PARS: Design Space Exploration for Low-Latency Ring-LWE via HLS

Eskisehir Technical University
List of group members: Mehmet Atakan Kalkar Adem Savaş
Supervisor = Assist. Prof. Dr. İsmail San
xohw20_170 - https://www.youtube.com/watch?v=wVZGrSAV1GI&feature=youtu.be

Abstract

Today's public key cryptographic algorithms are based on mainly three difficult mathematical problems: the integer factorization problem, the discrete logarithm problem, or the elliptic-curve discrete logarithm problem. Modern public key cryptosystems are heavily relying on them that perform operations over large integers, e.g., 1024-bit. These problems can be solved with Shor's algorithm by using a quantum computer with a million of qubits which have enough computational power to break such difficult problems that operate over large integers. However, present day quantum computers do not have enough qubits for necessary computational power to break any real cryptosystem. Even though quantum computers with such a big computing power have not been produced yet, these computers are expected to be produced in the near future. Therefore, today's cryptographic algorithms will be under threat when such complex quantum computers are invented. Scientists have already started to study on new quantum-resistant cryptographic algorithms for the post-quantum era. There are a few difficult mathematical primitives to construct a quantum-resistant cryptosystem and ring learning with error (Ring-LWE) is one of them which is based on lattice-based cryptography. Ring-LWE is promising thanks to its computational efficiency compared to other available counterparts. However, it consists of computationally heavy forward and inverse number theoretical transform and a lot of integer multiplications and additions. Considering available hard integer multipliers and adders within DSP blocks inside Xilinx FPGAs, the computational speed of the Ring-LWE can be enhanced by exploiting the DSP blocks with a custom special hardware architecture. In this study, our objective is to explore the design space for efficient hardware architectures of Ring-LWE based encryption and decryption. Designing hardware architectures at register transfer level (RTL) level using VHDL or Verilog is a very slow design process. So, exploring different hardware architectures at RTL level requires a lot of design efforts. However, design space exploration is easier through a high-level synthesis (HLS) tool. Besides, design and verification via HLS are orders of magnitude faster than at the RTL level. In this study, we will use Vivado HLS tool to describe our hardware architectures for Ring-LWE from a high-level language, C/C++. This will allow us to explore different hardware designs of Ring-LWE encryption and decryption under many different design parameters and constraints. The efficient hardware accelerators will then be integrated to a system-on-chip (SoC) architecture to create an application that encrypts, and decrypts texts stored in the DDR off-chip memory. In this regard, low-cost and low-power SoC solution provides us faster and more secure data transfer with a better security. Vivado HLS tool and ZedBoard (Xilinx Zynq-7000 SoC Architecture) will be used for this project.

1. Introduction

In the long run, achieving resistance against quantum attacks is vital because there are still devices based on classical architectures. The Reason that we choose that topic for our project is the usability of HLS on cryptographic applications is largely unexplored, unlike deep learning and digital signal processing applications. However, the acceleration of cryptographic applications is as important as the acceleration of deep learning and digital signal processing applications. Post-quantum cryptography provides an excellent case study to evaluate the HLS for next-generation public-key cryptosystem design. Studies have been done about HLS acceleration of quantum cryptography applications show that the performance of the cryptography algorithms can be improved. So, our main objective is to realize a post-quantum cryptography problem on hardware with less power consumption, reliable and high performance with minimum latency compare to pure software. The main problems we focus on solving are to identify functions that are called too many times and require a long time with software profiling. Then, substitute these functions of the algorithm to obtain more memory efficient with less latency and HLS compatible version of the algorithm. Another problem that we focused on is experimenting with HLS optimization directives and HLS configurations like the target clock period for exploring the effects in terms of utilization and latency.

2. Design and Method

RING LEARNING WITH ERRORS (RING-LWE)

Lattice-based cryptography is considered a significant candidate for quantum-secure public key cryptography due to its wide applicability. Ring-LWE problem is one of the popular lattice-based cryptography algorithms. Ring-LWE uses polynomial arithmetic with Number Theoretical Transform (NTT) as a core function. NTT can be considerable as Fast Fourier Transform which operated in the integer domain. The Ring-LWE cryptosystem requires samples from a discrete Gaussian distribution to construct the error polynomials during the key generation and encryption operation. Accordingly, Ring-LWE uses the Knuth-Yao algorithm[1] to sample from a discrete Gaussian distribution. In this study, Ruan de Clercq's open-source GitHub repository was used[2]. This repository allows us 128-bit and 256-bit security with specified parameters. We concentrated on 128-bit security with 256 number of coefficients, 7681 as modulus and sigma as 11,31/sqrt(2*pi) for this study. **Figure 1** shows the block diagram of Ring-LWE encryption and decryption. In this diagram Gaussian Sampler part indicates the Knuth-Yao algorithm, Encoder part indicates the Forward NTT algorithm, and the Decoder part indicates the inverse NTT algorithm. The flow of the algorithm can be seen easily in **figure 2**. We examined our entire work in 5 different phases. **Figure 3** shows all phases that we will mention.

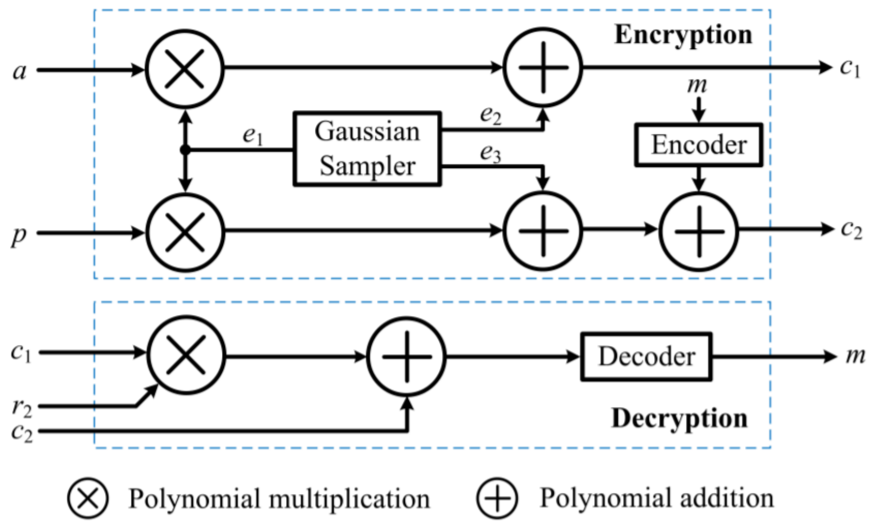


Figure 1. Block diagram of Ring Learning Errors[3].

Algorithm 2: Ring-learning with errors (LWE) encryption algorithm.

```
   Input  : $a, q \in \mathbb{Z}_q, m \in \{0,1\}^n, \omega_n \in \mathbb{Z}_q^n$
   Output: $c_1, c_2 \in \mathbb{Z}_q^n$
1  Gaussian Sampler
2    $e_1 \leftarrow$ Gaussian-Sampler$(n, q, \sigma)$
3    $e_2 \leftarrow$ Gaussian-Sampler$(n, q, \sigma)$
4    $e_3 \leftarrow$ Gaussian-Sampler$(n, q, \sigma)$
5  Encoder
6  for $i = 0 : n-1$ do
7  |   $m_e[i] \leftarrow \lfloor \frac{q}{2} \rfloor \times m[i]$
8  end
9  Number Theoretic Transform
10   $a \leftarrow$ NTT$(a, \omega_n)$
11   $p \leftarrow$ NTT$(p, \omega_n)$
12   $e_1 \leftarrow$ NTT$(e_1, \omega_n)$
13   $e_2 \leftarrow$ NTT$(e_2, \omega_n)$
14   $e_3 \leftarrow$ NTT$(e_3, \omega_n)$
15 Cipher-text computation
16   Multiplication in parallel
17     $c_{10} \leftarrow Mult1_{NTT}(a, e_1)$
18     $c_{20} \leftarrow Mult2_{NTT}(p, e_1)$
19   Addition in parallel
20     $c_{21} \leftarrow Add1(m_e, e_3)$
21     $c_1 \leftarrow Add2(c_{10}, e_2)$
22     $c_2 \leftarrow Add3(c_{20}, c_{21})$
23 Return$(c_1, c_2)$
```

Algorithm 3: Ring-LWE decryption algorithm.

```
   Input  : $c_1, c_2, r_2 \in \mathbb{Z}_q^n; \omega_n, \omega_n^{-1} \in \mathbb{Z}_q$
   Output: Original message $m$
1  Number Theoretic Transform
2    $c_1 \leftarrow$ NTT$(c_1, \omega_n)$
3    $r_2 \leftarrow$ NTT$(r_2, \omega_n)$
4  Ring computation
5    $m_{d1} \leftarrow Mult3_{NTT}(c_1, r_2)$
6    $m_d \leftarrow Add4(m_{d1}, c_2)$
7  Decoder
8  for $i = 0 : n-1$ do
9  |   if $(\lfloor \frac{q}{4} \rfloor) \leq m_d[i] \leq 3 \times \lfloor \frac{q}{4} \rfloor)$ then
10 |   |   $m[i] = 1;$
11 |   else
12 |   |   $m[i] = 0;$
13 |   end
14 end
15 Return$(m)$
```

Figure 2. Pseudocode algorithms of Ring-LWE Encryption and Decryption[3].

