

Cortex-M4에서 PQC 알고리즘 구현 가이드

PQC Algorithm Implementation Guide on Cortex-M4

국민대학교 사이버보안학과
암호 및 보안 공학 연구실

서 석 충
김 영 범
최 용 렬

- 목 차 -

제 1 장 개요

제 2 장 Cortex-M4 및 개발 환경 구축

제 2.1 절 Cortex-M4

제 2.2 절 CUBE-IDE 개발환경 구축 방법

제 2.3 절 kpqm4 빌드 및 벤치마크 방법

제 3 장 pqm4에 적용된 구현 기술 분석 및 가이드

제 3.1 절 keccak-f1600

제 3.2 절 Modular Arithmetic and Butterfly

제 3.3 절 Streaming coefficient strategy

제 3.4 절 Better Accumulation

제 3.5 절 Asymmetric Multiplication

제 3.6 절 Merging Strategy

제 3.7 절 Binary Field Multiplication

제 4 장 Cortex-M4 구현 기술 Tip 및 가이드

제 4.1 절 주파수 및 Cycles

제 4.2 절 malloc

제 4.3 절 flash-memory Writing and Read

제 1 장 개요

- 이 문서에서는 Cortex-M4에서 PQC를 구현한 pqm4 라이브러리 (벤치마킹 및 테스트 제공)에 적용된 구현 방법론을 분석하고, Kyber 기반 PQC 구현에 도움이 될 수 있는 가이드라인을 제공한다.

제 2 장 Cortex-M4 및 개발 환경 구축

제 2.1 절 Cortex-M4

□ ARM Architecture

- ARM(Advanced RISC Machine)은 기존의 저가형 프로세서인 AVR, MSP430에 비해 저전력 및 고성능을 지원하여 IoT 산업에서 널리 활용되고 있다. ARM의 기술이 점차 발전하면서, ARM은 Cortex-A, Cortex-R, Cortex-M 제품군으로 분류되었다.
- Cortex-M 시리즈는 32-bit 저가형 임베디드 프로세서 용도의 제품군이며, ARM Cortex-M4는 저가형 프로세서에서 가장 널리 활용되고 있는 MCU이다. Cortex-R 시리즈는 Cortex-A 시리즈와 비슷하지만, 내부에 Flash Memory와 RAM을 모두 내장하고 있어, 설계상에서는 유연성이 떨어질 수 있다. 주로 Cortex-R 시리즈는 실시간 처리를 위한 통신모뎀, 의료기기에 활용되고 있다.

□ ARM-M Series

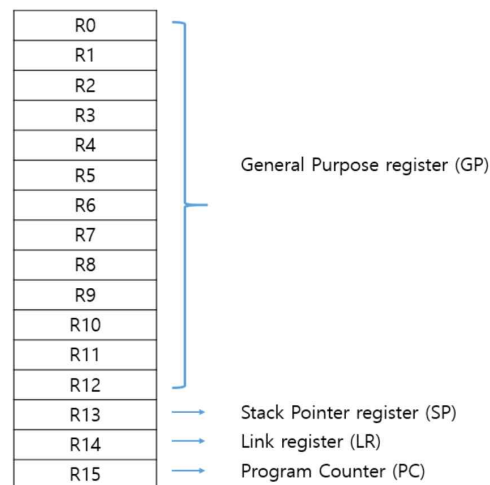
- ARMv7 아키텍처는 RISC 기반의 축소 명령어 세트이며 32-bit Cortex-M 시리즈의 주역이다. 그 중에서도 Cortex-M 시리즈는 고성능 임베디드 시스템을 제공하며 ARMv7기반의 프로세서는 32-bit 명령어를 사용하는 시스템과 동일한 성능으로 ROM의 메모리 공간을 줄여주어 비용을 절감시키고, 동일한 디바이스 프로파일 안에서 보다 크고 복잡한 애플리케이션을 개발할 수 있어 비용적인 측면에서 강점이 있게 개발되었다. ARMv7은 32-bit 연산이 가능한 ALU가 제공된다. 이러한 ARMv7의 특징으로는 ALU의 한쪽 입력에 Barrel shifter가 연결되어 있어 ALU에서 처리되는 인자 하나는 레지스터에 바로 들어오고 다른 인자 하나는 Barrel shifter를 거쳐 해당 레지스터에 shift된 값이 들어온다 (inlined-Barrel shifter).
- ARM Cortex-M4는 2010년에 출시되었으며 MDSP 확장 명령어 유닛이나, float형 연산용 FPU를 추가하여 Digital Signal Controller의 목적으로 제작되었다. FPU는 C에서 float의 데이터를 빠르게 연산이 가능하다. 하드웨어에 실수의 곱셈과 나눗셈이 내장되어 있으며 곱셈은 1 clock cycles안에 처리가 가능하다 나눗셈을 처리하는 데는 16 clock cycles가 소모된다는 특징이 있다. M0나 M0+는 32비트 저가형 마이크로컨트롤러 애플리케이션 용도이다. M0+는 M0에 비하여 더 높은 성능을 가지고 있으며, M0에 비하여 저전력으로 구동할 수 있다. M3은 HW divide와 연산을 위한 파이프라인이 더 추가 된다. M4는 floating point(FPU)와 디지털 시그널 프로세싱을 위한 DSP가 포함되어 있으며, NIST PQC 공모전의 성능평가 장비 중 하나이다. M7은 멀티로 연산을 하기 위한 dual-precision FPU와 superscalar pipelining 기능을 가지고 있다.

□ Cortex-M4

- 레지스터: Cortex-M4는 총 16개의 범용 레지스터(General-Purpose Registers)를 가지고 있다. 이들 중 13개는 R0~R12로 표기되며, 나머지 3개는 특별한 용도로 사용된다.

- R13: 스택 포인터(Stack Pointer, SP)
- R14: 링크 레지스터(Link Register, LR)
- R15: 프로그램 카운터(Program Counter, PC)

Cortex-M4는 **FPU(Floating Point Unit)**을 갖추고 있어, 추가로 32개의 단정밀도 부동소수점 레지스터(S0~S31)를 제공한다.



[그림 1] Cortex-M4 레지스터 개요

- 메모리 (stack) : Cortex-M4의 SRAM은 일반적으로 프로그램 데이터를 저장하고 CPU가 실행 중에 데이터를 빠르게 읽고 쓸 수 있는 공간이다. SRAM 크기는 SoC(System on Chip)에 따라 다르며, 32KB 부터 512KB 이상의 SRAM을 가진 마이크로컨트롤러들이 존재한다. pqm4의 벤치마크 대상인 STM32F407G 보드는 192KB의 SRAM이 존재한다.

SRAM을 효율적으로 사용하려면 Linker script를 통해 메모리 공간을 분할할 수 있다. 스택, 힙, 데이터 영역을 각각 따로 설정해서 메모리 충돌을 방지하고, 각 공간의 크기를 필요에 맞게 조정할 수 있다.

Cortex-M4에는 일부 칩에서 CCM이라는 메모리 공간이 제공된다. CCM은 일반 SRAM보다 CPU에 더 가까워서 빠른 데이터 접근이 가능하다. 실시간 처리나 고속 연산이 필요한 경우에 많이 쓰인다.

