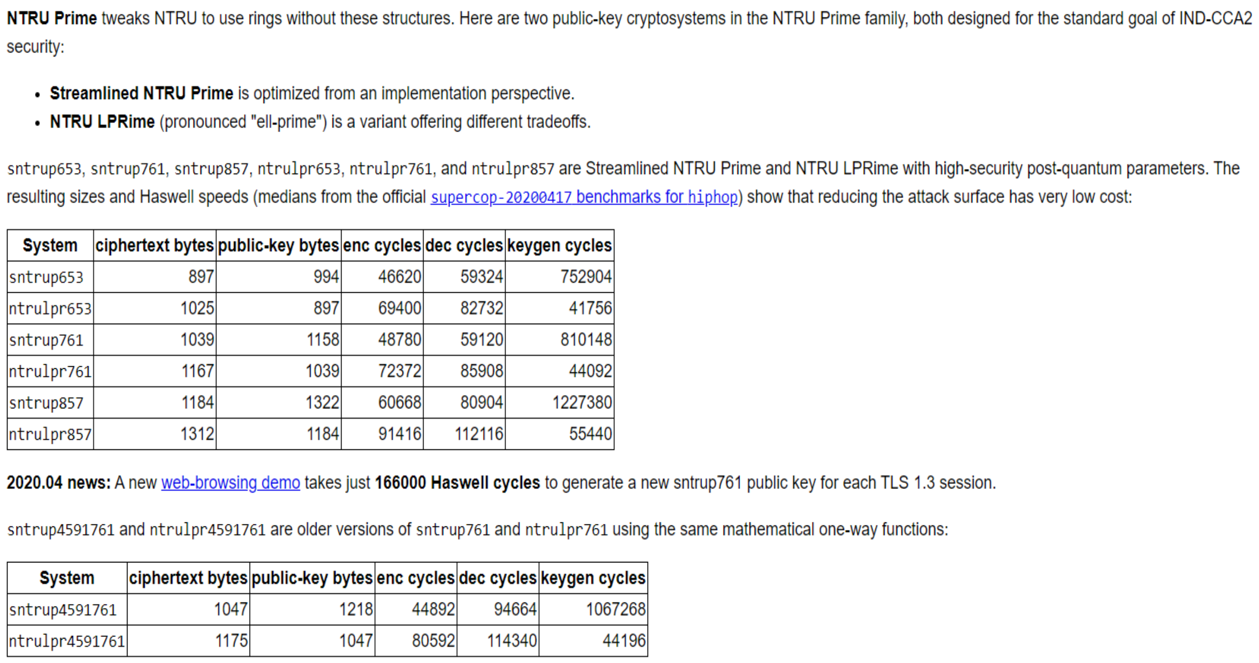
1- Introduction

In the world of Cryptography most of the scientists and engineers are working to come up with a better solution to secure the data. Most of the time which is the easiest way to do it is to work with software solution concerning math and algorithms .What we aim to do is to design hardware solutions to the developing process.This way of solutions are generally not sufficient for the high speed attacks such as as the title states quantum computers and computation power.

2.1-Cryptographic Algorithms: There is an on-going competition for Post Quantum

Cryptography Standardization and hardware performance will be one of the determining factors for this competition. Efficient implementations will be very critical in the near future. Constructions based on Lattice-Based cryptography are looking the most promising at the moment. In the following subsections, NIST 2nd round algorithms will be investigated.

2.1.1- NTRU Prime:NTRU is an open source public-key cryptosystem that uses lattice-based cryptography to encrypt and decrypt data.)Here it is explained that ideal lattice based systems were not sufficient enough that this approach had to be established called NTRU Prime It talks about the rings and NTRU Prime But the system uses rings without classic NTRU cryptosystem and typical Ring-LWE-based cryptosystems



NTRU Prime tweaks NTRU to use rings without these structures. Here are two public-key cryptosystems in the NTRU Prime family, both designed for the standard goal of IND-CCA2 security:

- **Streamlined NTRU Prime** is optimized from an implementation perspective.
- **NTRU LPRime** (pronounced "ell-prime") is a variant offering different tradeoffs.