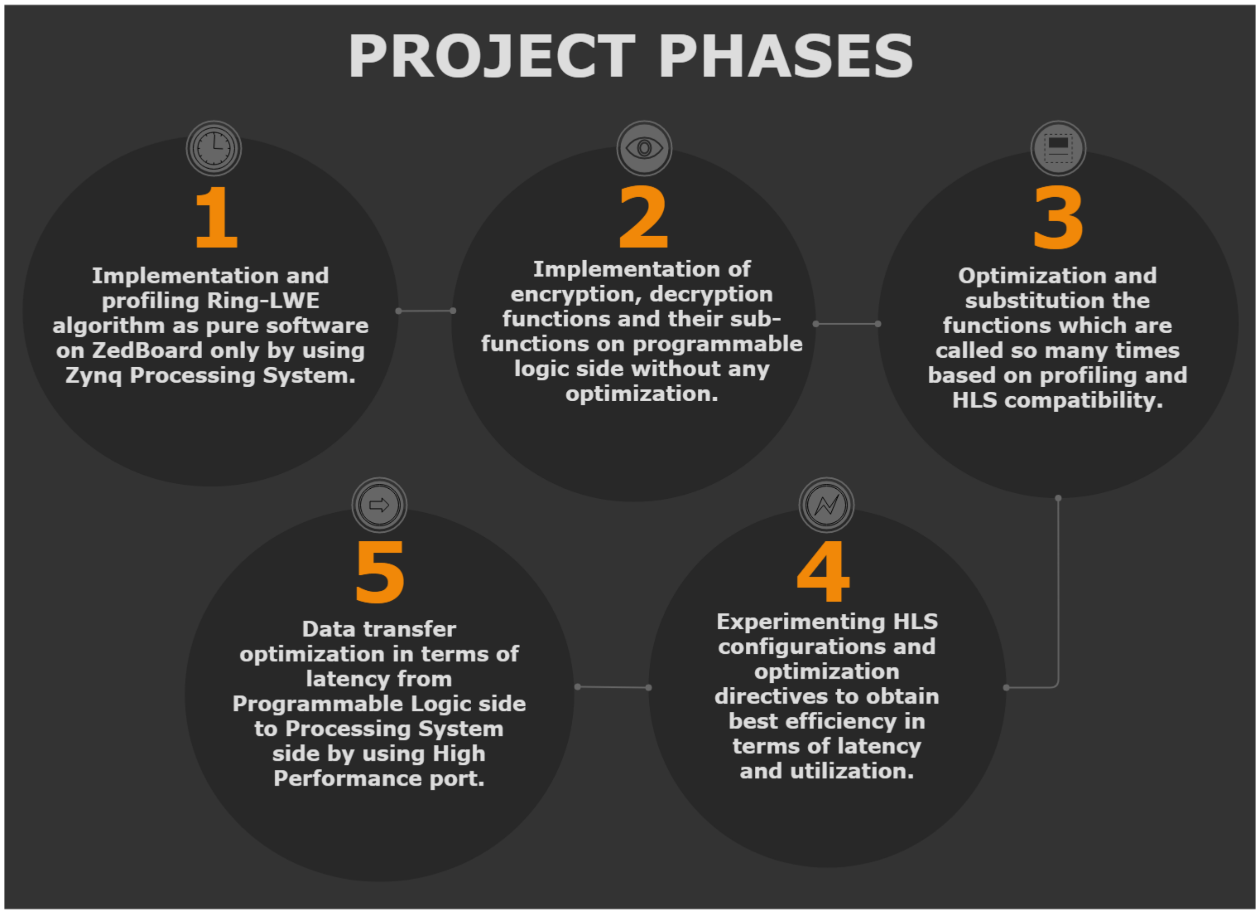
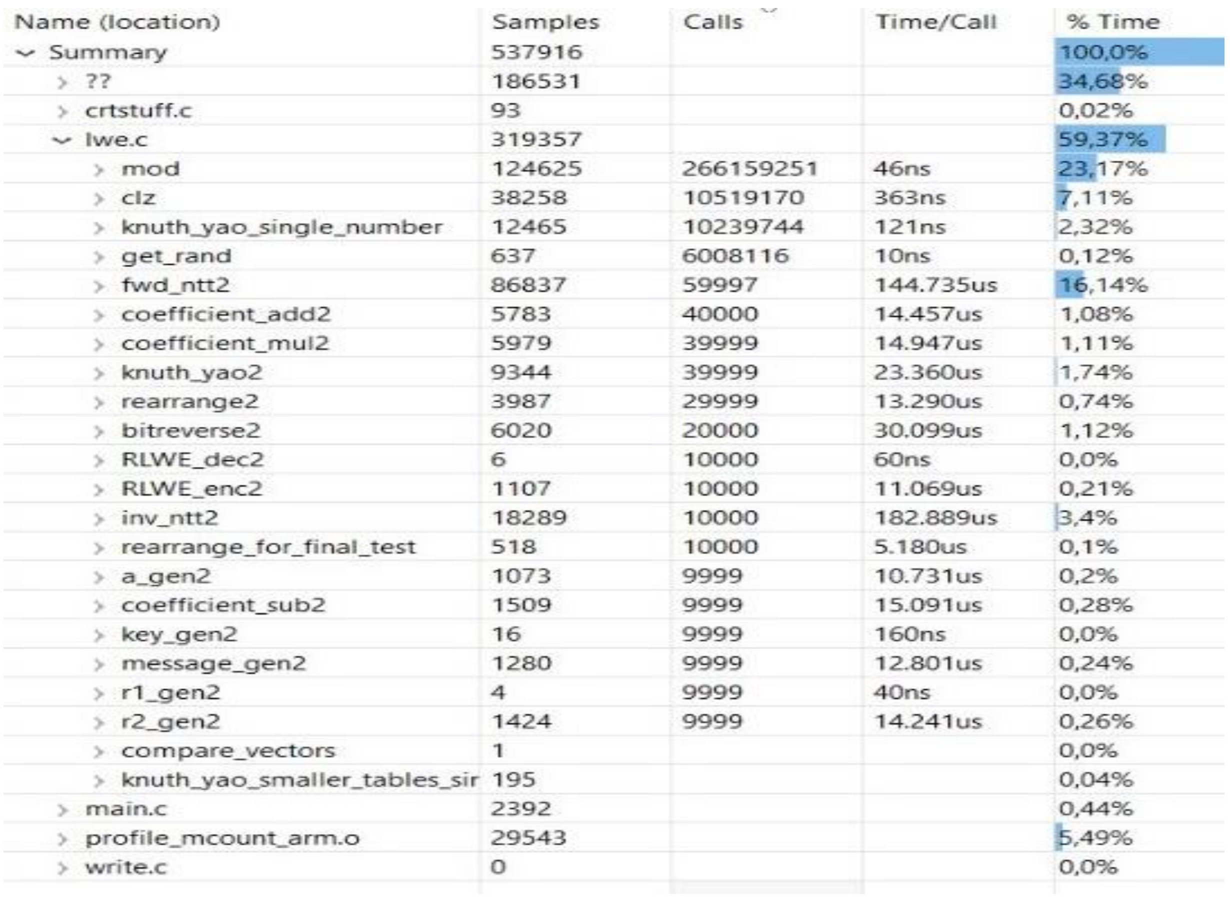

PROJECT PHASES
1
Implementation and profiling Ring-LWE algorithm as pure software on ZedBoard only by using Zynq Processing System.
2
Implementation of encryption, decryption functions and their sub-functions on programmable logic side without any optimization.
3
Optimization and substitution the functions which are called so many times based on profiling and HLS compatibility.
4
Experimenting HLS configurations and optimization directives to obtain best efficiency in terms of latency and utilization.
5
Data transfer optimization in terms of latency from Programmable Logic side to Processing System side by using High Performance port.

Figure 3. Phase diagram of workflow

2.1.Phase 1

The application of the algorithm was executed on Zynq Processing System as pure software. The main part of the algorithm consists of different sections that test the required functions for the algorithm. Also, software profiling was done on Xilinx SDK to determine the execution times of each routine. Thanks to the information which we obtained from profiling; we could determine critical pieces of the algorithm. As seen from **figure 4**, some functions are called by the algorithm so many times and they make up most of the execution time.



Name (location)	Samples	Calls	Time/Call	% Time
Summary	537916			100,0%
??	186531			34,68%
crtstuff.c	93			0,02%
lwe.c	319357			59,37%
mod	124625	266159251	46ns	23,17%
clz	38258	10519170	363ns	7,11%
knuth_yao_single_number	12465	10239744	121ns	2,32%
get_rand	637	6008116	10ns	0,12%
fwd_ntt2	86837	59997	144.735us	16,14%
coefficient_add2	5783	40000	14.457us	1,08%
coefficient_mul2	5979	39999	14.947us	1,11%
knuth_yao2	9344	39999	23.360us	1,74%
rearrange2	3987	29999	13.290us	0,74%
bitreverse2	6020	20000	30.099us	1,12%
RLWE_dec2	6	10000	60ns	0,0%
RLWE_enc2	1107	10000	11.069us	0,21%
inv_ntt2	18289	10000	182.889us	3,4%
rearrange_for_final_test	518	10000	5.180us	0,1%
a_gen2	1073	9999	10.731us	0,2%
coefficient_sub2	1509	9999	15.091us	0,28%
key_gen2	16	9999	160ns	0,0%
message_gen2	1280	9999	12.801us	0,24%
r1_gen2	4	9999	40ns	0,0%
r2_gen2	1424	9999	14.241us	0,26%
compare_vectors	1			0,0%
knuth_yao_smaller_tables_sir	195			0,04%
main.c	2392			0,44%
profile_mcount_arm.o	29543			5,49%
write.c	0			0,0%

Figure 4. Profiling result of Encryption/Decryption section of the algorithm.

