양자내성암호 NTRU의 HLS 구현

2020172025 융합전자공학과 김선빈
2021107075 융합전자공학과 최창림

- 목 차 -

1 양자내성암호 시스템 NTRU

1.1 PQC NTRU 개요

최근 컴퓨터 시장에서 차세대 컴퓨터 개발을 위해 양자컴퓨터 연구를 진행 중이다. 양자컴퓨터는 기존의 컴퓨터에서 하지 못하던 계산이 가능하며 기존의 암호시스템에서 기반으로 사용되던 소인수 분해 난제도 해결할 수 있다. 이는 기존의 암호시스템이 위험하다는 것을 뜻하며, 이를 막기위해 NIST 기관에서는 새로운 암호 시스템인 양자내성 암호시스템(Post-Quantum Cryptography, PQC)의 표준화 공모를 진행하고 있다. 양자내성 시스템은 여러가지 형태가 있는데 그 중에서도 가장 표준화 가능성이 큰 것이 NTRU이다. 현재 NTRU를 하드웨어로 개발하는 것을 연구로 하고 있고, 이를 HLS를 활용해 구현, 최적화, Emulation을 진행해보며 HLS 구현의 장단점과 특징을 파악해 보았다.

1.2 PQC NTRU 연산

NTRU는 크게 Key를 생성하는 Key Generation, 데이터를 암호화하는 Encryption, 암호화 데이터를 복호화하는 Decryption으로 나눌 수 있다. 그 중에서도 우리는 Encryption 단계를 구현하였다.

KeyGen′(*seed*)	Encrypt($\mathbf{h}, (\mathbf{r}, \mathbf{m})$)	Decrypt($(\mathbf{f}, \mathbf{f}_p, \mathbf{h}_q), \mathbf{c}$)
1. $(\mathbf{f}, \mathbf{g}) \leftarrow$ Sample_fg(*seed*)	1. $\mathbf{m}' \leftarrow \text{Lift}(\mathbf{m})$	1. if $\mathbf{c} \not\equiv 0 \pmod{(q, \Phi_1)}$ return $(0, 0, 1)$
2. $\mathbf{f}_q \leftarrow (1/\mathbf{f}) \bmod (q, \Phi_n)$	2. $\mathbf{c} \leftarrow (\mathbf{r} \cdot \mathbf{h} + \mathbf{m}') \bmod (q, \Phi_1\Phi_n)$	2. $\mathbf{a} \leftarrow (\mathbf{c} \cdot \mathbf{f}) \bmod (q, \Phi_1\Phi_n)$
3. $\mathbf{h} \leftarrow (3 \cdot \mathbf{g} \cdot \mathbf{f}_q) \bmod (q, \Phi_1\Phi_n)$	3. return $\mathbf{c}$	3. $\mathbf{m} \leftarrow (\mathbf{a} \cdot \mathbf{f}_p) \bmod (3, \Phi_n)$
4. $\mathbf{h}_q \leftarrow (1/\mathbf{h}) \bmod (q, \Phi_n)$		4. $\mathbf{m}' \leftarrow \text{Lift}(\mathbf{m})$
5. $\mathbf{f}_p \leftarrow (1/\mathbf{f}) \bmod (3, \Phi_n)$		5. $\mathbf{r} \leftarrow ((\mathbf{c} - \mathbf{m}') \cdot \mathbf{h}_q) \bmod (q, \Phi_n)$
6. return $((\mathbf{f}, \mathbf{f}_p, \mathbf{h}_q), \mathbf{h})$		6. if $(\mathbf{r}, \mathbf{m}) \in \mathcal{L}_r \times \mathcal{L}_m$ return $(\mathbf{r}, \mathbf{m}, 0)$
		7. else return $(0, 0, 1)$

Figure 9: The DPKE for the NTRU submission.