CCM의 주요 특징은 속도다. DSP 연산이나 인터럽트 처리처럼 실시간 연산이 필요한 데이터는 CCM에 저장하면 성능이 크게 향상된다. CCM은 용량이 상대적으로 적지만, 빠른 속도로 데이터를 처리할 수 있다.

이러한 CCM 메모리 영역을 사용하기 위해서는 CCM 메모리의 시작 주소인 0x10000000을 시작으로 하는 CCM 메모리를 링커 스크립트에서 설정해야 하며, CCM 메모리의 크기는 제공되는 보드마다 상이하다. 또한, CCM 메모리는 자동으로 초기화되지 않으므로 CCM 영역에 변수를 배치한다면 직접

코드 상에서 초기화해야 한다. CCM 영역을 사용하기 위해 변수에 `__attribute__` 속성을 부여할 수 있으며, 이는 비단 CCM 영역뿐만 아니라 다른 메모리 영역에 변수를 배치할 때도 사용할 수 있다.

pqm4의 벤치마크 대상인 STM32F407G 보드는 192KB의 RAM 중 CCM 메모리 64KB를 제외하고 128KB의 RAM 용량을 사용하며, Linker script를 통해 SRAM 공간을 2개의 SRAM1 (112KB), SRAM2 (16KB)로 분할하였다.

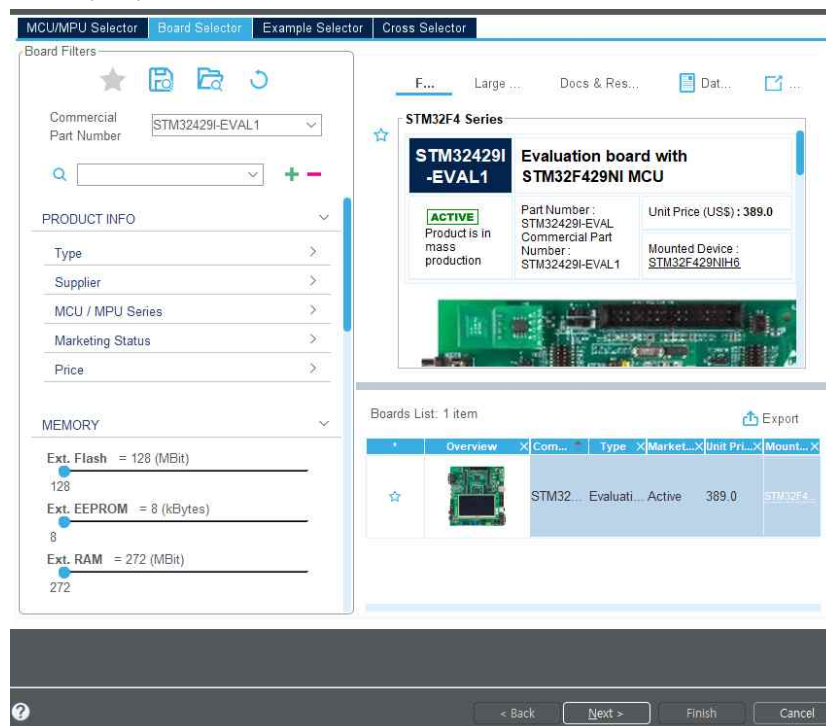
제 2.2 절 CUBE-IDE 개발환경 구축 방법

□ CUBE-IDE 설치

- CubeIDE는 STMicroelectronics사에서 배포하는 IDE로, STM의 MCU(Microcontroller)보드를 구동할 수 있는 IDE이다. Eclipse 기반으로 설계되었으며, STM32CubeMX와 통합되어 GPIO 설정, 주변 장치 설명 등 하드웨어 초기화 작업을 GUI로 수행할 수 있다.
- CubeIDE를 설치하기 위해서는 우선 다음 링크로 접속한다.
 - <https://www.st.com/en/development-tools/stm32cubeide.html>
- 해당 페이지에 접속한 뒤 페이지 하단에서 각 환경에 알맞은 설치 파일을 다운로드한다. 윈도우의 경우 STM32CubeIDE-Win 파일을 다운로드한다. 다운로드하기 위해서는 STM 사이트에 회원가입이 필요하다.
- 설치 파일을 실행하면 설치가 진행되며, 특별한 추가 설정 없이 설치를 완료한다.

□ CubeIDE 보드 선택 및 기본 설정

- CubeIDE를 실행한 뒤, File – New – STM32 Project를 클릭한다.
- 나타나는 팝업에서 Board Selector를 클릭하고, 사용하는 보드 기종을 Commercial Part Number에 입력한다.
- 사용하는 보드를 선택하고 Next를 클릭한 뒤, 프로젝트명을 입력하고 Finish를 클릭하면 프로젝트가 생성된다. 나타나는 peripheral 초기화 팝업에는 Yes를 클릭한다.



[그림 2] CubeIDE Board Selector 예시

- 예시에는 STM32429I-EVAL1 보드를 사용하였으므로, 해당 보드를 기준으로 설명한다.
- 프로젝트 생성 직후 열리는 .ioc 파일에서 기본적인 GPIO 설정 및 기타 설정이 가능하다.
- 먼저, Connectivity 탭을 클릭한 뒤, SDIO를 클릭하고 Mode를 Disable로 설정한다. SDIO를 사용하게 되면 코드 실행 시 SDIO 초기화 함수에서 Hard Fault Interrupt가 걸리게 된다.
- 그 뒤, USART1를 클릭하고 Baud Rate, Word Length 등의 USART 통신을 위한 기본 설정을 수행한다. 예시에서는 아래 그림과 같이 설정하였다.

▼ Basic Parameters	
Baud Rate	9600 Bits/s
Word Length	9 Bits (including Parity)
Parity	Odd
Stop Bits	1
▼ Advanced Parameters	
Data Direction	Receive and Transmit
Over Sampling	16 Samples

[그림 3] USART 통신을 위한 설정 예시

- 이제 ctrl+S를 눌러 .ioc의 변경 사항을 저장하고, 코드를 생성하는지 묻는 팝업에 모두 Yes를 클릭하면 기본 설정이 완료된다.

□ CubeIDE HelloWorld Project

- 먼저, 프로젝트 상단에 USER CODE BEGIN INCLUDES 아래에 stdio.h를 include한다.
- 그 뒤, USER CODE BEGIN PFP 아래에 다음 코드를 입력한다.

```
UART_HandleTypeDef UartHandle;

/* Private function prototypes
-----*/
#ifdef __GNUC__
/* With GCC, small printf (option LD Linker->Libraries->Small printf
set to 'Yes') calls __io_putchar() */
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */

/**
 * @brief Retargets the C library printf function to the USART.
 * @param None
 * @retval None
 */
PUTCHAR_PROTOTYPE {
/* Place your implementation of fputc here */
/* e.g. write a character to the EVAL_COM1 and Loop until the end of
transmission */
HAL_UART_Transmit(&UartHandle, (uint8_t *)&ch, 1, 0xFFFF);

return ch;
}
```