2.2. Phase 2

Encryption/Decryption section of the main algorithm takes 39,4 seconds which is 77,3% of the entire elapsed time for all main applications. So that, we focused on accelerate RLWE_enc2 and RLWE_dec2 functions based on results were obtained at phase1. These functions call lots of different functions inside. For these functions, we produced IP cores individually except the function Knuth-Yao. However, LFSR (linear feedback shift register) which is a shift register whose input bit is the output of a linear function of two or more of its previous states was implemented because of the HLS does not support the rand () function used to generate random numbers. LFSR which was implemented generates 16-bit random numbers by shifting the 32-bit LFSR twice before performing XORing with 31-bit LFSR. All produced IP cores connected to the processing system with the AXI4 Lite interface as a slave. No optimization or change was applied in this process. The reason that we did not apply any optimization or changes is to compare pure software versions of encryption/decryption functions with pure hardware accelerators of these functions. On **figure 5**, all operating functions and sub-functions of encryption/decryption part are demonstrated.

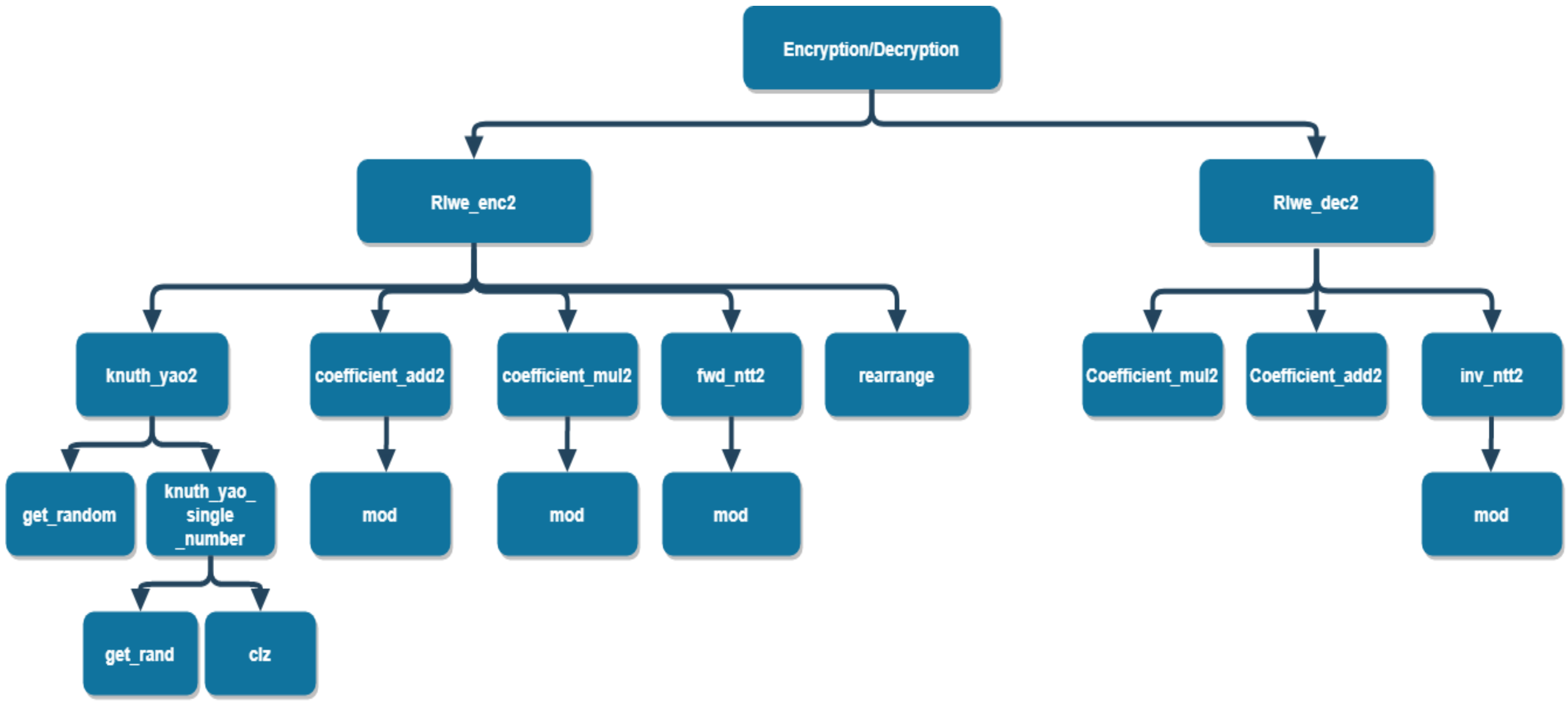


Figure 5. Functions and sub-functions of Encryption/Decryption part

2.3. Phase 3

As a consequence of information obtained by Profiling, it was noticed that the mod and clz (count leading zeros) functions were called many times by the methods. So, different versions were considered for clz and mod functions to reduce Time/Call value of these functions in profiling. Reducing the latency with substitute functions with new versions has great importance for design space exploration.

To summarize clz function; in a binary number, leading zeros are the zero digits in the most significant position of data, up to the position in which the first one is present. This function called about 10.5 million by knuth_yao_single_number function and 4 different versions were attempted.

Original version of clz function: This version count zeros in a 32-bit value by shifting the given value until this value equals to zero in a loop. As seen in **figure 6.1**, latency value can vary between 4 and 35.

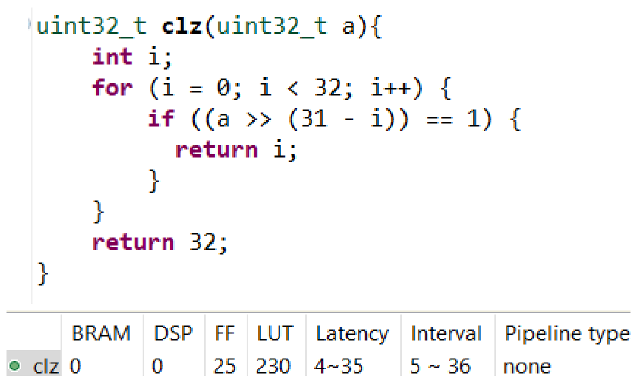
Clz version 1: It checks whether a 32-bit number is small with 50%, 75%, %87.5, %93.75, %96.875 of the maximum value it can get, and at the same time it shifts right and adds with numbers 16,8,4,2,1 respectively. So, this version performs these operations in 5 stages and reaches the result. As seen in **figure 6.2**, the latency value is 7 with ideal source utilization.

Clz version 2: This version checks given value's greatness over some constants value which are 0Xffff, 0xff, 0xf and 0x3 and produces Boolean value as 1 or 0. Then, some shifting and logical or operations are performed to obtain result. These processes can easily be seen from **figure 6.3**. and latency value is 34 with too much source utilization.

Clz version 3: This version performs log operation to the given integer value and converts the result to type of double. As seen **figure 6.4**, latency value is 244 with excessively high source utilization.

Clz version 4: This version adds the given number onto itself, shifted by 16, 8,4,2 and 1, respectively. Then, it multiplies the resulting value by a fixed number and returns its place in the array which created earlier. As seen **figure 6.5**, latency value is 11 with ideal source utilization.

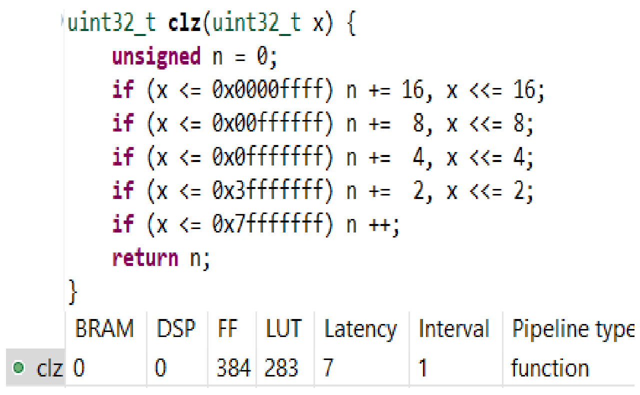
Consequently, latency is an important issue for acceleration. Although FF and LUT utilization increased compared to the original version as seen **figure 6.2**, it was decided to use the version with the lowest latency which is clz version 1. This version is more hardware-friendly compared to others because of shift and add operations which have small costs at hardware side.



```
uint32_t clz(uint32_t a){
    int i;
    for (i = 0; i < 32; i++) {
        if ((a >> (31 - i)) == 1) {
            return i;
        }
    }
    return 32;
}
```

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
● clz	0	0	25	230	4~35	5 ~ 36	none