`sntrup653`, `sntrup761`, `sntrup857`, `ntrulpr653`, `ntrulpr761`, and `ntrulpr857` are Streamlined NTRU Prime and NTRU LPRime with high-security post-quantum parameters. The resulting sizes and Haswell speeds (medians from the official `supercop-20200417` benchmarks for `hiphop`) show that reducing the attack surface has very low cost:

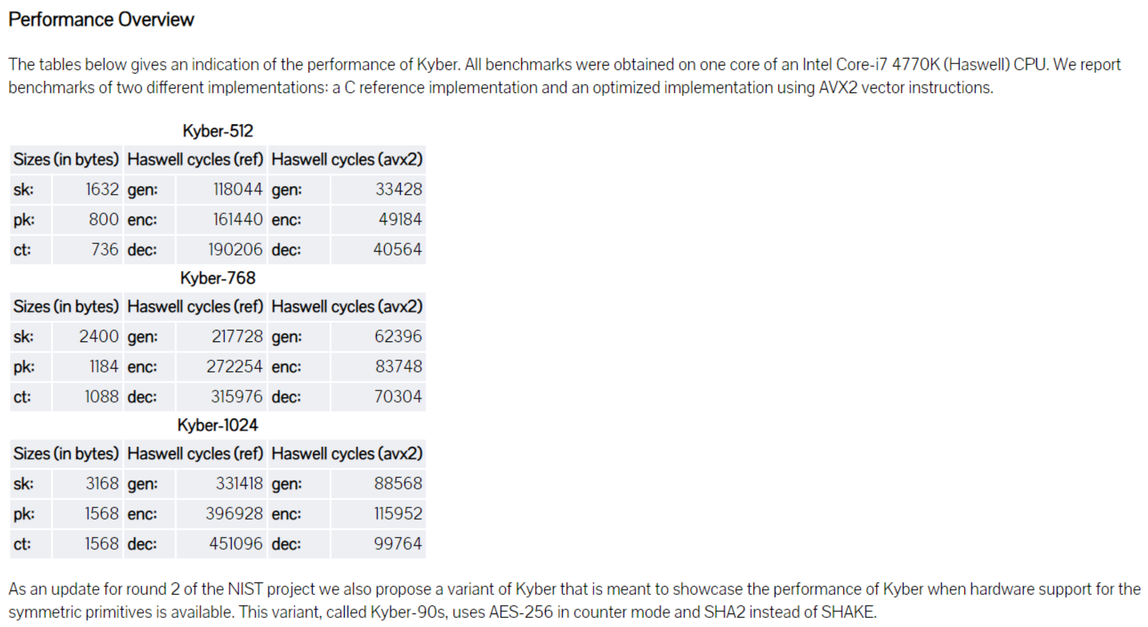
System	ciphertext bytes	public-key bytes	enc cycles	dec cycles	keygen cycles
sntrup653	897	994	46620	59324	752904
ntrulpr653	1025	897	69400	82732	41756
sntrup761	1039	1158	48780	59120	810148
ntrulpr761	1167	1039	72372	85908	44092
sntrup857	1184	1322	60668	80904	1227380
ntrulpr857	1312	1184	91416	112116	55440

2020.04 news: A new web-browsing demo takes just **166000 Haswell cycles** to generate a new sntrup761 public key for each TLS 1.3 session.

`sntrup4591761` and `ntrulpr4591761` are older versions of `sntrup761` and `ntrulpr761` using the same mathematical one-way functions:

System	ciphertext bytes	public-key bytes	enc cycles	dec cycles	keygen cycles
sntrup4591761	1047	1218	44892	94664	1067268
ntrulpr4591761	1175	1047	80592	114340	44196

2.1.2- Kyber:



Performance Overview

The tables below gives an indication of the performance of Kyber. All benchmarks were obtained on one core of an Intel Core-i7 4770K (Haswell) CPU. We report benchmarks of two different implementations: a C reference implementation and an optimized implementation using AVX2 vector instructions.