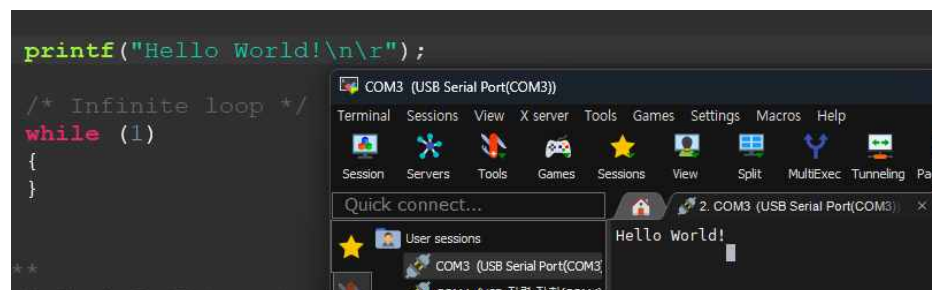
- 그리고 main 함수 내부에 USER CODE BEGIN 2 아래에 아래 코드를 입력한다.

```
UartHandle.Instance = USART1;

UartHandle.Init.BaudRate = 9600;
UartHandle.Init.WordLength = UART_WORDLENGTH_9B;
UartHandle.Init.StopBits = UART_STOPBITS_1;
UartHandle.Init.Parity = UART_PARITY_ODD;
UartHandle.Init.HwFlowCtl = UART_HWCONTROL_NONE;
UartHandle.Init.Mode = UART_MODE_TX_RX;
UartHandle.Init.OverSampling = UART_OVERSAMPLING_16;

if(HAL_UART_Init(&UartHandle) != HAL_OK) {
    /* Initialization Error */
    Error_Handler();
}
```

- 해당 코드를 입력했다면, 바로 아래에 printf("Hello World!\n\r");를 입력한다.
- 보드와 PC를 연결한다. STM32429I-EVAL1의 경우 ST-LINK/V2 케이블과 UART 통신을 위한 RS232 케이블을 연결한다.
- Tera Term 혹은 MobaXterm 등의 시리얼 통신이 가능한 터미널을 실행한 뒤, CubeIDE 상단바의 재생 모양 버튼을 눌러 프로젝트를 빌드 후 실행한다.
- 실행 결과가 터미널에 아래와 같이 정상적으로 출력되는 것을 확인 할 수 있다.



[그림 4] 터미널에 출력된 printf 결과

제 3 절 kpbm4 빌드 및 벤치마크 방법

□ kpbm4 빌드

- 환경은 Ubuntu 24.04 LTS를 기준으로 설명한다.
- 터미널에 다음 명령어를 입력하여 dependency를 설치한다.
 - `sudo apt install gcc-arm-none-eabi build-essential libusb-1.0-0-dev cmake make python3-pip`
- 다음 명령어를 입력하여 stlink를 설치한다.
 - `git clone https://github.com/stlink-org/stlink .`
 - `cmake .`
 - `make`
 - `sudo cp bin/* /usr/local/bin`
 - `sudo cp lib/*.so* /lib32`
 - `sudo cp config/udev/rules.d/49-stlinkv* /etc/udev/rules.d/`
- 여기까지 진행한 뒤, 보드를 PC와 연결하고 아래 `lsusb`와 `st-info --probe` 명령어로 보드가 인식되는지 확인한다. 보드가 인식된다면 보드에 대한 정보가 출력된다.
 - 만약 `stlink /usr/local/share/stlink/config/chips: No such file or directory` 오류가 발생한다면 `stlink/config/chips`의 파일을 `/usr/local/share/stlink/config/chips`로 복사한다.
- 이제 kpbm4를 빌드하기 위해, 먼저 레포지토리를 clone한다.
 - `git clone https://github.com/COALA-5/kpbm4`
- kpbm4 디렉토리로 이동한 다음, 아래 명령어로 빌드를 수행한다. 빌드 대상 보드는 NUCOLO-L4R5ZI이다.
 - `sudo make -j16 PLATFORM=nucleo-l4r5zi`
- bin 폴더에 .bin 파일들이 생성되었다면 정상적으로 빌드된 것이다.

□ kpbm4 벤치마크 방법

- 보드가 PC와 연결된 상태에서, kpbm4 디렉토리에 진입한다.
- 먼저 바이너리 파일을 보드에 flash해야 하므로, 다음 명령어를 사용하여 보드에 flash한다.
 - `st-flash write bin/<binary_file_name>.bin 0x8000000`
 - 예를 들어, SMAUG-T1의 벤치마크를 구동할 경우 `st-flash write bin/crypto_kem_smaugt1_m4_test.bin 0x8000000`을 실행한다.
 - 해당 명령어를 실행하면 [그림 5]와 같이 바이너리 파일이 플래시된다.

```
➔ kpbm4 git:(main) X st-flash write bin/crypto_kem_smaugt1_m4_test.bin 0x8000000
st-flash 1.8.0-60-g5280bcf
2024-10-12T18:48:34 INFO common.c: STM32L4Rx: 640 KiB SRAM, 2048 KiB flash in at least 4 KiB pages.
file bin/crypto_kem_smaugt1_m4_test.bin md5 checksum: 8fcaecf25f2ac362796ed825e4b8b046, stlink checksum: 0x0033c1ef
2024-10-12T18:48:34 INFO common_flash.c: Attempting to write 31432 (0x7ac8) bytes to stm32 address: 134217728 (0x80000000)
EraseFlash - Page:0x0 Size:0x1000 -> Flash page at 0x80000000 erased (size: 0x1000)
EraseFlash - Page:0x1 Size:0x1000 -> Flash page at 0x80010000 erased (size: 0x1000)
EraseFlash - Page:0x2 Size:0x1000 -> Flash page at 0x80020000 erased (size: 0x1000)
EraseFlash - Page:0x3 Size:0x1000 -> Flash page at 0x80030000 erased (size: 0x1000)
EraseFlash - Page:0x4 Size:0x1000 -> Flash page at 0x80040000 erased (size: 0x1000)
EraseFlash - Page:0x5 Size:0x1000 -> Flash page at 0x80050000 erased (size: 0x1000)
EraseFlash - Page:0x6 Size:0x1000 -> Flash page at 0x80060000 erased (size: 0x1000)
EraseFlash - Page:0x7 Size:0x1000 -> Flash page at 0x80070000 erased (size: 0x1000)

2024-10-12T18:48:34 INFO flash_loader.c: Starting Flash write for F2/F4/F7/L4
2024-10-12T18:48:34 INFO flash_loader.c: Successfully loaded flash loader in sram
2024-10-12T18:48:34 INFO flash_loader.c: Clear DFSR
2024-10-12T18:48:34 INFO flash_loader.c: Clear CFSR
2024-10-12T18:48:34 INFO flash_loader.c: Clear HFSR
2024-10-12T18:48:35 INFO common_flash.c: Starting verification of write complete
2024-10-12T18:48:35 INFO common_flash.c: Flash written and verified! jolly good!
```

[그림 5] st-flash 명령어를 실행한 모습

- 이제 실제로 보드에 flash된 바이너리를 실행하여 출력값을 읽기 위해 다음 명령어를 실행한다.
 - `python3 hostside/host_unidirectional.py`
 - 만약 `/dev/ttyUSB0` 또는 `/dev/ttyACM0`을 읽을 수 없다면 `sudo chmod 777 /dev/ttyUSB0`를 실행하여 읽을 수 있는 상태로 전환한다.
- 마지막으로 보드의 reset 버튼을 클릭하여 바이너리를 실행하면 출력된 결과가 [그림 6]과 같이 터미널에 나타난다.



```
→ kpopm4 git:(main) X python3 hostside/host_unidirectional.py
Found ACM: ttyACM0
> Returned data:
=====
DONE key pair generation!
DONE encapsulation!
DONE decapsulation!
OK KEYS

+
DONE key pair generation!
DONE encapsulation!
DONE decapsulation!
OK KEYS

+
DONE key pair generation!
DONE encapsulation!
DONE decapsulation!
OK KEYS
```

[그림 6] 보드에 플래시된 바이너리를 실행한 결과.
실행하는 바이너리마다 출력 결과는 다를 수 있다.