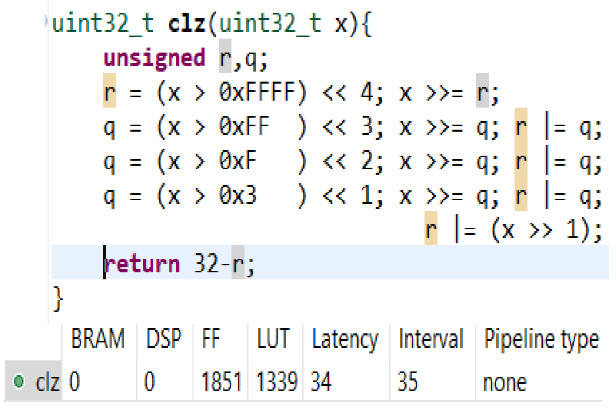
Figure 6.1. Original version of Clz.



```
uint32_t clz(uint32_t x) {
    unsigned n = 0;
    if (x <= 0x0000ffff) n += 16, x <<= 16;
    if (x <= 0x00ffffff) n +=  8, x <<= 8;
    if (x <= 0x0fffffff) n +=  4, x <<= 4;
    if (x <= 0x3fffffff) n +=  2, x <<= 2;
    if (x <= 0x7fffffff) n ++;
    return n;
}
```

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
● clz	0	0	384	283	7	1	function

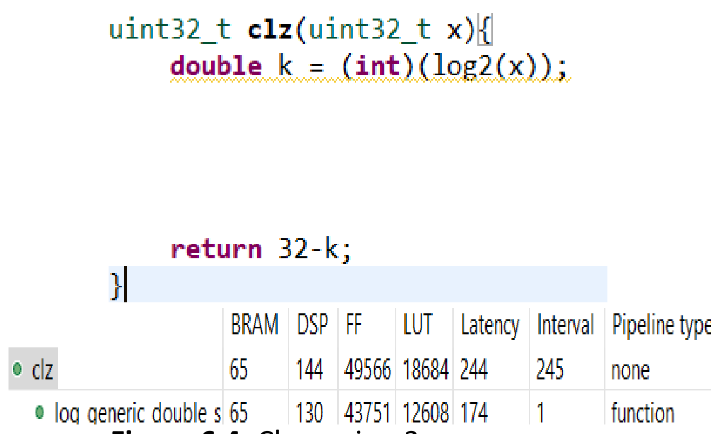
Figure 6.2. Clz version 1.



```
uint32_t clz(uint32_t x){
    unsigned r,q;
    r = (x > 0xFFFF) << 4; x >>= r;
    q = (x > 0xFF   ) << 3; x >>= q; r |= q;
    q = (x > 0xF    ) << 2; x >>= q; r |= q;
    q = (x > 0x3    ) << 1; x >>= q; r |= q;
                                     r |= (x >> 1);
    return 32-r;
}
```

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
● clz	0	0	1851	1339	34	35	none

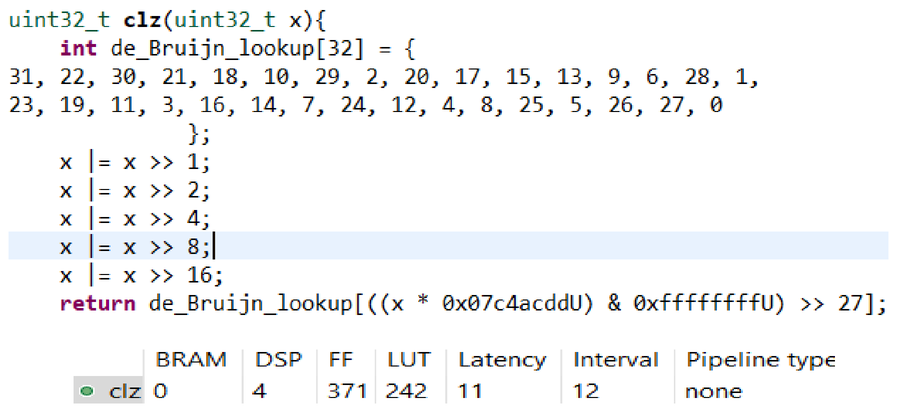
Figure 6.3. Clz version 2.



```
uint32_t clz(uint32_t x){
    double k = (int)(log2(x));



    return 32-k;
}
```

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
● clz	65	144	49566	18684	244	245	none
● log generic double s	65	130	43751	12608	174	1	function

Figure 6.4. Clz version 3.



```
uint32_t clz(uint32_t x){
    int de_Bruijn_lookup[32] = {
31, 22, 30, 21, 18, 10, 29, 2, 20, 17, 15, 13, 9, 6, 28, 1,
23, 19, 11, 3, 16, 14, 7, 24, 12, 4, 8, 25, 5, 26, 27, 0
                            };
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    return de_Bruijn_lookup[((x * 0x07c4acddU) & 0xffffffffU) >> 27];
```

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
● clz	0	4	371	242	11	12	none

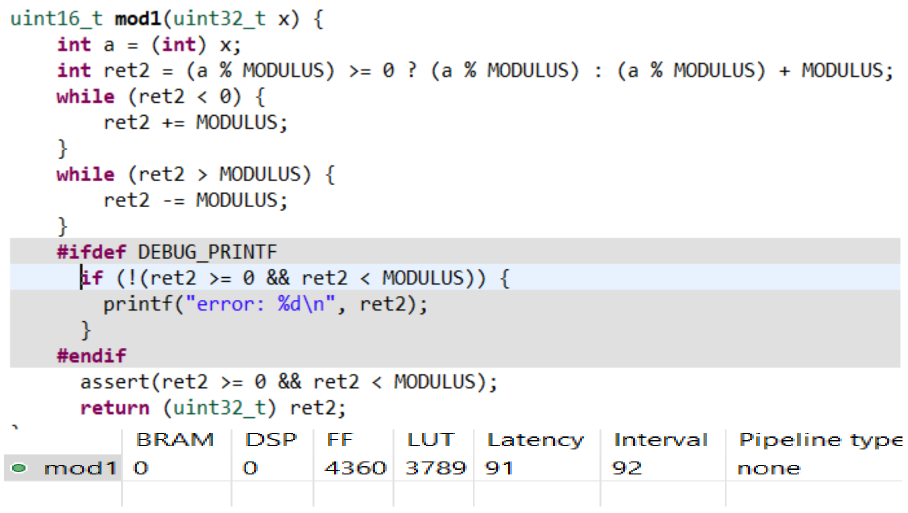
Figure 6.5. Clz version 4.

Mod function returns the remainder after a number is divided by the divisor. This function called by coefficient_mul2, coefficient_add2, fwd_ntt2 and inv_ntt2. Only 1 different version was attempted for this function.

The original version of the mod: In this version, as long as the dividend is a negative integer, the remainder value is added with modulus in a loop until result becomes 0. Otherwise, the remainder value is subtracted from modulus until the result becomes less than modulus value. As seen **figure in 7.1**, the latency value is 91 with 4360 FF and 3789 LUT usage.

Mod new version : In this version, an algorithm with the same result as the original version was obtained without using the modular process that spent a lot of clock cycles. Instead of '%' operator, operations are done with shift and multiplication operations with much more hardware friendly in this way.

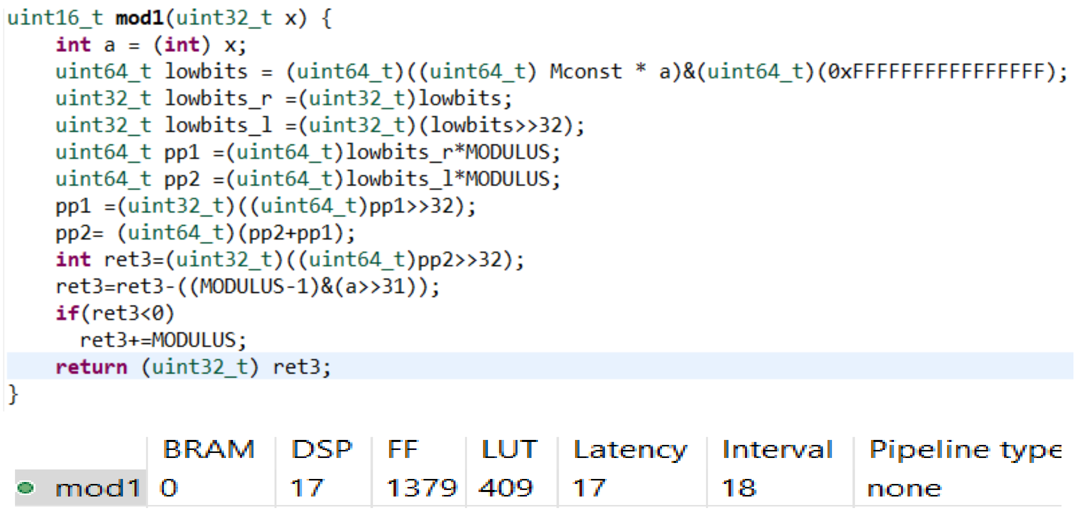
According to results in **figure 7.2.**, the new version of mod function with less source utilization and less latency was decided to use for further experiments of our project.



```
uint16_t mod1(uint32_t x) {
    int a = (int) x;
    int ret2 = (a % MODULUS) >= 0 ? (a % MODULUS) : (a % MODULUS) + MODULUS;
    while (ret2 < 0) {
        ret2 += MODULUS;
    }
    while (ret2 > MODULUS) {
        ret2 -= MODULUS;
    }
    #ifdef DEBUG_PRINTF
      if (!(ret2 >= 0 && ret2 < MODULUS)) {
        printf("error: %d\n", ret2);
      }
    #endif
      assert(ret2 >= 0 && ret2 < MODULUS);
      return (uint32_t) ret2;
```

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
● mod1	0	0	4360	3789	91	92	none

Figure 7.1. Original version of mod function.