Kyber-512

Sizes (in bytes)		Haswell cycles (ref)		Haswell cycles (avx2)	
sk:	1632	gen:	118044	gen:	33428
pk:	800	enc:	161440	enc:	49184
ct:	736	dec:	190206	dec:	40564

Kyber-768

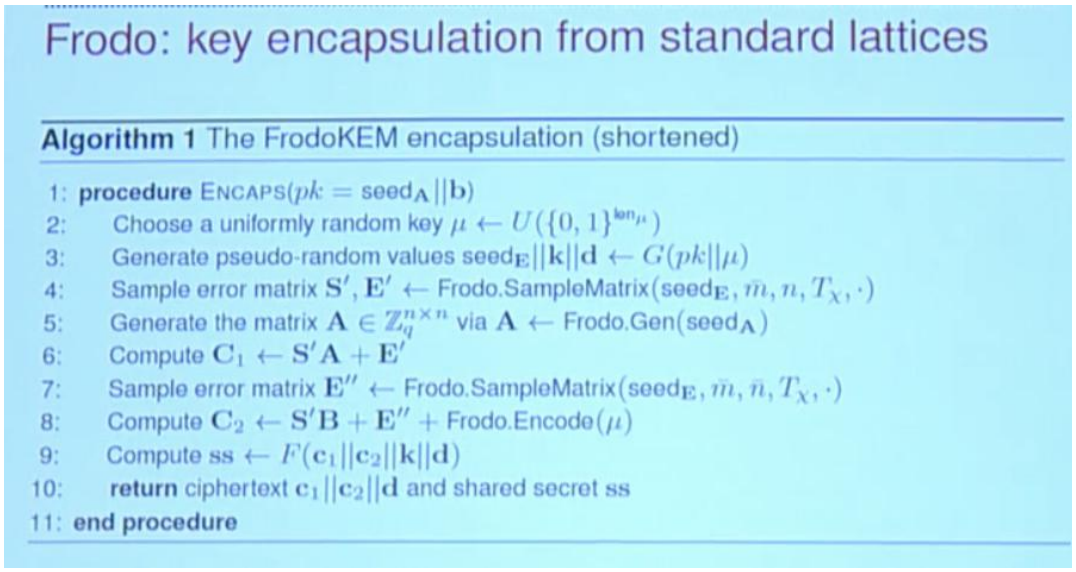
Sizes (in bytes)		Haswell cycles (ref)		Haswell cycles (avx2)	
sk:	2400	gen:	217728	gen:	62396
pk:	1184	enc:	272254	enc:	83748
ct:	1088	dec:	315976	dec:	70304

Kyber-1024

Sizes (in bytes)		Haswell cycles (ref)		Haswell cycles (avx2)	
sk:	3168	gen:	331418	gen:	88568
pk:	1568	enc:	396928	enc:	115952
ct:	1568	dec:	451096	dec:	99764

As an update for round 2 of the NIST project we also propose a variant of Kyber that is meant to showcase the performance of Kyber when hardware support for the symmetric primitives is available. This variant, called Kyber-90s, uses AES-256 in counter mode and SHA2 instead of SHAKE.

2.1.3- Frodo:



Frodo: key encapsulation from standard lattices

Algorithm 1 The FrodoKEM encapsulation (shortened)

```
1: procedure ENCAPS($pk = \text{seed}_\mathbf{A} || \mathbf{b}$)
2:     Choose a uniformly random key $\mu \leftarrow U(\{0,1\}^{\text{len}_\mu})$
3:     Generate pseudo-random values $\text{seed}_\mathbf{E} || \mathbf{k} || \mathbf{d} \leftarrow G(pk || \mu)$
4:     Sample error matrix $\mathbf{S}', \mathbf{E}' \leftarrow$ Frodo.SampleMatrix($\text{seed}_\mathbf{E}, \overline{m}, n, T_\chi, \cdot$)
5:     Generate the matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ via $\mathbf{A} \leftarrow$ Frodo.Gen($\text{seed}_\mathbf{A}$)
6:     Compute $\mathbf{C}_1 \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
7:     Sample error matrix $\mathbf{E}'' \leftarrow$ Frodo.SampleMatrix($\text{seed}_\mathbf{E}, \overline{m}, \overline{n}, T_\chi, \cdot$)
8:     Compute $\mathbf{C}_2 \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}'' +$ Frodo.Encode($\mu$)
9:     Compute $\mathbf{ss} \leftarrow F(\mathbf{c}_1 || \mathbf{c}_2 || \mathbf{k} || \mathbf{d})$
10:    return ciphertext $\mathbf{c}_1 || \mathbf{c}_2 || \mathbf{d}$ and shared secret $\mathbf{ss}$
11: end procedure
```