제 3 장 pqm4에 적용된 구현 기술 분석 및 가이드

제 3.1 절 keccak-f1600

□ keccak-f1600 기반의 SHA-3 and SHAKE

- FIPS-202에 정의된 SHA-3는 keccak 순열함수 (keccak-f1600)을 기반으로 설계된 암호학적 해시함수이다. 또한, 같은 문서에 정의된 SHAKE는 확장 가능한 출력함수로 SHA-3와 동일하게, keccak 순열함수를 사용한다.
- NIST PQC 및 KpqC 공모전에 제출된 알고리즘들은 주로 값을 샘플링하거나, 스킴에 필요한 해시 함수로 FIPS 202 알고리즘(SHAKE, SHA-3)을 사용하기 때문에 Cortex-M4에서 적절한 구현을 선택하는 것이 중요하다. 추가적으로 cSHAKE라는 알고리즘도 존재한다(SP 800-185). cSHAKE는 SHAKE와 동일한 작동 방식을 가지지만, 새로운 입력인 "사용자 정의 문자열"이 추가된다. 이 문서에서는 keccak 순열 함수, SHAKE, SHA-3의 기본 동작 과정에 대해서는 다루지 않는다.

□ SHA-3 및 SHAKE의 Cortex-M4 구현

- SHA-3와 SHAKE의 개발자들이 제공하는 XKCP 라이브러리(<https://github.com/XKCP/XKCP>)에는 SHA-3와 SHAKE의 핵심 함수인 keccak 순열의 최적화 구현물들이 각 아키텍처마다 존재한다. 초기에는 이러한 구현물들이 대부분 Cortex-M4에서의 구현 최적화 연구에 사용되었으나, 2024년까지 다양한 최적화가 이루어지면서 pqm4는 최신 구현 최적화 방법론을 통합한 구현을 사용하고 있다. 가장 최신의 keccak-f1600에 대한 최적화는 2023년에 수행되었다¹⁾. pqm4에서 사용하고 있는 keccak-f1600에 대한 코드는 (<https://github.com/mupq/pqm4/blob/master/common/keccakf1600.S>)에서 확인할 수 있다.
- pqm4에서 사용하는 SHA-3, SHAKE, cSHAKE 코드를 독립적으로 사용하는 방법은 다음과 같다. 테스트 함수를 기준으로 서술한다.
 - pqm4의 서브모듈인 mupq의 common 폴더에서 fips202.c fips202.h sp800-185.c sp800-185.h keccakf1600.h를 가져온다. (<https://github.com/mupq/mupq/tree/master/common>)
 - pqm4의 common 폴더에서 keccakf1600.S keccaktest.c를 가져온다. (<https://github.com/mupq/pqm4/tree/master/common>)
 - fips202.c fips202.h sp800-185.c sp800-185.h keccakf1600.h keccakf1600.S keccaktest.c를 CubeIDE에서 빌드하고 실행한다.

제 3.2 절 Modular Arithmetic and Butterlfy

□ 개요

- 격자기반의 PQC 알고리즘은 모듈러 곱셈, 모듈러 감산을 효율적으로 구현하는 것이 중요하다. 본 절에서는 Cortex-M4에서 구현 방법론에 대해 살펴본다. 빠른 상수시간 구현 달성을 위해, 대부분의 구현은 주로 Montgomery, Barrett, Improved Plantard Arithmetic을 사용한다.

□ Cortex-M4 명령어 세트

- 본 절에서 설명하는 어셈블리 코드의 이해를 돕고자, Cortex-M4의 명령어를 설명한다.

명령어	설명
mov r0, r1	$r0 \leftarrow r1$
mov r0, #7	$r0 \leftarrow 7$ 동일계열 : movw 는 하위 16-bit, movt 는 상위 16-bit
add r0, r1, r2	$r0 \leftarrow r1 + r2$, 동일계열 : adds (sets flags), adc (with carry), adcs
ror r0, r1, r2	$r0 \leftarrow r1 \gg r2$ (rotate right by r2) 동일계열 : lsl , lsr , asr
mul r0, r1, r2	$r0 \leftarrow (r1 \times r2) \bmod 2^{32}$
mla r0, r1, r2	$r0 \leftarrow r0 + (r1 \times r2) \bmod 2^{32}$
mls r0, r1, r2	$r0 \leftarrow r0 - (r1 \times r2) \bmod 2^{32}$
smull r0, r1, r2, r3	$(r1 \parallel r0) \leftarrow r1 \times r2$, smull for signed, umull for unsigned
smlal r0, r1, r2, r3	$(r1 \parallel r0) \leftarrow (r1 \parallel r0) + (r1 \times r2)$, smlal for signed, umlal for unsigned
smulbb r0, r1, r2	$r0 \leftarrow (r1 \text{의 하위 } 16\text{-bit}) \times (r2 \text{의 하위 } 16\text{-bit})$, 동일계열 : smulbt , smultb , smultt (t는 레지스터의 상위 16-bit를 의미)
smuad r0, r1, r2	$r0 \leftarrow ((r1 \text{의 하위 } 16\text{-bit}) \times (r2 \text{의 하위 } 16\text{-bit}))$ $+ ((r1 \text{의 상위 } 16\text{-bit}) \times (r2 \text{의 상위 } 16\text{-bit}))$
smulwb r0, r1, r2	$r0 \leftarrow ((r1 \times (r2 \text{의 하위 } 16\text{-bit})) \text{의 상위 } 32\text{-bit})$, 동일계열 : smulwt
smlabb r0, r1, r2	$r0 \leftarrow r0 + ((r1 \text{의 하위 } 16\text{-bit}) \times (r2 \text{의 하위 } 16\text{-bit}))$, 동일계열 : smlabt , smlatb , smlatt (t는 레지스터의 상위 16-bit를 의미)
smlawb r0, r1, r2	$r0 \leftarrow r0 + ((r1 \times (r2 \text{의 하위 } 16\text{-bit})) \text{의 상위 } 32\text{-bit})$, 동일계열 : smlawt
pkhtb r0,r1,r2, asr #n	$r0 \leftarrow ((r1 \text{의 상위 } 16\text{-bit}) \ll 16) \mid (r2 \gg n)$
uadd16 r0, r1, r2	$r0 \leftarrow (((r1 \text{의 상위 } 16\text{-bit}) + (r2 \text{의 상위 } 16\text{-bit}))$ $\parallel ((r1 \text{의 하위 } 16\text{-bit}) + (r2 \text{의 하위 } 16\text{-bit})))$, 동일계열 : usub16

□ 16-bit Modular Multiplication (모듈러 곱셈)

- 초기에 16-bit 모듈러 곱셈에 대한 격자기반 암호의 레퍼런스 구현의 접근은 Montgomery 곱셈을 사용하는 것이었다. 그러나 최근 연구에서 제안된 Improved Plantard 곱셈 방식을 사용하면 더 효율적으로 구현할 수 있다²⁾. NTT/iNTT 과정에서 회전 인자(Twiddle Factor)와의 모듈러 곱셈을 Improved Plantard 곱셈은 2 cycles에 구현한다. (Montgomery 곱셈은 3 cycles) 하기 알고리즘은 참조 논문에서 제공된 어셈블리의 비교이다.