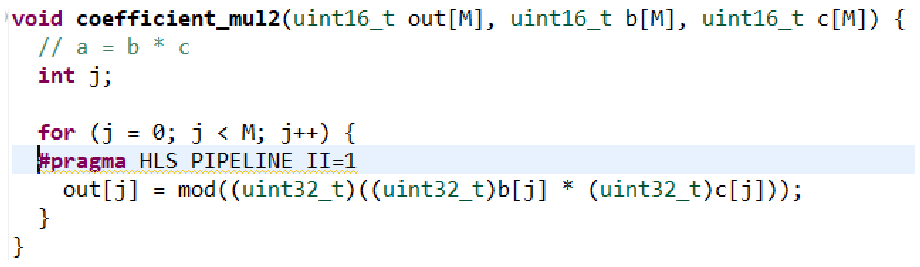
```
uint16_t mod1(uint32_t x) {
    int a = (int) x;
    uint64_t lowbits = (uint64_t)((uint64_t) Mconst * a)&(uint64_t)(0xFFFFFFFFFFFFFFFF);
    uint32_t lowbits_r =(uint32_t)lowbits;
    uint32_t lowbits_l =(uint32_t)(lowbits>>32);
    uint64_t pp1 =(uint64_t)lowbits_r*MODULUS;
    uint64_t pp2 =(uint64_t)lowbits_l*MODULUS;
    pp1 =(uint32_t)((uint64_t)pp1>>32);
    pp2= (uint64_t)(pp2+pp1);
    int ret3=(uint32_t)((uint64_t)pp2>>32);
    ret3=ret3-((MODULUS-1)&(a>>31));
    if(ret3<0)
      ret3+=MODULUS;
    return (uint32_t) ret3;
}
```

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
● mod1	0	17	1379	409	17	18	none

Figure 7.2. Version 1 of mod function.

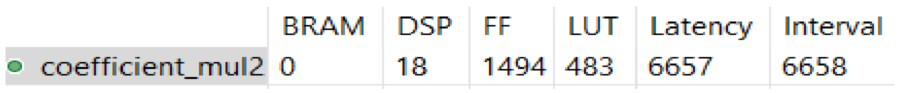
2.4 Phase 4

Vivado HLS provides pragmas that can be used to optimize the design: reduce latency, improve throughput performance, and reduce area and device resource utilization of the resulting RTL code. When these pragma techniques were considered, these techniques were realized to be important for reducing latency. A lot of loops especially nested loops are used in the Ring LWE algorithm. So, using Pipelining which improves the hardware function's performance by exploiting the parallelism between loop iterations evaluated. As seen in **figure 8.1**, this application has been attempted on Coefficient_mul2 function which is one of the sub-functions which algorithm is used. As seen in **figure 8.2** and **figure 8.3,** FF, LUT, and especially latency values are greatly reduced when we apply this method on this function. In a sequential language such as C/C++, the operations in a loop are executed sequentially and the next iteration of the loop can only begin when the last operation in the current loop iteration is complete. As seen in **figure 8.4** and **figure 8.5**, the effect of the Pipelining method is clearly visible. Finally, when this method was set appropriately in all functions, a great improvement in acceleration was recorded. In addition, the flattening technique used for nested loops was evaluated. However, since the loops we applied this technique did not meet the perfect or semi-perfect condition, the flatten technique was not applied.



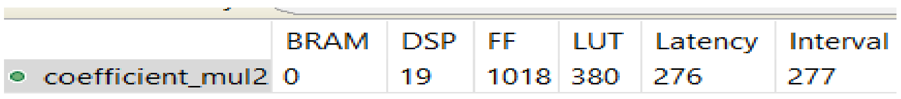
```
void coefficient_mul2(uint16_t out[M], uint16_t b[M], uint16_t c[M]) {
  // a = b * c
  int j;

  for (j = 0; j < M; j++) {
#pragma HLS PIPELINE II=1
    out[j] = mod((uint32_t)((uint32_t)b[j] * (uint32_t)c[j]));
  }
}
```

Figure 8.1. Coefficient_mul2 function.



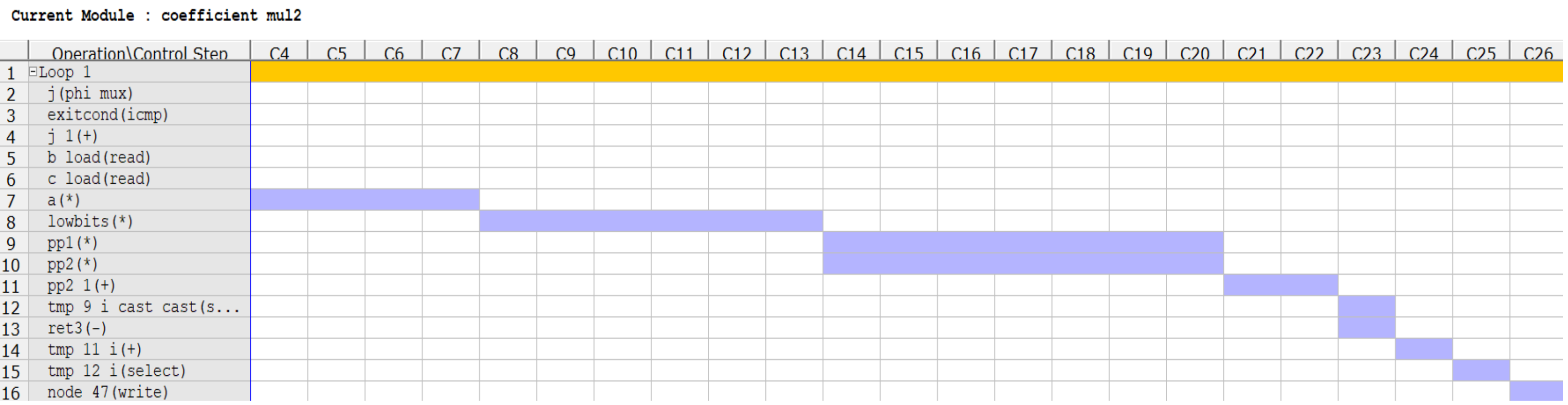
	BRAM	DSP	FF	LUT	Latency	Interval
● coefficient_mul2	0	18	1494	483	6657	6658

Figure 8.2. Coefficient_mul2 function without pipeline optimization.



	BRAM	DSP	FF	LUT	Latency	Interval
● coefficient_mul2	0	19	1018	380	276	277

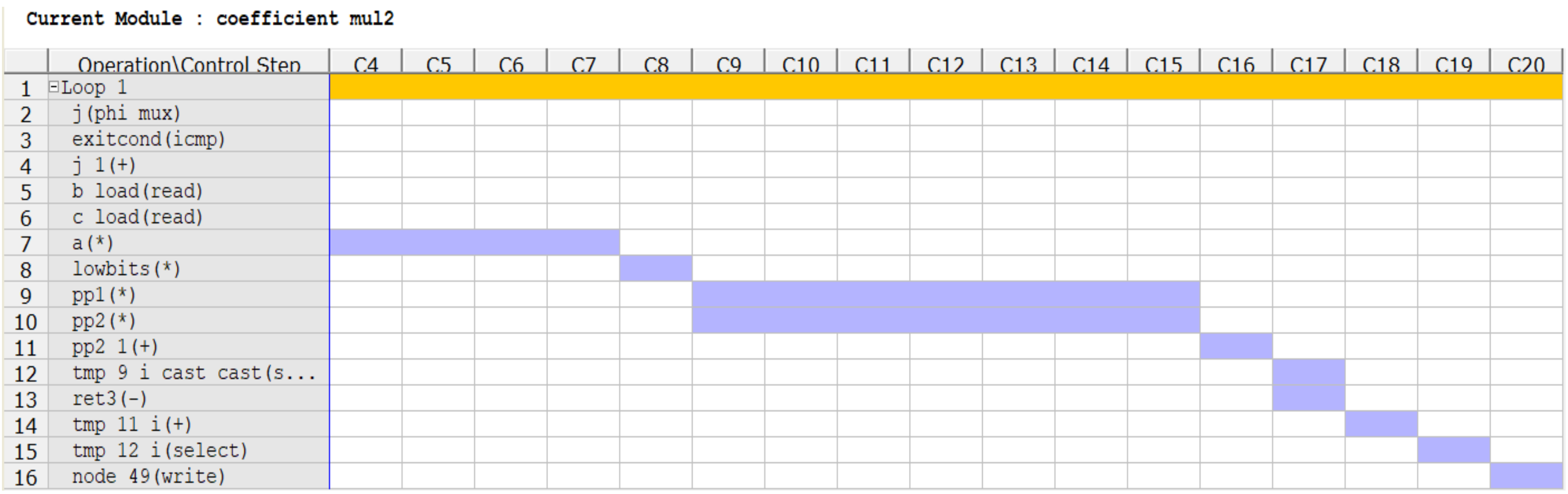
Figure 8.3. Coefficient_mul2 function with pipeline optimization.



Current Module : coefficient mul2

	Operation\Control Step	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19	C20	C21	C22	C23	C24	C25	C26
1	⊟Loop 1	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
2	j(phi mux)																							
3	exitcond(icmp)																							
4	j 1(+)																							
5	b load(read)																							
6	c load(read)																							
7	a(*)	🟦	🟦	🟦	🟦																			
8	lowbits(*)					🟦	🟦	🟦	🟦	🟦	🟦													
9	pp1(*)											🟦	🟦	🟦	🟦	🟦	🟦	🟦						
10	pp2(*)											🟦	🟦	🟦	🟦	🟦	🟦	🟦						
11	pp2 1(+)																		🟦	🟦				
12	tmp 9 i cast cast(s...																				🟦			
13	ret3(-)																				🟦			
14	tmp 11 i(+)																					🟦		
15	tmp 12 i(select)																						🟦	
16	node 47(write)																							🟦

Figure 8.4. Coefficient_mul2 function without pipeline.