[Figure 1] NTRU 암,복호화 단계

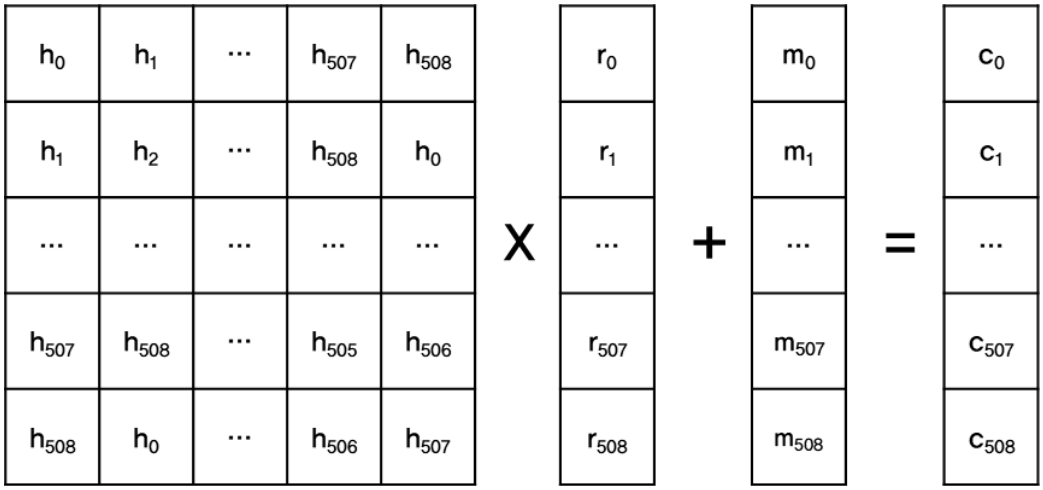
NTRU는 공개키 h, 개인키 r, 데이터 m 3가지 요소로 암호화 연산을 진행한다. 공개키 h는 11bit의 계수를 가지는 509차항을 가지는 다항식으로 표현되고 개인키 m과 데이터 m은 2bit의 계수를 가지는 509차항을 가지는 다항식으로 표현된다.

데이터	표현	계수 범위
m`	$m_0 + m_1x + m_2x^2 + \cdots + m_{508}x^{508}$	0, 1, 2
r	$r_0 + r_1x + r_2x^2 + \cdots + r_{508}x^{508}$	0, 1, 2
h	$h_0 + h_1x + h_2x^2 + \cdots + h_{508}x^{508}$	0, 1, 2, ⋯, 1023

[Figure 2] NTRU 구성요소

NTRU의 Encryption 단계는 크게 3가지 단계로 나눌 수 있다. 첫 번째는 입력받은 데이터 m을 Lift하여 0, 1, 2로 이루어진 값을 -1, 0, 1로 바꿔주는 단계이다. 두 번째는 공개키 h와 개인키 r을 곱하고 Lift m을 더해주는 과정이다. 이는 509차항 다항식 두개를 곱하고 마지막으로 509차항 다항식을 더해주는 과정으로 볼 수 있다. 마지막은 다항식 곱셈으로 인해 발생하는 509차항이 넘어가는 다항식을 509차 이하의 다항식으로 바꿔주는 modular과정이다. 두 번째 과정과 마지막 과정을 하나의 과정으로 보고 matrix의 곱셈으로 보면 [Figure 3]처럼 표현할 수 있다. 이는 결과적으로 convolution 연산의 형태를 가지게 된다.

우리는 이를 Verilog로 굉장히 복잡한 과정을 통해 구현한 경험이 있다. 하지만 HLS를 활용하면 이를 상대적으로 간단하게 구현할 수 있고 최적화를 진행할 수 있을 것이라고 생각해 구현을 진행하였고 Verilog로 구현한 결과를 비교하는 과정을 진행하였다.



[Figure 1] NTRU 연산

2 HLS를 활용한 NTRU HW 구현

2.1 Convolution, Modular

PQC NTRU에서의 encryption 과정에서 중요한 연산으로는 mod Φ_1 Φ_n 과, mod q (q = 1024) 두 가지가 있다. Φ_1 Φ_n 은 x^n-1을 뜻하고, 우리가 사용하는 파라미터에서의 n값은 509이다. mod Φ_1 Φ_n 연산은 두 다항식의 곱셈이 convolution 형태를 띄게 되는 이유이다. 우선 두 mod 연산 중 비교적 간단한 mod q 부터, 1) CPU 대비 HW로 구현했을 때의 장점과, 2) Verilog 대비 C++을 이용한 HLS를 이용했을 때의 장단점 및 특징에 대해서 정리한다.