<p>Algorithm 1 The 2-cycle improved Plantard multiplication by a constant on Cortex-M4</p> <p>Require: An l-bit $a \in [-2^{l-1}, 2^{l-1})$, a precomputed $2l$-bit integer bq' where b is a constant and $q' = q^{-1} \bmod \pm 2^{2l}$</p> <p>Ensure: $r_{top} = ab(-2^{-2l}) \bmod \pm q$, $r_{top} \in (-\frac{q}{2}, \frac{q}{2})$</p> <ol style="list-style-type: none"> 1: $bq' \leftarrow bq'^{-1} \bmod \pm 2^{2l}$ \triangleright Precomputed 2: smulwb r, bq', a $\triangleright r \leftarrow \lfloor \frac{abq'}{2^{2l}} \rfloor^l$ 3: smlabb r, r, q, q^{2^α} $\triangleright r_{top} \leftarrow \lfloor q[r] + q^{2^\alpha} \rfloor^l$ 4: return r_{top} 	<p>Algorithm 2 The 3-cycle signed Montgomery multiplication on Cortex-M4</p> <p>Require: Two l-bit signed integers a, b such that $ab \in [-q2^{l-1}, q2^{l-1}]$</p> <p>Ensure: $r_{top} = ab2^{-l} \bmod \pm q, r_{top} \in (-q, q)$</p> <ol style="list-style-type: none"> 1: mul c, a, b 2: smulbb $r, c, -q^{-1}$ $\triangleright r \leftarrow [c]_l \cdot (-q^{-1})$ 3: smlabb r, r, q, c $\triangleright r_{top} \leftarrow [[r]_l \cdot q] + [c]_l$ 4: return r_{top}
--	---

[표 1] (좌) : Montgomery 곱셈, (우) : Improved Plantard 곱셈에 대한 비교

- Improved Plantard 곱셈의 장점은 Montgomery 곱셈과 비교하여 감산 대상의 범위가 더 증가하고, 감산 후 범위가 더 줄어들어 들었다는 장점이 있다. 또한, 모듈러 곱셈이 2 cycles에 동작함에 따라, Montgomery 곱셈과 비교하여 더 빠르다. 전체 스킴에 대한 영향은 Kyber를 기준으로 하기와 같다.
 - Montgomery 곱셈 : NTT 변환 후 PointMul 과정 전에 16-bit 모듈러 감산이 필요함
 - Improved Plantard 곱셈 NTT : 변환 후 PointMul 과정 전에 16-bit 모듈러 감산이 필요 없음

- Improved Plantard 곱셈을 기반으로 Cortex-M4에서의 CT butterfly 구현 방식은 하기 알고리즘과 같다³⁾. 하나의 32-bit 레지스터에 2개의 계수가 패킹되어있는 상태에서의 butterfly 연산이다.

<p>Algorithm 3 Double CT butterfly on Cortex-M4</p> <p>Require: Two 32-bit packed signed integers a, b (each containing a pair of 16-bit signed coefficients), the 32-bit twiddle factor ζ</p> <p>Ensure: $a = (a_{top} + b_{top}\zeta) \parallel (a_{bottom} + b_{bottom}\zeta)$, $b = (a_{top} - b_{top}\zeta) \parallel (a_{bottom} - b_{bottom}\zeta)$</p> <ol style="list-style-type: none"> 1: smulwb t, ζ, b 2: smulwt b, ζ, b 3: smlabb t, t, q, q^{2^α} 4: smlabb b, b, q, q^{2^α} 5: pkhtb $t, b, t, \text{asr}\#16$ 6: usub16 b, a, t 7: uadd16 a, a, t 8: return a, b 	<pre>.macro doublebutterfly_plant a0, a1, twiddle, tmp, q, qa smulwb Wtmp, Wtwiddle, Wa1 smulwt Wa1, Wtwiddle, Wa1 smlabt Wtmp, Wtmp, Wq, Wqa smlabt Wa1, Wa1, Wq, Wqa pkhtb Wtmp, Wa1, Wtmp, asr#16 usub16 Wa1, Wa0, Wtmp uadd16 Wa0, Wa0, Wtmp .endm</pre>
--	---

[표 2] (좌) : Kyber의 Plantard 곱셈 기반의 CT Butterfly, (우) pqm4 적용 예시
(https://github.com/mupq/pqm4/blob/master/crypto_kem/kyber768/m4fspeed/fastntt.S)

□ 16-bit Modular Reduction (모듈러 감산)

- 16-bit 모듈러 감산은 Barrett 산술을 사용하는 것이 일반적이며, 최근 제안된 연구에서는 Barrett 감산을 Cortex-M4에서 효율적으로 구현하는 방법론이 제안되었다.⁴⁾ Cortex-M4의 레지스터 크기가 32-bit이므로, 16-bit 감산을 위해서는 하나의 레지스터에 2개의 데이터를 패킹한 후에 구현하는 것이 일반적이다. 대부분의 PQC 구현물들이 Signed 자료형을 사용하는 것을 바탕으로, Barrett 감산도 Signed 방식으로 구현하는 것이 일반적이며, 최종 출력 범위는 $[-(q-1)/2, (q-1)/2]$ 로 보장된다. 부호있는 감산이기 때문에 NTT/iNTT 과정 내에서만 사용가능하며, 데이터 packing을 위한 마지막 감산을 위해서는 Algorithm 4의 Packed Barrett Reduction 사용해야한다.

Algorithm 4 Packed Barrett Reduction	Algorithm 5 Improved Packed Barrett Reduction
Require: $a = (a_t \parallel a_b)$ Ensure: $c = (c_t \parallel c_b) \bmod \pm q$ with c_t, c_b in $[0, q)$ <ol style="list-style-type: none"> 1: smulbb $t_0, a, \left\lfloor \frac{2^{26}}{q} \right\rfloor$ 2: smultb $t_1, a, \left\lfloor \frac{2^{26}}{q} \right\rfloor$ 3: asr $t_0, t_0, \#26$ 4: asr $t_1, t_1, \#26$ 5: smulbb t_0, t_0, q 6: smulbb t_1, t_1, q 7: pkhbt $t_0, t_0, t_1, \text{ls}\#16$ 8: usub16 r, a, t_0 9: return r 	Require: $a = (a_t \parallel a_b)$ Ensure: $c = (c_t \parallel c_b) \bmod \pm q$, with c_t, c_b in $[-\frac{q-1}{2}, \frac{q-1}{2}]$ <ol style="list-style-type: none"> 1: smlawb $t_0, -\left\lfloor \frac{2^{32}}{q} \right\rfloor, a, 2^{15}$ 2: smlabt t_0, q, t_0, a 3: smlawt $t_1, -\left\lfloor \frac{2^{32}}{q} \right\rfloor, a, 2^{15}$ 4: smulbt t_1, q, t_1 5: add $t_1, a, t_1, \text{ls}\#16$ 6: pkhbt $c, t_0, t_1, \text{ls}\#16$ 7: return c