2.2- Number Theoretical Transform:

The NTT is a generalization of the classic DFT to finite fields. With a lot of work, it basically lets one perform fast convolutions on integer sequences without any round-off errors, guaranteed. Convolutions are useful for multiplying large numbers or long polynomials, and the NTT is asymptotically faster than other methods like Karatsuba multiplication.

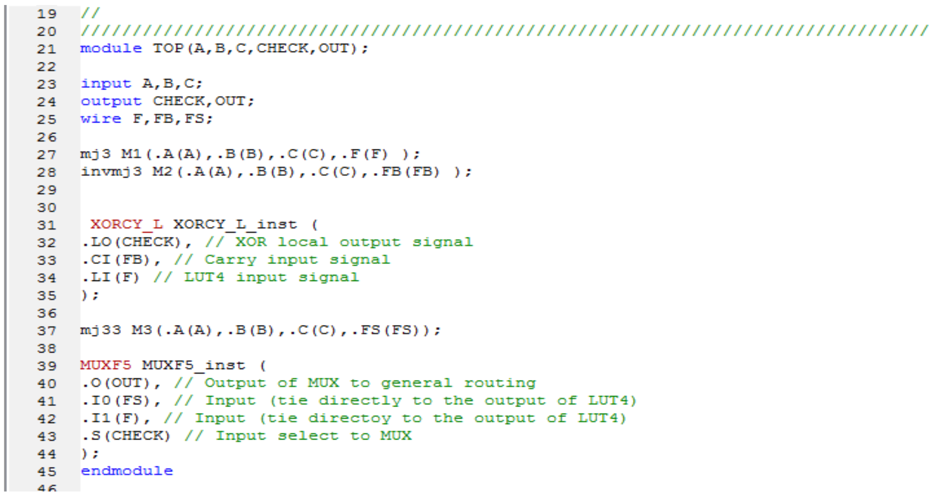
```
****************************************************************************************************
import random
import math
import sympy
#for generating psi
#calculate: G**E mod M
def expo_reference(G,E,M):
    if (E <= 1):
        lenbit = 1
    else:
        lenbit = int(math.log( E ) / math.log( 2 )) + 1
    T1 = 1;
    for i in range(lenbit):
        T1 = (T1*T1) % M
        ei = (E >> (lenbit-(i+1) )  )%2
        if (ei == 1):
            T1 = (T1*G)%M
    return T1


#reference polynomial multiplication function
def refpolmul(A,B,M):
    C = [0]*(2*len(A))
    D = [0]*(len(A))
    for indexA,elemA in enumerate(A):
        for indexB,elemB in enumerate(B):
            C[indexA+indexB] = (C[indexA+indexB] + elemA*elemB)%M

    for i in range (len(A)):
        D[i] = (C[i] - C[i+len(A)])%M
    return D

#
def NTTAlghw(X, Y, W, wordlen, verbose, L, l, Butter_index_sel):


    Z = [0]* L
    Zres = [0]* L
    for i in range (L):
        Z[i] = X[i]

    for i in range (1,l+1,1):
        m = (2**(l-i))
        for j in range (0, (2**(i-1)), 1):
            t = 2*j*m

            for k in range (0, m, 1):
                if (i==l):
                    T1 = (Z[t+k] + Z[t+k+m]) % Y
                    T2 =  (Z[t+k] - Z[t+k+m])% Y

                    Zres[Butter_index_sel[t+k]  ] = T1
                    Zres[Butter_index_sel[t+k+m]] = T2

                    if (verbose == 1):
                        print (i, j, k, t+k, t+k+m)
                        #print (t+k, t+k+m)

                else:
                    zitak = W[( 2**(i-1) )*k]
                    T1 = (Z[t+k] + Z[t+k+m]) % Y
                    T2 =  (zitak*(Z[t+k] - Z[t+k+m])) % Y

                    Z[t+k] = T1
                    Z[t+k+m] = T2

                    if (verbose == 1):
                        print (i, j, k, t+k, t+k+m, hex(zitak))
                        #print (t+k, t+k+m, ( 2**(i-1) )*k)
                        #print (i, ( 2**(i-1) )*k)


    return Zres








#multiply A with B

def nttpolmul (A,B,M,psi,psiinv,ninv, W, WINV, wordlen, L, l, Butter_index):
    sizeNTT = len(A)
    Abar = [0]*sizeNTT
    Bbar = [0]*sizeNTT
    psitemp = 1



    for i in range ( sizeNTT ):
        Abar[i]= (A[i]*psitemp) % M
        Bbar[i]= (B[i]*psitemp) % M
        psitemp = (psitemp*psi) % M


    A_NTT = NTTAlghw(Abar, M, W, wordlen, 0, L, l, Butter_index)
    B_NTT = NTTAlghw(Bbar, M, W, wordlen, 0, L, l, Butter_index)