Current Module : coefficient mul2

	Operation\Control Step	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19	C20
1	⊟Loop 1	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
2	j(phi mux)																	
3	exitcond(icmp)																	
4	j 1(+)																	
5	b load(read)																	
6	c load(read)																	
7	a(*)	🟦	🟦	🟦	🟦													
8	lowbits(*)					🟦												
9	pp1(*)						🟦	🟦	🟦	🟦	🟦	🟦	🟦					
10	pp2(*)						🟦	🟦	🟦	🟦	🟦	🟦	🟦					
11	pp2 1(+)													🟦				
12	tmp 9 i cast cast(s...														🟦			
13	ret3(-)														🟦			
14	tmp 11 i(+)															🟦		
15	tmp 12 i(select)																🟦	
16	node 49(write)																	🟦

Figure 8.5. Coefficient_mul2 function with pipeline

2.4. Phase 5

As seen in **figure 9.1**, it was noticed that by adding a master port connected to a high-performance port on the processing system to the slave accelerator IP without a master port wherein PL (Programmable Logic) part used in the previous phases might decrease the communication time with PS (Processing System) part. As seen **figure 9.2**, by producing both slave and master accelerator IP, it was provided to access the result directly to on-chip memory with activating HP0(High-Performance Port 0) in the processing system. Consequently, output data of accelerator IP directly send to PS reduces communication time. Latency difference is ignored which caused by the GPIO IP core slowing down the bandwidth of the AXI interconnect.

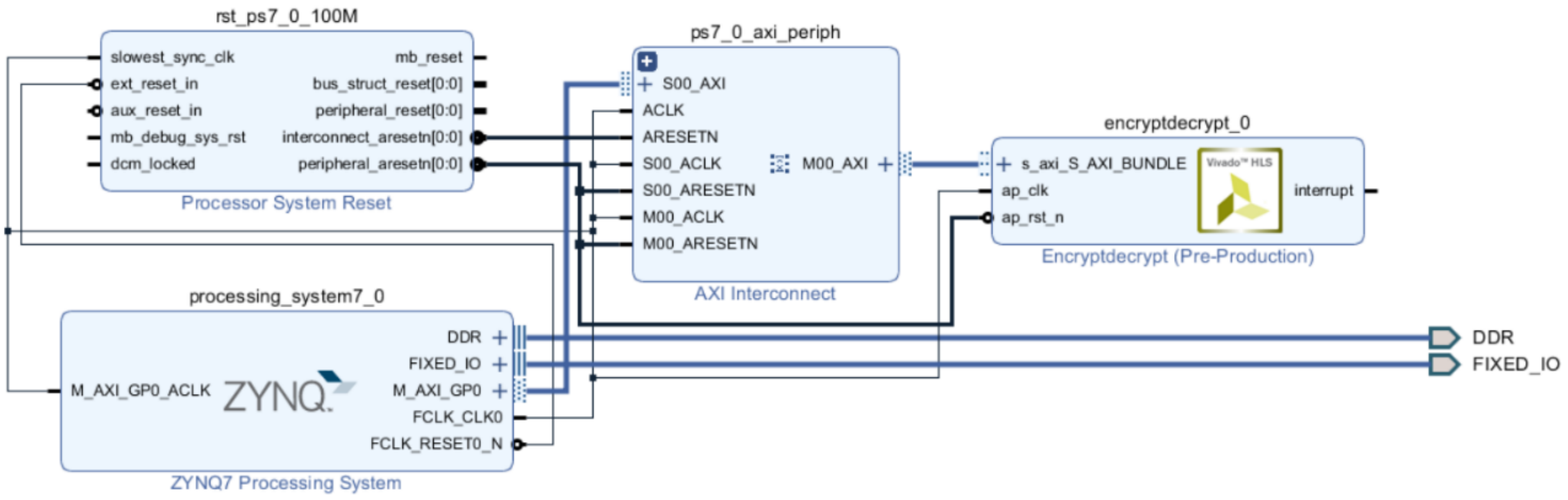


Figure 9.1. Block design of slave accelerator which was used in previous phases.

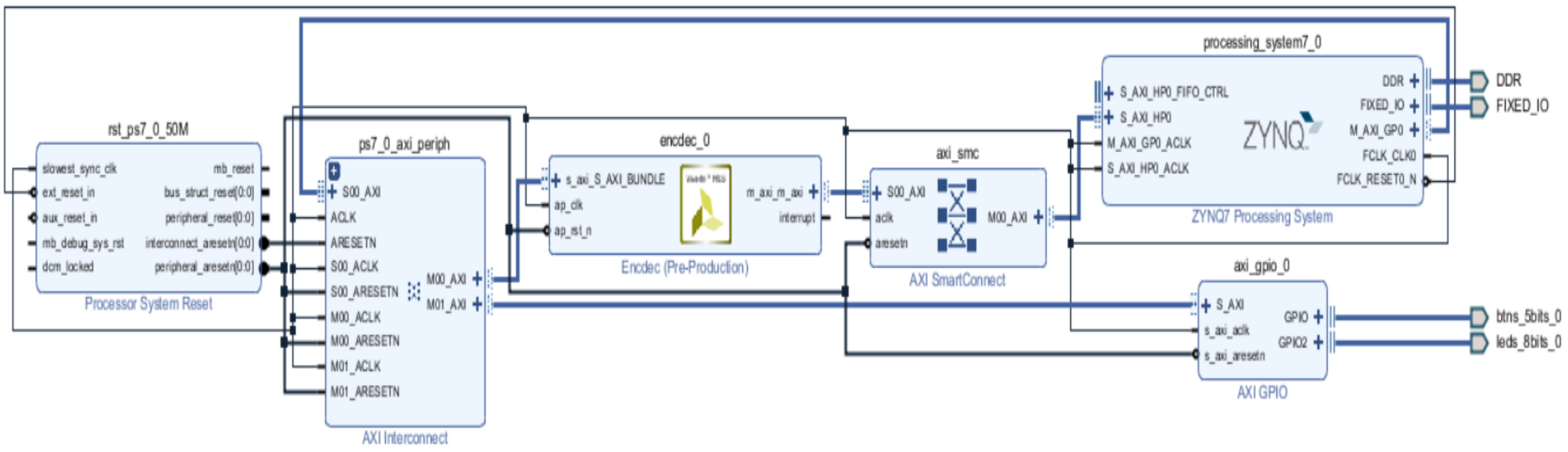


Figure 9.2. Both slave and master IP connections in block design.

3. Empirical Setup

Zynq 7000(ZedBoard) was used for our experiments. Also, Xilinx SDK, Vivado, and Vivado HLS tools were used. SD card interface is used for running the application of the algorithm on ZedBoard. Also, the SD card contains .bit file for programming the FPGA. UART channel is used for communicating with our computer via the PuTTY terminal. Buttons and LEDs were defined at the Programming Logic side to control our application in terms of start/stop or success/fail.

4. Empirical Results

Experiments have done with the Processing system with 667 MHz clock frequency (ARM Cortex A9 Dual-core CPU) and most of the experiments, Programmable Logic part tuned with 100 MHz clock frequency rate.

Hardware implementation results could be considered as **6,67** times faster for 100MHz clock frequency with programmable logic because of the clock frequency difference between the programmable logic part and the processing system part. However, operating with lower frequency at the programmable logic part provides lower power consumption.

PHASE 1 RESULTS

Code sections of algorithm’s main part	Elapsed Time	Clock Cycles
Knuth_yao_smaller_tables2	0.381 sec	0.25×10^9
Knuth_yao2	9.72 sec	$6{,}47 \times 10^9$
Knuth_yao_smaller_tables_single_number	0.313 sec	0.20×10^9
Fwd/inv_ntt2	0.948 sec	0.63×10^9
Encryption/Decryption	39.4 sec	26.3×10^9
Total	50.932 sec	33.9×10^9

Table 1. Software application results of Ring-LWE

Target clock frequency set as 10ns on HLS for all IP cores produced in phase 1. In this phase, all experiments were done with 100MHz PL clock frequency. In **table 1**, Encryption/Decryption part takes 39,4 seconds which is the 77,3% of the entire elapsed time for all main applications.

PHASE 2 RESULTS

Functions	Elapsed Time	Clock Cycles
Coefficient_add2	32.01 µs (Software) 162.43 µs (Hardware accelerated)	21318 (Software) 108180 (Hardware accelerated)
Coefficient_mul2	31.9 µs (Software) 370.6 µs (Hardware accelerated)	21244 (Software) 212566 (Hardware accelerated)
Fwd_ntt2	395.93 µs (Software) 1542.15 µs (Hardware accelerated)	263688 (Software) 1027070 (Hardware accelerated)
Rearrange2	13.57 µs (Software) 81.43 µs (Hardware accelerated)	9040 (Software) 54232 (Hardware accelerated)
Inv_ntt2	493.05 µs (Software) 1972.41 µs (Hardware accelerated)	328370 (Software) 1313628 (Hardware accelerated)
RLWE_enc2	1825.79 µs (Software) 3714.63 µs (Hardware accelerated)	1215976 (Software) 2473944 (Hardware accelerated)
RLWE_dec2	561.82 µs (Software) 2301.98 µs (Hardware accelerated)	374172 (Software) 1533118 (Hardware accelerated)

Table 2. Results of encryption/decryption functions and its sub-functions without any optimization.