[표 3] Kyber에 적용된 (좌) : 8 cycles Barrett 감산, (우) 6 cycles Barrett 감산

(https://github.com/FasterKyberDilithiumM4/FasterKyberDilithiumM4/blob/main/crypto_kem/kyber768/new/macros.i)

- Improved Plantard 방법론을 통해 packing된 감산 또한 가능하며, 출력범위는 $[-(q+1)/2, q/2]$ 를 보장한다. Improved Packed Barrett Reduction과 마찬가지로, 부호 있는 감산이기 때문에 NTT/iNTT 과정 내에서만 사용할 수 있으며, 데이터 packing을 위한 마지막 감산을 위해서는 Algorithm 4의 Packed Barrett Reduction 사용해야 한다. 하기는 packing된 Improved Plantard의 16-bit 감산 방법론이다.

Algorithm 6 Double Plantard reduction for packed coefficients
Require: A 32-bit packed integer $a = a_{top} \parallel a_{bottom}$ where a_{top}, a_{bottom} are two 16-bit signed coefficients Ensure: $r = (a_{top} \bmod \pm q) \parallel (a_{bottom} \bmod \pm q), -\frac{q+1}{2} \leq r_{top}, r_{bottom} < \frac{q}{2}$ 1: $\text{const} \leftarrow (-2^{2l} \bmod q) \cdot (q^{-1} \bmod \pm 2^{2l}) \bmod \pm 2^{2l}$ ▷ precomputed 2: smulwb t, const, a 3: smulwt a, const, a 4: smlabt $t, t, q, q2^\alpha$ 5: smlabt $a, a, q, q2^\alpha$ 6: pkhbt $r, a, t, \text{asr}\#16$ 7: return r

[표 4] Kyber에 적용된 Improved Plantard 감산 기법

(https://github.com/mupq/pqm4/blob/master/crypto_kem/kyber768/m4fspeed/macros.i)

□ 32-bit Modular Multiplication (모듈러 곱셈)

- 32-bit 모듈러 곱셈은 일반적으로 Montgomery 산술을 사용한다. 32-bit Improved Plantard 곱셈을 진행하기 위해서는 64-bit 사전 계산값이 필요하다. 따라서 Montgomery 곱셈을 사용하는 것이 최선이며, 32-bit 크기의 레지스터를 갖는 Cortex-M4에서는 하나의 레지스터에 하나의 계수를 처리한다.
- Cortex-M4에서 32-bit Montgomery 곱셈에 대한 연구는 2020년에 제안된 이후에 크게 달라진 것은 없다. 하기 알고리즘은 참조 논문에서의 CT 및 GS butterfly에 대한 Cortex-M4 코드이다. CT butterfly의 Line 4~6번이 Montgomery 곱셈 수행 부분이다.

$$-CT(a, b) = a + \text{Mont}(b \cdot \omega), a - \text{Mont}(b \cdot \omega)$$

<pre>.macro CT_butterfly p0, p1, twiddle ; q=8380417, qinv=4236238847 ; Input: p0, p1, twiddle ; Output: p0, p1 smull tmp0, tmp1, p1, twiddle mul p1, tmp0, qinv smlal tmp0, tmp1, p1, q sub p1, p0, tmp1 add p0, p0, tmp1 .endm</pre>	<pre>.macro GS_butterfly p0, p1, twiddle ; q=8380417, qinv=4236238847 ; Input: p0, p1, twiddle ; Output: p0, p1 sub tmp0, p0, p1 add p0, p0, p1 smull tmp1, p1, tmp0, twiddle mul tmp0, tmp1, qinv smlal tmp1, p1, tmp0, q .endm</pre>
---	---

[표 5] (좌) : Dilithium의 CT Butterfly, (우) Dilithium의 GS Butterfly
https://github.com/mupq/pqm4/blob/master/crypto_sign/dilithium2/m4f/macros.i

□ 32-bit Modular Reduction (모듈러 감산)

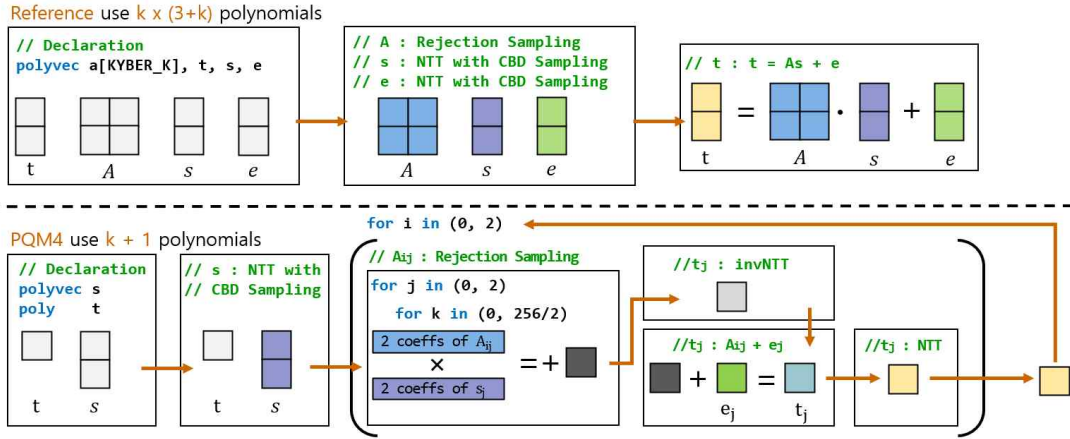
- 32-bit Modular Reduction은 크게 두 가지 방식으로 구현한다. Dilithium에서는 Barrett 감산의 변형을 사용하여, 32-bit로 표현되는 양/음수의 값에 대해서 $(-\frac{3}{4}q, \frac{3}{4}q)$ 범위내로 감산을 수행한다. 최종적으로 $(0, q)$ 범위내로 표현하고 싶으면 $(-\frac{3}{4}q, 0)$ 범위 내에 있는 값에다가 q 를 더해주면 된다. 하기는 Dilithium의 reference C 코드의 pqm4 Asm 표현이다.