(1) Mod q의 HW구현 및 HLS 구현의 특징

Mod q 연산은 연산 과정에 나오는 다항식의 각 계수의 값이 2048를 넘으면, 2048로 나누는 연산이다. 1027 mod 2048 = 1027, 2050 mod 2048 = 2 와 같은 연산을 수행한다. 이를 CPU로 구현 시, 16bit * 16bit 혹은 16bit + 16bit의 연산 후 얻어진 16bit 결과값의 최상위 여섯 비트를 지우는 연산이 추가적으로 필요하다. 최상위 다섯 비트를 지우는 이유는, unsigned 11bit로는 최대 2048까지

표현이 가능하므로, 이 11bit의 값이 결국 연산 후 2048로 modular 한 값이 되기 때문이다. 하지만 이를 HW로 구현하면 11bit wire끼리 연산을 수행하며, 12번째 bit의 carry는 자동 overflow되게 함으로써 modular 2048의 연산을 추가적인 HW 없이 구현할 수 있다. 이 연산은 verilog에서는 [Figure 4]과 같은 선언으로 10-bit wire선언이 가능하다.

```
wire [10:0] a;
```

[Figure 4] 11-bit wire를 verilog에서 선언하는 방법

HLS에서는 C++언어를 사용하므로 기본적인 short type등을 사용하면 16-bit wire처럼 작동하기 때문에, n-bit wire를 구현할 수 있는 header file인 ap_int.h를 추가하고 [Figure 5]와 같은 새로운 type의 변수를 선언해주어야 한다. 이와 같은 선언을 통해 원하는 비트수의 wire를 사용하여 우리의 의도에 맞는 12-bit overflow를 발생시킬 수 있다. 실제 연산결과도 우리가 의도한 대로 수행이 되었음을 debugging을 통해 확인하였다.

```
ap_uint<11> rh_block[POLY_SIZE];
ap_uint<11> h_block[POLY_SIZE];
ap_uint<2> r_block[POLY_SIZE];
ap_uint<2> m_block[POLY_SIZE];
```

[Figure 5] 11-bit, 2-bit wire를 HLS에서 선언하는 방법

(1) Convolution의 HW구현 및 HLS 구현의 특징

다음으로 mod $\Phi_1 \Phi_n$ 은 연산 결과 중 x^n 이상의 지수를 갖게 되는 값을 x^n-1로 나눈 나머지를 결과 값으로 만들어주는 연산이다. 우리가 구현한 모듈에서 n = 509 이므로, $\Phi_1 \Phi_n$ 은 x^{509}-1를 뜻하고, x^{509}-1 mod (x^{509}-1) = 0 이므로, x^{509} mod (x^{509}-1) = 1 이고, 즉 x^{509}차 이상의 항은 그 지수 값에서 509를 뺀 값으로 나눠진 값을 가지며, 이를 예를 들면 아래와 같이 modular 된다.

$$3x^{511} + 5x^3 + 1 \bmod (\Phi_1 \Phi_n) = 5x^3 + 3x^2 + 1$$

이와 같은 modular 연산은 덧셈, 뺄셈시에는 지수의 변화가 없으므로 적용되지 않으며, 곱셈 시에 적용이 된다. 따라서 encryption에서 r * h 연산을 할 때 이 modular 연산이 적용되는데, 결과적으로 정리하게 되면 이 r * h 연산은 convolution 연산 형태를 띄게 된다. Convolution 연산은 11bit인 h와 * 2bit인 r