Target clock frequency set as 10ns on HLS for all IP cores produced in phase 2. In this phase, all experiments were done with 100MHz PL clock frequency. Because of no optimization applied, the results are not exactly as intended.

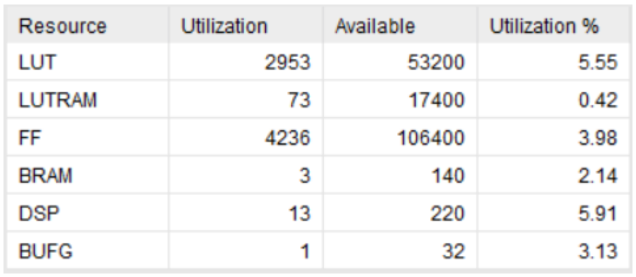
PHASE 3 RESULTS

Functions	Elapsed Time	Clock Cycle
RLWE_enc2 **Mod: original version** **Clz : original version**	3853 µs	2566518
RLWE_enc2 **Mod: new version** **Clz : original version**	707,77 µs	469464
RLWE_enc2 **Mod: new version** **Clz : new version**	542,47 µs	362122
RLWE_dec2 **Mod: original version**	2302.11 µs	1533204
RLWE_dec2 **Mod: new version**	346.30 µs	230634

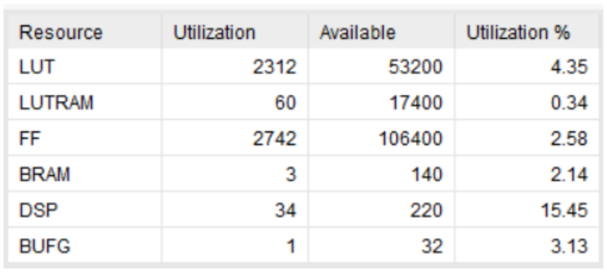
Table 3. Experiments with different versions of mod and clz functions

Target clock frequency set as 10ns on HLS for all IP cores produced in phase 3. In this phase, all experiments were done with 100MHz PL clock frequency. mod function substitution has very big impact on the elapsed time (**5,44x-6,65x performance improvement)**. Also, clz function substitution improves the performance **(1.3x speedup).** In **figure 10.1** and **figure 10.2,** DSP usage of RLWE_dec2 increased because of the substitution of mod operation with a new version. Same as RLWE_enc2 in **figure 10.3** and **figure 10.4,** source utilizations except for BRAM and BUFG increased with a new version of mod operation.

PHASE 2-3 UTILIZATIONS

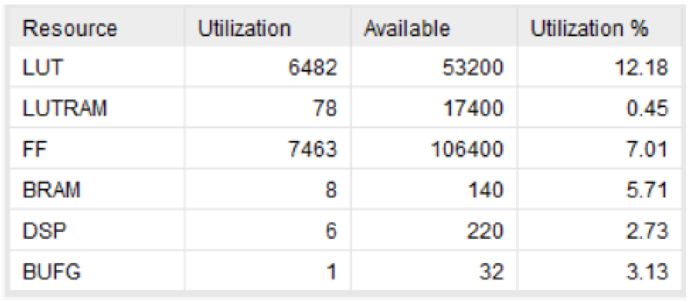


Resource	Utilization	Available	Utilization %
LUT	2953	53200	5.55
LUTRAM	73	17400	0.42
FF	4236	106400	3.98
BRAM	3	140	2.14
DSP	13	220	5.91
BUFG	1	32	3.13



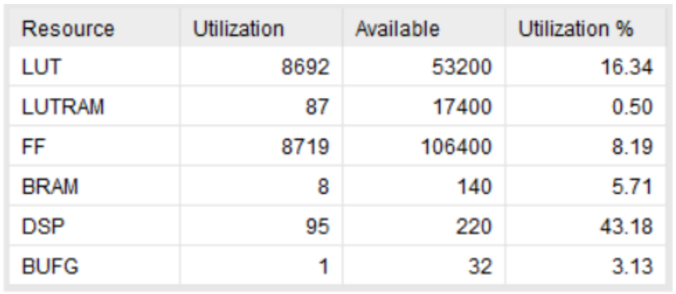
Resource	Utilization	Available	Utilization %
LUT	2312	53200	4.35
LUTRAM	60	17400	0.34
FF	2742	106400	2.58
BRAM	3	140	2.14
DSP	34	220	15.45
BUFG	1	32	3.13

Figure 10.1.RLWE_dec2 utilization with original mod **Figure 10.2.** RLWE_dec2 utilization with new mod



Resource	Utilization	Available	Utilization %
LUT	6482	53200	12.18
LUTRAM	78	17400	0.45
FF	7463	106400	7.01
BRAM	8	140	5.71
DSP	6	220	2.73
BUFG	1	32	3.13

Figure 10.3. RLWE_enc2 utilization with original mod



Resource	Utilization	Available	Utilization %
LUT	8692	53200	16.34
LUTRAM	87	17400	0.50
FF	8719	106400	8.19
BRAM	8	140	5.71
DSP	95	220	43.18
BUFG	1	32	3.13

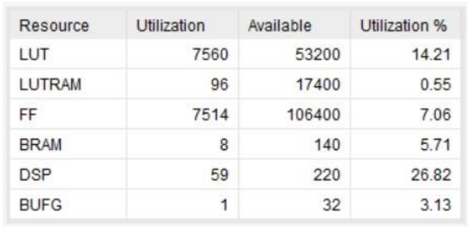
Figure 10.4. RLWE_enc2 utilization with new mod and new clz

PHASE 4 RESULTS

Function	HLS Target Clock Cycle	Elapsed Time	Clock Cycle
RLWE_enc2	2,5	590 µs	459808
	10	341 µs	393280
RLWE_dec2	2,5	379 µs	252908
	10	208 µs	276746
Encdec (RLWE_enc2 + RLWE_dec2)	2,5	825 µs	567432
	10 (50 MHz)	860 µs	591505

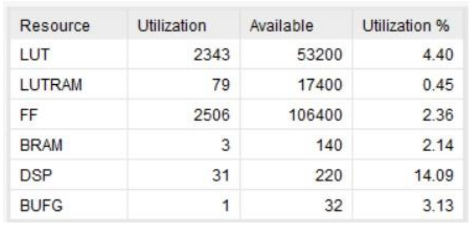
Table 4. Results of optimized functions with HLS PIPELINE pragmas

Encdec function accelerator produced with combining RLWE_enc2 and RLWE_dec2 as a single function. Target clock frequency set as 10ns and 2.5ns on HLS for all IP cores produced in phase 4. In this phase, all experiments were done with 100MHz PL clock frequency except Encdec function which produced with 10 target clock cycle. In conclusion, pipeline directives effects to results positively and produced IP cores with 10ns target clock frequency show better performance compared to 2.5ns. **Figure 11.1, figure 11.2** and **figure 11.3** shows utilization report of pipelined RLWE_dec2, RLWE_enc2 and encdec functions, respectively.



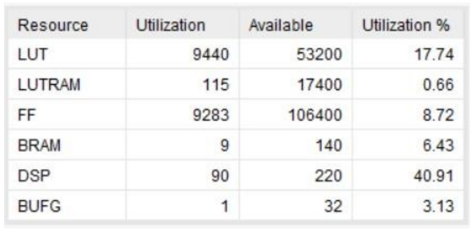
Resource	Utilization	Available	Utilization %
LUT	7560	53200	14.21
LUTRAM	96	17400	0.55
FF	7514	106400	7.06
BRAM	8	140	5.71
DSP	59	220	26.82
BUFG	1	32	3.13

Fig 11.1 RLWE_dec2 utilization



Resource	Utilization	Available	Utilization %
LUT	2343	53200	4.40
LUTRAM	79	17400	0.45
FF	2506	106400	2.36
BRAM	3	140	2.14
DSP	31	220	14.09
BUFG	1	32	3.13

Fig 11.2. RLWE_enc2 utilization



Resource	Utilization	Available	Utilization %
LUT	9440	53200	17.74
LUTRAM	115	17400	0.66
FF	9283	106400	8.72
BRAM	9	140	6.43
DSP	90	220	40.91
BUFG	1	32	3.13

Fig 11.3. Encdec utilization

PHASE 5 RESULTS

Accelerated Encdec Function (RLWE_enc2 + Rlwe_dec2)

HLS target clock cycle	Elapsed Time	Clock Cycle
10ns	728 µs	483494
12.75ns	662 µs	439228
17ns	**483 µs**	**321044**

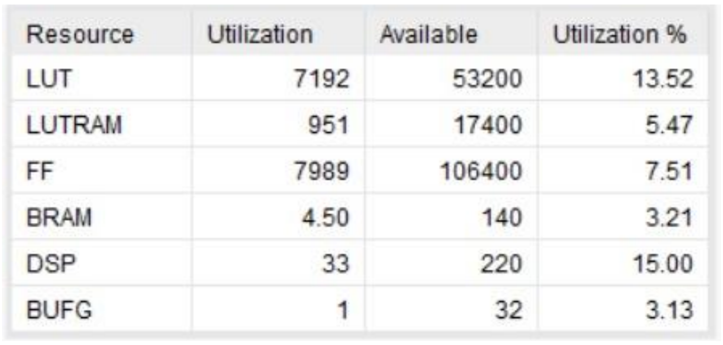
Table 5. Results of improved data transfer

Pure Software Encdec Function(RLWE_enc2 + RLWE_dec2)

Function	Elapsed Time	Clock Cycle
Encdec	2390.19 µs	1591864

Table 6. Software implementation of Encdec function

In this phase, all experiments were done with 70MHz PL clock frequency. Data transfer optimization has been done with using High Performance port by using master AXI interface. Thanks to this optimization our data transfer latency reduces 72us to 9 us. **Figure 12** shows source utilization of encdec function produced with 17n target clock frequency.