Dilithium reference C code	pqm4 Asm code
<pre>int32_t reduce32(int32_t a) { int32_t t; t = (a + (1 << 22)) >> 23; t = a - t*Q; return t; }</pre>	<pre>.macro redq a, tmp, q add Wtmp, Wa, #4194304 asrs Wtmp, Wtmp, #23 mls Wa, Wtmp, Wq, Wa .endm</pre>

[표 6] Dilithium ref 코드 및 pqm4 Asm 코드 비교

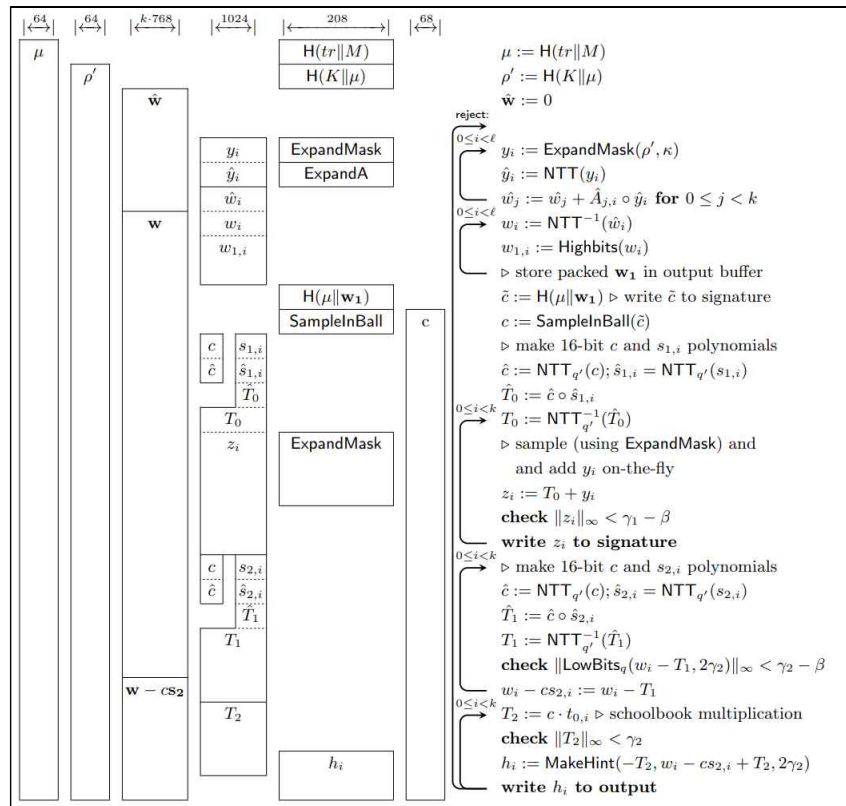
제 3.3 절 Streaming coefficient strategy

- Crystals-Kyber의 구현은 계수 단위로 공개행렬을 생성하여 할 수 있다. 비밀 벡터의 경우 NTT 변환을 수행해야하기 때문에 미리 저장하여 구현한다. 하기 그림은 계수에 대한 스트리밍 구현 방안의 도식도이다.



[그림 7] pqm4의 Kyber 스트리밍 구현 도식도

- 5)논문에서는 다항식을 압축하고, 대체 NTT를 사용하고, 특정 곱셈을 위해 교과서 방법으로 풀백하여 메모리 점유율을 낮추는 방법을 제안하였다. Dilithium의 키/서명 생성 알고리즘에서 공개 행렬은 보안 레벨에 따른 다항식을 필요로 하기 때문에 임베디드 기기에서 stack에 선언해서 사용하는 것은 비효율적이다. 따라서 최적화를 위해 PQM4와 동일하게 전체 공개행렬을 선언하지 않고 필요할 때 A와 y의 요소만 즉석에서 생성하는 방법을 상기 논문에서는 채용하였다.



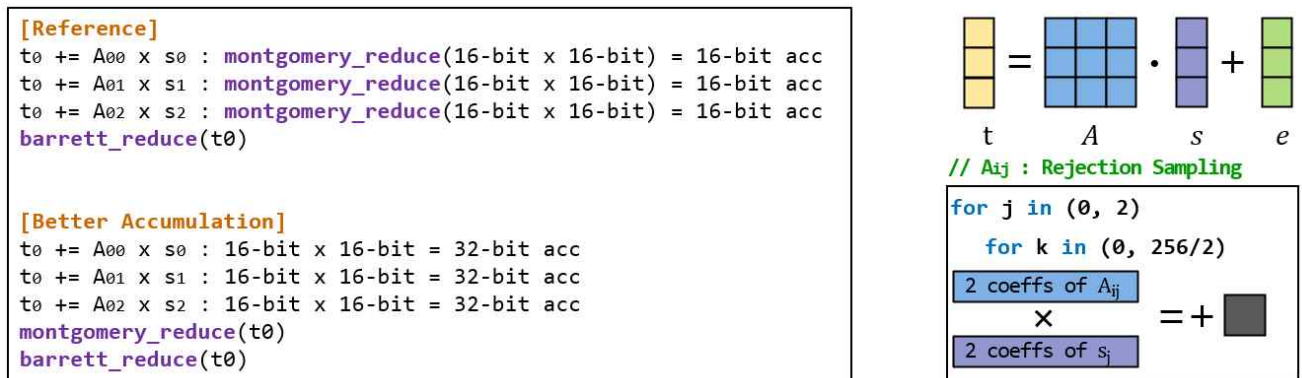
[그림 8] Dilithium의 서명 스트리밍 구현 도식도

Kyber reference C code	pqm4 Kyber code
<pre> void indcpa_keypair_derand (uint8_t pk[KYBER_INDCPA_PUBLICKEYBYTES], uint8_t sk[KYBER_INDCPA_SECRETKEYBYTES], const uint8_t coins[KYBER_SYMBYTES]) { unsigned int i; uint8_t buf[2*KYBER_SYMBYTES]; const uint8_t *publicseed = buf; const uint8_t *noiseseed = buf+KYBER_SYMBYTES; uint8_t nonce = 0; polyvec a[KYBER_K], e, pkpv, skpv; memcpy(buf, coins, KYBER_SYMBYTES); buf[KYBER_SYMBYTES] = KYBER_K; hash_g(buf, buf, KYBER_SYMBYTES+1); gen_a(a, publicseed); for(i=0;i<KYBER_K;i++) poly_getnoise_eta1(&skpv.vec[i], noiseseed, nonce++); for(i=0;i<KYBER_K;i++) poly_getnoise_eta1(&e.vec[i], noiseseed, nonce++); polyvec_ntt(&skpv); polyvec_ntt(&e); for(i=0;i<KYBER_K;i++) { polyvec_basemul_acc_montgomery (&pkpv.vec[i], &a[i], &skpv); poly_tomont(&pkpv.vec[i]); } polyvec_add(&pkpv, &pkpv, &e); polyvec_reduce(&pkpv); pack_sk(sk, &skpv); pack_pk(pk, &pkpv, publicseed); } </pre>	<pre> void indcpa_keypair_derand (unsigned char *pk, unsigned char *sk, const unsigned char *coins) { polyvec skpv; poly pkp; unsigned char buf[2 * KYBER_SYMBYTES]; unsigned char *publicseed = buf; unsigned char *noiseseed = buf + KYBER_SYMBYTES; unsigned char nonce = 0; hash_g(buf, coins, KYBER_SYMBYTES); for (int i = 0; i < KYBER_K; i++) poly_getnoise(skpv.vec + i, noiseseed, nonce++); // 비밀 벡터에 대한 NTT 변환 polyvec_ntt(&skpv); for (int i = 0; i < KYBER_K; i++) { // 공개행렬 A의 계수 Streaming을 통한 점별 곱셈 수행 후 누적 matacc(&pkp, &skpv, i, publicseed, 0); // 누적된 행렬 곱셈 결과 다항식에 대한 iNTT 변환 poly_invntt(&pkp); // 에러(노이즈) 덧셈 후 NTT 변환 (Kyber는 공개키가 NTT 도메인에 존재) poly_addnoise(&pkp, noiseseed, nonce++); poly_ntt(&pkp); // 공개키 packing poly_tobytes(pk+i*KYBER_POLYBYTES, &pkp); } polyvec_tobytes(sk, &skpv); memcpy (pk + KYBER_POLYVECBYTES, publicseed, KYBER_SYMBYTES); } </pre>