의 연산으로 이루어지는데, 이는 비교적 critical path가 짧아 가벼운 연산이면서, modular q를 제외하고도 509^2번의 연산을 수행해야 하므로 CPU로 수행하기에는 쉽지 않은 연산이다. 따라서 전용 HW를 구현하여 해당 연산기를 병렬적으로 배치함으로써 509 cycle에 연산을 수행하도록 할 수 있다. 이 Convolution을 Verilog로 구현 시 하나의 모듈을 구성하여, left shift, right shift를 통해 wire와 reg를 저장하며 연산을 수행하는 등, 연산의 최적화를 위해 여러 복잡한 아이디어와 구현 능력을 필요로 한다.

```
52      always@(*)  begin
53          n_Cnt = 0;
54          case(c_State)
55              RUN:    n_Cnt = c_Cnt + 1;
56          endcase
57      end
58
59      // calculations on R*H and save it in n_rh
60
61      always@(*)  begin
62          n_rh = 0;
63          case(c_State[1])
64              1'b1    :   n_rh = {TNR_Result[9839:0], TNR_Result[9851:9840]}; // RUN, DONE
65          endcase
66      end
67
68      assign o_CONV_Result = c_rh;    // final rotation for rh
69
70      TNR_Multiplication inst_TNR_Multiplication_1 (.i_H(i_CONV_Gen), .i_Tnr(i_CONV_Tnr), .c_rh(c_rh), .o_Data(TNR_Result));
71
72
73
74  endmodule
```

[Figure 6] Convolution submodule HW의 일부 Verilog 코드

하지만 이는 HLS로 구현 시 훨씬 빠르게 설계할 수 있었다.

```
   for(int k = 0; k < POLY_SIZE-3; k++) {
#pragma HLS pipeline II=1
      for (int i=1; i<POLY_SIZE-3 -k; i++) {
#pragma HLS UNROLL
         rh_block[k] += r_block[k+i] * h_block[POLY_SIZE-3 - i];
      }
      for (int i=0; i<k+1; i++)
#pragma HLS UNROLL
         rh_block[k] += r_block[k-i] * h_block[i];
   }
```

[Figure 7] Convolution HW의 HLS 코드

또한, Vitis HLS를 이용한 C Synthesis를 통해 covolution 연산을 수행하는 for문이 얼마나 ALU를 사용하고 몇 cycle만에 연산이 종료되는 지 등을 쉽게 파악할 수 있었다. 목표하는 cycle수에 맞는지 확인하고, pipelining 혹은 unrolling을 적용해보며 목표치에 도달하도록 하는 데에 HLS가 Verilog 코딩 대비 얼마나

강력한 장점이 있는지 느낄 수 있었고, 코드도 훨씬 간결하게 작성되는 것을 과제를 진행하며 느낄 수 있었다. 하지만, verilog는 posedge에 따라 counter를 만들어 원하는 clock cycle에 맞추어 mux의 조건문을 설정하거나, left-shift, right-shift의 타이밍을 조정하는 등의 세세한 코딩이 가능하였는데, HLS에서는 자동적으로 clock cycle을 조정해주며 연산을 수행하기 때문에 clock을 counting할 수가 없었고, 이런 점에서 정교한 코딩을 하는 데에 어려움이 있었다.

2.2 NTRU의 HLS 구현 Novelty

NTRU를 HLS를 활용하여 구현할 때 Verilog로 구현할 때와는 다른 점이 많아 여러가지를 느껴볼 수 있었다. 우선은 몇 가지 장점들을 느낄 수 있었다. 첫 번째 장점은 메모리 interface가 쉽다는 점이였다. 몇 가지 명령어를 추가하면 설계자가 의도한 size의 데이터를 메모리로부터 받아 모듈에 전송하는 것을 설계하는데 편하다는 것을 느꼈다. 두 번째는 빠른 synthesis과 debugging 과정 이였다. Verilog를 이용해서 synthesis이후 debugging 과정을 처리하기 위해서는 여러 가지 tool을 사용해 몇 가지 과정을 처리해야 하지만 HLS를 활용하면 상대적으로 빠른 synthesis과정과 HLS툴에서 모듈 내 ALU 사용 상황을 알려주는 등 debugging이 수월하다는 것을 느꼈다. 하지만 몇 가지 불편한 부분도 느낄 수 있었다. 첫 번째로 암호화 시스템이 producer-consumer 형태가 아닌 feedback 형태를 지니고 있어 HLS를 활용하기 힘들다는 점이었다. 암호화 시스템은 이전의 연산결과가 현재 입력데이터와 연산에 연관되는 형태를 지닌 경우가 많아서 이러한 단점이 발생하였다. 두 번째는 정확한 복잡한 timing 조정이 힘들다는 점이였다. HLS에서는 자동으로 Clock cycle을 적용하여 최적화를 진행하기 때문에 timing이 복잡한 모듈을 설계하기 힘들다는 것을 느꼈다. 마지막으로는 이러한 타이밍 설정이 힘들다 보니, register에 저장된 데이터를 도중에 shift하면서 연산기의 개수나 연산 시간을 획기적으로 줄일 수 있는데 이를 적용하는 데에 종합적인 어려움을 겪었다. 우리가 구현한 PQC NTRU 알고리즘은 최적화를 위해 clock cycle timing에 따라 input을 조절하여야 하는데 이 구현이 쉽지 않아 NTRU HLS의 최적화에 많은 고민을 하였었고, HLS가 Producer-Consumer Algorithm에 강력한 이유와, feedback이 존재하는 구조의 설계가 어려운 이유를 알 수 있었다.