    C_NTT = [0]*sizeNTT

    for i in range(sizeNTT):
        C_NTT[i] = (A_NTT[i]*B_NTT[i]) % M

    C = NTTAlghw(C_NTT, M, WINV, wordlen, 0, L, l, Butter_index)

    psiinvtemp = 1
    for i in range(sizeNTT):
        C[i] = (C[i]*psiinvtemp*ninv) % M
        psiinvtemp = (psiinvtemp * psiinv) % M
    return C

##########################################

primegen = 1
random.seed(0)

#construct NTT parameters
l = 4
L = 2**l
wordlen = 20
primelen = 20

if (primegen == 1):
    while (1):
        q = random.randint(2**(primelen-1)+1,2**primelen-1)
        q = (q & ( 2**primelen - 2**(l+1) ))+1
        if (sympy.isprime(q) == True):
            if (q%(2*L) == 1):
                break
else:
    q = 0xc0001

#generate a 2nth root of unity
exp = (q-1)//(2*L)
alpha = 1
while (1):
    alpha = alpha + 2
    psi = expo_reference(alpha,exp,q)

    for i in range (2*L):
        T1 = expo_reference(psi,i+1,q)
        if (T1==1):
            if(i != 2*L-1):
                break
    if (i == 2*L-1):
        break

#psi = 19
print ("psi = ", psi )
```

```
#checkpoint
T2 = expo_reference(psi,2*L,q)
if (T2 != 1):
    print ("Something wrong with your math for psi")

psiinv = expo_reference(psi,2*L-1,q)

#checkpoint
T2 = (psi*psiinv) % q
if (T2 != 1):
    print ("Something wrong with your math for psiinv")


Linv = expo_reference(L,q-2,q)
#checkpoint
T2 = (L*Linv) % q
if (T2 != 1):
    print ("Something wrong with your math for ninv")


w = (psi*psi) % q
print ("w  = ", w )
winv = (psiinv*psiinv) % q

W = [0]*L
WINV = [0]*L
T1 = 1
T2 = 1
for i in range ( L//2 ):
    W[i] = T1
    WINV[i] = T2
    T1 = T1*w % q
    T2 = T2*winv % q

#for i in range( L//2 ):
#    if (W[i] == 0xA7948):
#        print ("W[",i,"]=",hex(W[i]))

Butter_index = [0]*L
Butter_index_new = [0]*L

for i in range (L):
    Butter_index[i] = i

for i in range (l-1):
    numiterin = (L//2) // (2**i)
    numiterout = (2**i)
    for j in range (numiterout):
        for k in range (numiterin):
            Butter_index_new[k + j*2*numiterin] = Butter_index[2*k + j*2*numiterin]
            Butter_index_new[k + j*2*numiterin + numiterin] = Butter_index[2*k + j*2*numiterin
+ 1]


    for j in range (L):
        Butter_index[j] = Butter_index_new[j]
```

```
#generate random polynomials to multiply
A = []
B = []
for i in range( L ):
    A.append(random.randint(2**(primelen-1)+1,2**primelen-1) % q)
    B.append(random.randint(2**(primelen-1)+1,2**primelen-1) % q)




Cref = refpolmul(A, B, q)

CNTT = nttpolmul (A,B,q,psi,psiinv,Linv, W, WINV, wordlen, L, l, Butter_index)

for i in range( len(Cref) ):
    if (Cref[i] != CNTT[i] ):
        print ("Wrong Mul")
        break

if (i == len(Cref)-1):
    print ("Correct Mul")
else:
    print ("Wrong Mul")


**************************************************************************
```