Resource	Utilization	Available	Utilization %
LUT	7192	53200	13.52
LUTRAM	951	17400	5.47
FF	7989	106400	7.51
BRAM	4.50	140	3.21
DSP	33	220	15.00
BUFG	1	32	3.13

Fig 12. Encdec function with target 17ns utilization

Pure software implementation of both RLWE_enc2 and RLWE_dec2 function takes 2390 microseconds and accelerated encdec function takes 483 microseconds as seen in **table 5** and **table 6**. Achieved speedup for encdec function is about **5**. If we could operate accelerated encdec function at the same clock frequency with the processor system, we would execute encdec function approximately **47** times faster. Pure software implementation of Enc/Dec part takes approximately 39,8 seconds. After the optimizations and hardware implementations we have done, Enc/Dec part takes approximately 20,7 seconds. Therefore, we achieved approximately 48% acceleration at Encryption/Decryption part of the entire main application. **Figure 13** demonstrates project illustration with details of Ring-LWE encryption/decryption.

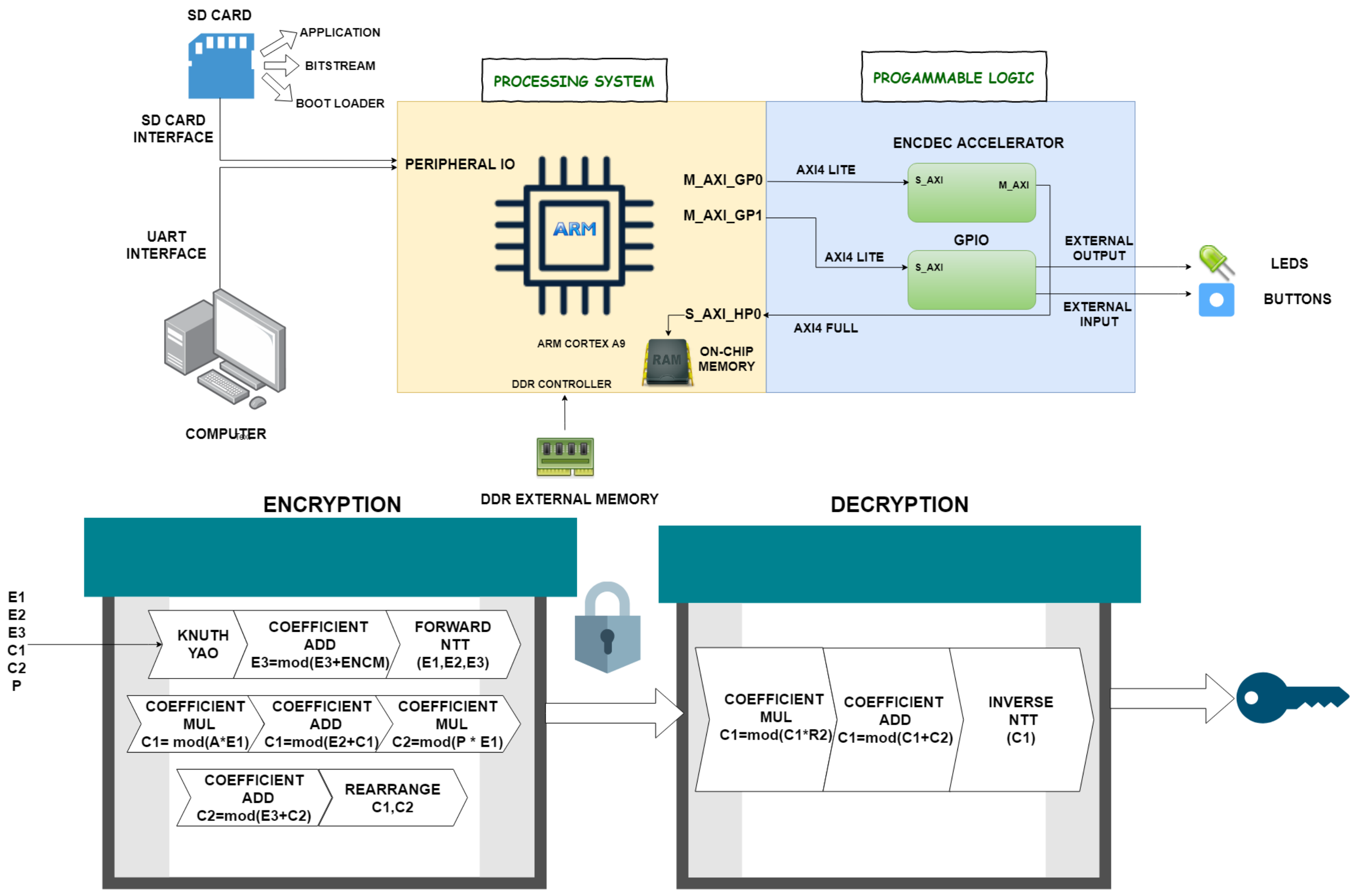


Fig 13. Project Illustration

5. Conclusion

As is known to all, the safety issue has significance importance. It can be easily seen here, Ring-LWE quantum-resistant cryptography algorithm can be accelerated due to results represented in Empirical Results. Vivado HLS tool offers us fast design space exploration. Therefore, optimization of performance metrics like memory usage, CPU time, etc. can be done properly.

According to the results that we obtained, we achieved an important amount of speedup rate by replacing some functions with new versions and using some optimization directives provided by the Vivado HLS tool. At the end of the project, memory access time is reduced by using high-performance port by master AXI interface.

5.1 Technical Complexity

The ring-lwe algorithm is computationally intensive and consisted of a lot of functions, so it was difficult to understand and apply the ring lwe algorithm at the beginning of the project. With the contributions of the System on Chip course given by our advisor Mr. San, we were able to work from profiling in the first stage of our project to data transfer optimization in the last stage.

5.2 Implementation

We realized our implementation with Zynq-7000 ZedBoard provided by our university Eskisehir Technical University within the scope of the System on Chip course.

5.3 Marketability/Innovation

As an example, it is predicted that the RSA-2048-bit password will be broken by 2026 with 1/7 probability and 2031 with 1/2 probability. It is observed that countries are very uncomfortable with these developments, take steps to take measures, support researchers, direct their companies to develop special solutions, open source platforms to researchers. E.g., The US Standard Institute (NIST) has announced that by 2022, quantum-resistant algorithms will be developed in collaboration with universities and the private sector.

Consequently, designing crypto processors with using System on Chip solution like we did is both faster and reliable way to implement post-quantum cryptographic algorithm.

5.4 Reusability

Our HLS-produced IP core for Ring-LWE encryption/decryption can be reusable both synthesizing in HLS with given source codes or can be added to block design with given IP file in the GitHub repository.

GitHub Repository : https://github.com/atakankalkar/pars_designspaceexp_xohw20_170

6. Acknowledgement

As team members, we would like to thank our instructor İsmail SAN for the help in both System-On-Chip (SoC) design and hardware-efficient implementation of Ring-LWE algorithm especially contributes to the project to achieve good results by defining and implementing of mod operation in figure 7.2.

References

[1] A. C. Yao, “Context-Free Grammars and Random Number Generation,” in *Combinatorial Algorithms on Words*, Berlin, Heidelberg, 1985, pp. 357–361.

[2] https://github.com/ruandc/Ring-LWE-Encryption

[3] T. N. Tan and H. Lee, “Efficient-Scheduling Parallel Multiplier-Based Ring-LWE Cryptoprocessors,” *Electronics*, vol. 8, no. 4, p. 413, Apr. 2019, doi: 10.3390/electronics8040413.

[3] Flexible Accelerators for Early Adoption of Ring-LWE Post-quantum Cryptography,” *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 2, pp. 1–17, Mar. 2020, doi: 10.1145/3378164.

[4] E. Ozcan and A. Aysu, “High-Level-Synthesis of Number-Theoretic Transform: A Case Study for Future Cryptosystems,” *IEEE Embedded Syst. Lett.*, pp. 1–1, 2020, doi: 10.1109/LES.2019.2960457.

[5] R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede, “Efficient Software Implementation of Ring-LWE Encryption,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, Grenoble, France, 2015, pp. 339–344, doi: 10.7873/DATE.2015.0378.

[6] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, “Compact Ring-LWE Cryptoprocessor,” in *Advanced Information Systems Engineering*, vol. 7908, C. Salinesi, M. C. Norrie, and Ó. Pastor, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 371–391.

[7] K. Kawamura, M. Yanagisawa, and N. Togawa, “A loop structure optimization targeting high-level synthesis of fast number theoretic transform,” in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, Santa Clara, CA, Mar. 2018, pp. 106–111, doi: 10.1109/ISQED.2018.8357273.