3 NTRU HW 구현 결과

3.1 NTRU Synthesis 결과

[Optimizing 이전]

C++로 작성된 NTRU를 HLS를 활용해 Synthesis를 진행하였다. 처음 작성한 코드로 Synthesis를 돌렸을 때 결과는 다음과 같았다.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined
NTRU				-	-	-	-	-	-	dataflow
read_h_1				-	43	172.000	-	43	-	no
read_r_1				-	43	172.000	-	43	-	no
read_m_1				-	43	172.000	-	43	-	no
entry_proc				-	0	0.0	-	0	-	no
enc_1				-	-	-	-	-	-	no
enc_1_Pipeline_VITIS_LOOP_43_1				-	274	1.096E3	-	274	-	no
enc_1_Pipeline_j_loop_VITIS_LOOP_87_6				-	520	2.080E3	-	520	-	no
enc_1_Pipeline_VITIS_LOOP_92_7				-	-	-	-	-	-	no
enc_1_Pipeline_VITIS_LOOP_93_8				-	-	-	-	-	-	no
enc_1_Pipeline_VITIS_LOOP_94_9				-	-	-	-	-	-	no
VITIS_LOOP_57_3				-	2593864	1.038E7	5096	-	509	no
VITIS_LOOP_59_4				-	2540	1.016E4	5	-	508	no
VITIS_LOOP_63_5				-	2550	1.020E4	5	-	510	no

[Figure 8] NTRU baseline synthesis 결과

VITIS_LOOP_57_3은 아래 [Figure 9]의 for문인데, Convolution 연산을 수행하는 부분이다. C++와 HLS를 학습하며 Final Project를 진행하다 보니, 내부 register를 생성하여 input data를 저장하는 작업을 수행하는 데에도 상당한 노력과 학습이 필요하였고, Final Project 기간 직전에 HW bitstream까지 생성하여 pipeline과 Unroll을 적용하지 못하고 오롯이 HLS의 자동적인 최적화에 기댈 수밖에 없었다. 그 결과 convolution loop를 수행하는 데에 약 250만 cycle을 소요하게 되었다.

```
57    for(int k = 0; k < POLY_SIZE-3; k++) {
58       for (int i=1; i<POLY_SIZE-3 -k; i++) {
59          rh_block[k] += r_block[k+i] * h_block[POLY_SIZE-3 - i];
60       }
61       for (int i=0; i<k+1; i++)
62 #pragma HLS UNROLL
63          rh_block[k] += r_block[k-i] * h_block[i];
64    }
```

[Figure 9] Optimizing 이전 Convolution 코드

이후 교수님의 조언과 수업시간에 배운 idea와 HW 지식을 사용하여 convolution를 수정하였다. 우선 input_r 값을 저장하는 r_block은 ternary polynomial r의 계수를 저장하고 있는데, r은 0, 1, 2만을 계수를 갖기에 [Figure 10]과 같이 input을 0, 1, 2로 받는 mux처럼 구성하고, 그 결과에 따라 h_block의 input을 그대로, 혹은 음의 값으로, 혹은 0이 더해지도록 구성하였다. 이를