2.3- Verilog :

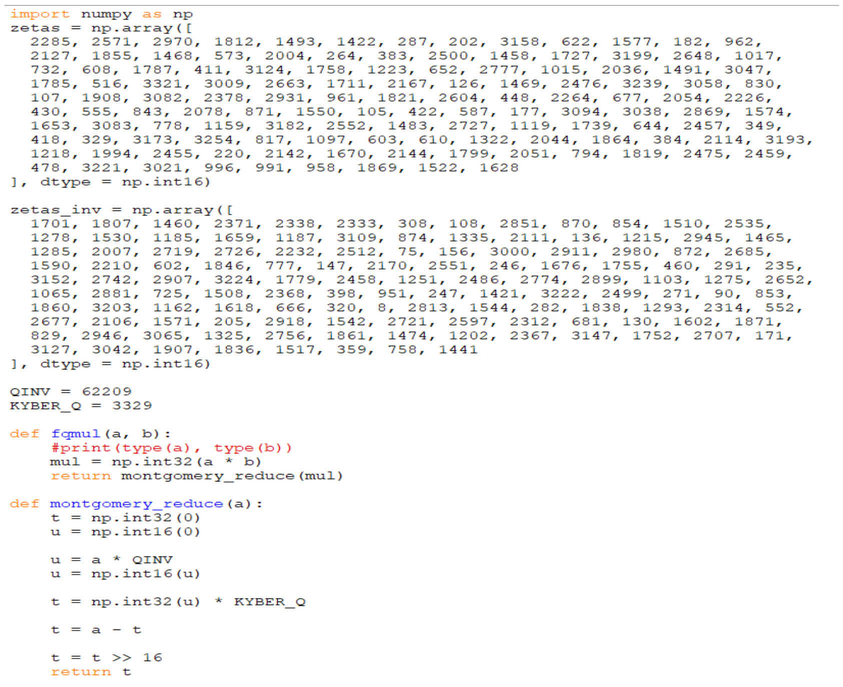
Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits, as well as in the design of genetic circuits. It provides the infrastructure for the circuits involved in post-quantum cryptography and performs visualizations for design and implementation.