통해 곱셈기 대신 MUX를 사용하며 HW의 면적 효율을 증가시킬 수 있었다. 다음으로 HLS pipelining을 적용하여 최적화를 시도하였더니, 약 50만 cycle에 연산을 종료하는 것을 볼 수 있었다.

```
   for(int j = 0; j < POLY_SIZE-3; j++) {
//#pragma HLS pipeline II=509
        i_loop: for(int i = 0; i < POLY_SIZE-3; i++) {
#pragma HLS pipeline II=2
            if (r_block[(509 -i) % 509] == 2) {
                rh_block[j] -= h_block[(j+i) % 509];
            }
            else if (r_block[(509 -i) % 509] == 0) {
                rh_block[j] += 0;
            }
            else if (r_block[(509 -i) % 509] == 1){
                rh_block[j] += h_block[(j+i) % 509];
            } // NTRU r doesn't have coefficient -1.
        }
    }
```

[Figure 10] Optimizing 이후 Convolution 코드

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
NTRU				-	-	-	-	-	-	dataflow	94	0	11410	11542	0
read_h_1				-	43	194.000	-	43	-	no	0	0	593	298	0
read_r_1				-	43	194.000	-	43	-	no	0	0	593	298	0
read_m_1				-	43	194.000	-	43	-	no	0	0	593	298	0
entry_proc				-	0	0.0	-	0	-	no	0	0	66	20	0
enc_1				-	-	-	-	-	-	no	2	0	1252	2958	0
enc_1_Pipeline_VITIS_LOOP_43_1				-	274	1.233E3	-	274	-	no	0	0	605	1577	0
enc_1_Pipeline_VITIS_LOOP_70_3_i_loop				-	518177	2.332E6	-	518177	-	no	0	0	503	600	0
enc_1_Pipeline_j_loop_VITIS_LOOP_87_4				-	520	2.340E3	-	520	-	no	0	0	112	285	0
enc_1_Pipeline_VITIS_LOOP_92_5				-	-	-	-	-	-	no	0	0	2	22	0
enc_1_Pipeline_VITIS_LOOP_93_6				-	-	-	-	-	-	no	0	0	2	22	0
enc_1_Pipeline_VITIS_LOOP_94_7				-	-	-	-	-	-	no	0	0	2	22	0

[Figure 11] NTRU Optimizing 이후 Synthesis 결과

3.2 NTRU HLS Emulation 결과

작성한 NTRU 코드를 대상으로 Emulation을 동작해 보았고 그 결과를 분석하였다. 우선 처음에는 baseline NTRU 코드를 대상으로 Emulation을 진행하였다.

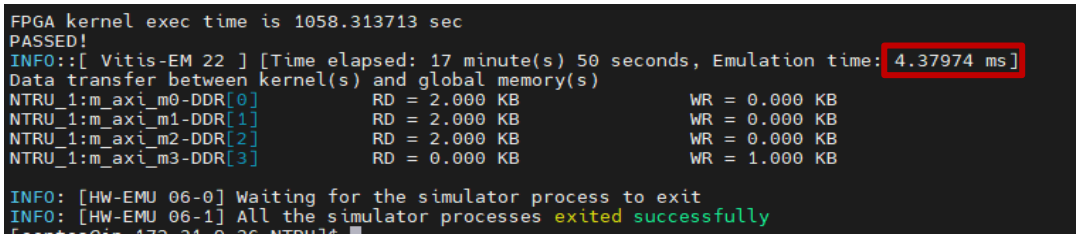
[Baseline]

```
Device[0]: program successful!


h[i]
FPGA kernel exec time is 0.079977 sec
```

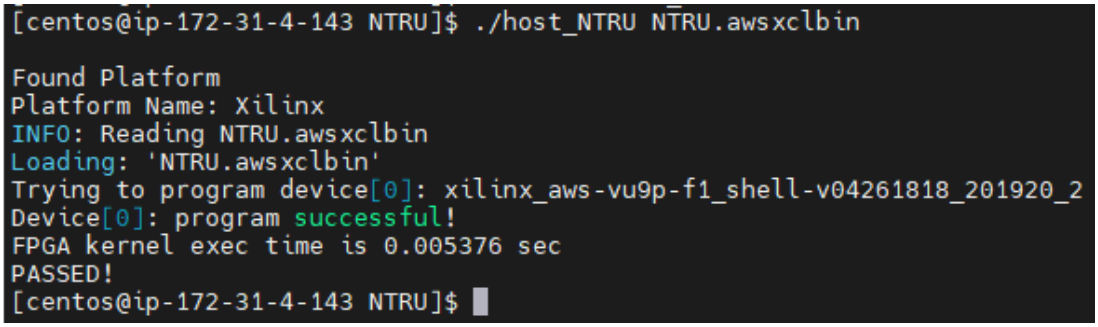
[Figure 12] Baseline NTRU SW_emu 결과

baseline NTRU 코드를 sw_emu를 돌렸을 때는 약 80ms 정도의 시간이 걸린 것을 확인 할 수 있었다.



[Figure 13] Baseline NTRU hw_emu 결과

그리고 hw_emu를 실행했을 때는 4.3ms으로 작동시간이 많이 줄어든 것을 볼 수있었다. 이는 위의 synthesis 결과에서의 latency를 clock 속도를 4ns(250mhz)으로 가정하고 계산하면 비슷한 결과가 나오는 것을 확인할 수 있었다.

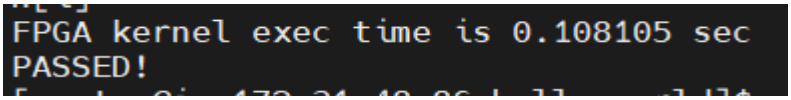


[Figure 14] Baseline NTRU FPGA test 결과

마지막으로 FPGA test를 진행하였다. 결과는 5ms로 hw_emu와 비슷한 결과를 얻어냈다.

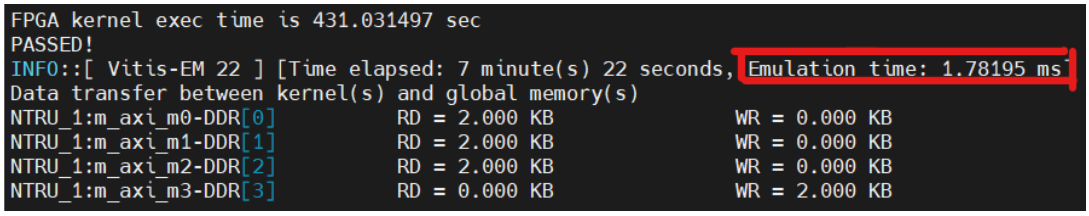
[Optimizing]

다음에는 Convolution 연산을 Optimizing한 코드를 대상으로 emulation을 진행하였다. 다만 코드를 수정하는데에 시간을 많이 써서 FPGA test를 진행하지 못하여 Optimizing 결과는 hw_emu 결과까지만 얻을 수 있었다.



[Figure 15] Opt NTRU SW_emu 결과

Opt NTRU sw_emu결과는 108ms로 Baseline NTRU 대비 오히려 늘어난 것을 확인할 수 있다. 이는 Software 적으로 좀 더 코드가 복잡해져서 runtime이 늘어났다고 보았다. 다음으로 HW_emu 결과는 아래 [Figure 16]과 같은데, Optimizing 이전보다 약 2.5배 감소한 결과를 보이는 것을 볼 수 있었다.



[Figure 16] Opt NTRU HW_emu 결과

3.3 결론

양자내성암호 시스템 NTRU는 기존의 암호시스템에 비해 큰 size의 데이터를 사용하고 복잡한 연산과정을 가지고 있다. 이를 Verilog를 이용해서 구현하는 것은 많은 복잡한 과정과 최적화를 위한 idea를 계속해서 구상해야 한다. 하지만 이번 수업을 통해 HLS를 활용하여 구현을 진행하면서 NTRU의 구조에 대해 파악하여 구현을 하고 많은 latency를 발생시키는 부분을 파악하여 최적화를 진행해볼 수 있었다. HLS를 활용하면 다른 알고리즘을 구현하는데 많은 도움이 될 것 같다는 생각이 들었다.