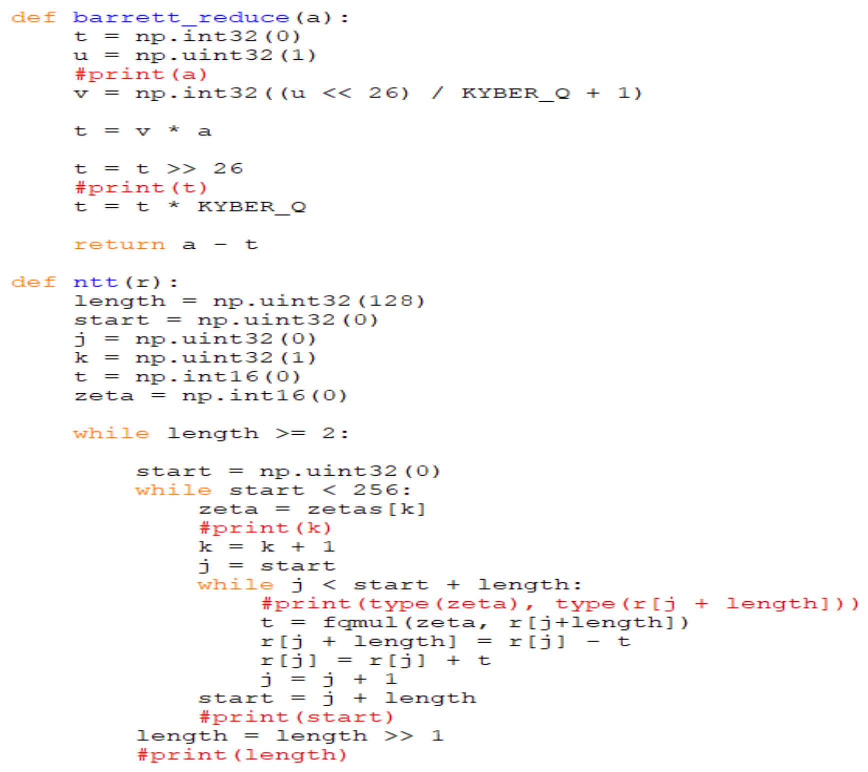
```verilog
//
////////////////////////////////////////////////////////////////////////////////////
module TOP(A,B,C,CHECK,OUT);

input A,B,C;
output CHECK,OUT;
wire F,FB,FS;

mj3 M1(.A(A),.B(B),.C(C),.F(F) );
invmj3 M2(.A(A),.B(B),.C(C),.FB(FB) );


 XORCY_L XORCY_L_inst (
.LO(CHECK), // XOR local output signal
.CI(FB), // Carry input signal
.LI(F) // LUT4 input signal
);

mj33 M3(.A(A),.B(B),.C(C),.FS(FS));

MUXF5 MUXF5_inst (
.O(OUT), // Output of MUX to general routing
.I0(FS), // Input (tie directly to the output of LUT4)
.I1(F), // Input (tie directoy to the output of LUT4)
.S(CHECK) // Input select to MUX
);
endmodule
```

2.4- Hardware Implementation of Kyber:

Kyber is a lattice based scheme. In this project it is aimed to implement the Kyber cryptographic scheme posted on NIST second round submissions for post-quantum cryptography on an FPGA. In order to implement, It is needed to understand the fundamental operation of lattice-based cryptographic schemes, polynomial multiplication. However, multiplying two polynomials with enormous coefficients is expensive. This is why it is necessary to use an efficient method, number theoretic transform. NTT functions are implemented in Python3 from the reference implementation of Kyber.



```python
import numpy as np
zetas = np.array([
  2285, 2571, 2970, 1812, 1493, 1422, 287, 202, 3158, 622, 1577, 182, 962,
  2127, 1855, 1468, 573, 2004, 264, 383, 2500, 1458, 1727, 3199, 2648, 1017,
  732, 608, 1787, 411, 3124, 1758, 1223, 652, 2777, 1015, 2036, 1491, 3047,
  1785, 516, 3321, 3009, 2663, 1711, 2167, 126, 1469, 2476, 3239, 3058, 830,
  107, 1908, 3082, 2378, 2931, 961, 1821, 2604, 448, 2264, 677, 2054, 2226,
  430, 555, 843, 2078, 871, 1550, 105, 422, 587, 177, 3094, 3038, 2869, 1574,
  1653, 3083, 778, 1159, 3182, 2552, 1483, 2727, 1119, 1739, 644, 2457, 349,
  418, 329, 3173, 3254, 817, 1097, 603, 610, 1322, 2044, 1864, 384, 2114, 3193,
  1218, 1994, 2455, 220, 2142, 1670, 2144, 1799, 2051, 794, 1819, 2475, 2459,
  478, 3221, 3021, 996, 991, 958, 1869, 1522, 1628
], dtype = np.int16)

zetas_inv = np.array([
  1701, 1807, 1460, 2371, 2338, 2333, 308, 108, 2851, 870, 854, 1510, 2535,
  1278, 1530, 1185, 1659, 1187, 3109, 874, 1335, 2111, 136, 1215, 2945, 1465,
  1285, 2007, 2719, 2726, 2232, 2512, 75, 156, 3000, 2911, 2980, 872, 2685,
  1590, 2210, 602, 1846, 777, 147, 2170, 2551, 246, 1676, 1755, 460, 291, 235,
  3152, 2742, 2907, 3224, 1779, 2458, 1251, 2486, 2774, 2899, 1103, 1275, 2652,
  1065, 2881, 725, 1508, 2368, 398, 951, 247, 1421, 3222, 2499, 271, 90, 853,
  1860, 3203, 1162, 1618, 666, 320, 8, 2813, 1544, 282, 1838, 1293, 2314, 552,
  2677, 2106, 1571, 205, 2918, 1542, 2721, 2597, 2312, 681, 130, 1602, 1871,
  829, 2946, 3065, 1325, 2756, 1861, 1474, 1202, 2367, 3147, 1752, 2707, 171,
  3127, 3042, 1907, 1836, 1517, 359, 758, 1441
], dtype = np.int16)

QINV = 62209
KYBER_Q = 3329

def fqmul(a, b):
    #print(type(a), type(b))
    mul = np.int32(a * b)
    return montgomery_reduce(mul)

def montgomery_reduce(a):
    t = np.int32(0)
    u = np.int16(0)

    u = a * QINV
    u = np.int16(u)

    t = np.int32(u) * KYBER_Q

    t = a - t

    t = t >> 16
    return t
```



```
def barrett_reduce(a):
    t = np.int32(0)
    u = np.uint32(1)
    #print(a)
    v = np.int32((u << 26) / KYBER_Q + 1)

    t = v * a

    t = t >> 26
    #print(t)
    t = t * KYBER_Q

    return a - t

def ntt(r):
    length = np.uint32(128)
    start = np.uint32(0)
    j = np.uint32(0)
    k = np.uint32(1)
    t = np.int16(0)
    zeta = np.int16(0)

    while length >= 2:

        start = np.uint32(0)
        while start < 256:
            zeta = zetas[k]
            #print(k)
            k = k + 1
            j = start
            while j < start + length:
                #print(type(zeta), type(r[j + length]))
                t = fqmul(zeta, r[j+length])
                r[j + length] = r[j] - t
                r[j] = r[j] + t
                j = j + 1
            start = j + length
            #print(start)
        length = length >> 1
        #print(length)
```

3- Discussion and Conclusion:

Post-Quantum Cryptography is a promising field and its prominence has not come out yet still it will be discovered soon when quantum computers are involved in our daily lives. To conclude, we are in need of further development in the hardware part of the solutions in order to keep up with the quantum world which current software security may not be enough. By the research conducted we were able to have a better understanding of the hardware process that is being implemented. More than that this research project aims not only to approach the topic with this end and its up and coming aspect but also it strives for coming up with assistive discoveries for further research.We aim to further develop circuitry design using verilog. By proceeding the work in this project, it is expected to design a hardware and implement the studied cryptographic algorithms. Last but not least we are looking forward to working this project in days to come with the same team and our supervisor since as far as we believe we have tried our best and built some convenient understanding of the subject and came up with practical results.

References

- https://www.nayuki.io/page/number-theoretic-transform-integer-dft
- https://www.intel.cn/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf
- https://www.microsoft.com/en-us/research/project/microsoft-seal/#!publications
- https://www.microsoft.com/en-us/research/wp-content/uploads/2017/06/sealmanual_v2.2.pdf
- https://dblp.org/pers/=/=Ouml=zt=uuml=rk:Erdin=ccedil=.html
- Erdinç Öztürk-https://fens.sabanciuniv.edu/tr/faculty-members/detail/2659

Source Codes:

- https://ntruprime.cr.yp.to/index.html
- https://github.com/pq-crystals/kyber/blob/master/ref/ntt.c
- https://pq-crystals.org/
- https://www.microsoft.com/en-us/research/project/microsoft-seal/