[표 7] Kyber ref 코드 및 pqm4 Kyber streaming 코드 비교

제 3.4 절 Better Accumulation

- Module-LWE (이 절에서는 Kyber를 예시로 삼는다)의 핵심연산인 $A \cdot s + e$ 을 계산 결과 또한 R_{3329} 의 원소여야하기 때문에 $A \cdot s$ 연산 시 각 다항식의 계수 곱셈은 레퍼런스 구현의 경우 Montgomery 감산을 통해 16-bit로 감산 및 누적 후 마지막에 Barrett 감산을 수행한다. $As+e$ 에서 누적 매개변수를 32-bit 다항식을 사용하고 마지막 다항식 곱셈에서만 Montgomery 감산을 수행하면 각 계수별 곱셈 시 Montgomery 감산을 사용하지 않아 성능을 가속화 할 수 있다. Kyber의 q 는 12-bit이기 때문에 곱셈 시 최대 24-bit이고 k 번 누적 시 최대 29-bit 이므로 32-bit 자료형에 대해 오버플로가 발생하지 않는다.⁶⁾ 단 누적 다항식을 사용하기 위해서는 $32 \times 256/8$ bytes의 추가 스택을 사용해야 한다. 하기 그림은 Better Accumulation의 기본 아이디어를 보여준다.



[그림 9] Better Accumulation의 아이디어 (Kyber 예시)

- pqm4에서 현재 Kyber 및 Dilithium의 “speed” 버전 구현물에 해당 아이디어가 탑재되어 있으며, 국내 KpqC 알고리즘 중 Module 구조를 갖는 격자기반 암호체계에도 적용이 가능할 것으로 예상된다. 하기 표는 pqm4 Kyber의 코드에서 Better Accumulation이 적용된 파트의 일부분을 보여준다. `r_tmp` 배열은 다항식 하나의 크기로, 32-bit로 값을 누적하고 최종적으로 누적된 값에 대해서 barrett 감산을 적용한다. 처음에 값을 생성하는 api가 `matacc_cache32`이고, 이후에 값을 누적하는 api가 `matacc_opt32`이다.

pqm4 Kyber matacc.c (matacc_cache32)	pqm4 Kyber matacc.c (matacc_opt32)
<pre> void matacc_cache32(poly* r, const polyvec *b, polyvec *b_prime, unsigned char i, const unsigned char *seed, int transposed) { unsigned char buf[XOF_BLOCKBYTES+2]; xof_state state; int16_t c[4]; // 32-bit 다항식의 누적 공간 int32_t r_tmp[KYBER_N]; int j = 0; // 16-32 if (transposed) xof_absorb(&state, seed, i, j); else xof_absorb(&state, seed, j, i); xof_squeezeblocks(buf, 1, &state); // 점별 곱셈 16-bit x 16-bit를 32-bit로 누적 matacc_asm_cache_16_32(r_tmp, b->vec[j].coeffs, c, buf, zetas, &state, b_prime->vec[j].coeffs); // 32-32 KYBER_K - 2 times for(j=1;j<KYBER_K - 1;j++) { if (transposed) xof_absorb(&state, seed, i, j); else xof_absorb(&state, seed, j, i); xof_squeezeblocks(buf, 1, &state); // 점별 곱셈 16-bit x 16-bit를 32-bit로 누적 matacc_asm_cache_32_32(r_tmp, b->vec[j].coeffs, c, buf, zetas, &state, b_prime->vec[j].coeffs); } // 32-16 if (transposed) xof_absorb(&state, seed, i, j); else xof_absorb(&state, seed, j, i); xof_squeezeblocks(buf, 1, &state); // 점별 곱셈 16-bit x 16-bit를 32-bit로 누적 후 // 마지막에 16-bit로 감산 수행 matacc_asm_cache_32_16(r->coeffs, b->vec[j].coeffs, c, buf, zetas, &state, b_prime->vec[j].coeffs, r_tmp); } </pre>	<pre> void matacc_opt32(poly* r, const polyvec *b, const polyvec *b_prime, unsigned char i, const unsigned char *seed, int transposed) { unsigned char buf[XOF_BLOCKBYTES+2]; xof_state state; int16_t c[4]; // 32-bit 다항식의 누적 공간 int32_t r_tmp[KYBER_N]; int j = 0; // 16-32 if (transposed) xof_absorb(&state, seed, i, j); else xof_absorb(&state, seed, j, i); xof_squeezeblocks(buf, 1, &state); // 점별 곱셈 16-bit x 16-bit를 32-bit로 누적 matacc_asm_opt_16_32(r_tmp, b->vec[j].coeffs, c, buf, &state, b_prime->vec[j].coeffs); // 32-32 KYBER_K - 2 times for(j=1;j<KYBER_K - 1;j++) { if (transposed) xof_absorb(&state, seed, i, j); else xof_absorb(&state, seed, j, i); xof_squeezeblocks(buf, 1, &state); // 점별 곱셈 16-bit x 16-bit를 32-bit로 누적 matacc_asm_opt_32_32(r_tmp, b->vec[j].coeffs, c, buf, &state, b_prime->vec[j].coeffs); } // 32-16 if (transposed) xof_absorb(&state, seed, i, j); else xof_absorb(&state, seed, j, i); xof_squeezeblocks(buf, 1, &state); // 점별 곱셈 16-bit x 16-bit를 32-bit로 누적 후 // 마지막에 16-bit로 감산 수행 matacc_asm_opt_32_16(r->coeffs, b->vec[j].coeffs, c, buf, &state, b_prime->vec[j].coeffs, r_tmp); } </pre>

[표 8] pqm4 Kyber 코드의 Better accumulation 적용 파트
(좌, 우 코드 모두에 적용되어있음)

제 3.5 절 Asymmetric Multiplication

- Module-LWE (이 절에서는 Kyber를 예시로 삼는다)의 핵심연산인 $A \cdot s + e$ 을 계산할 때 Crystals-Kyber의 비밀벡터 s 의 비밀벡터는 고정되고 반복되어 사용된다. Kyber는 Incomplete NTT를 사용하므로 NTT 도메인 위의 A, s 의 곱셈은 2개 계수끼리의 school-book 기반의 곱셈을 수행한다. Crystals-Kyber에서 다항식 곱셈을 위해 Incomplete NTT 변환을 수행하면 링의 분할은 $X^{256} + 1 = \prod_{i=0}^{127} (X^2 - \zeta^{2i+1})$ 과 같다. 처음에 $A_{0j} \cdot s_i$ ($i, j \in [0, k)$) 계산 시 s_i 의 원소와 ζ^{2i+1} 의 곱셈에 대한 몽고메리 감산의 결과를 저장한다면 다른 행에 대한 곱셈 시 일부 계산을 생략할 수 있다⁷⁾. 단, $k \times 16 \times 128/8$ bytes의 추가 스택을 사용해야하는 단점이 있다. 예를 들어 $X^2 - \zeta^{2i+1}$ 상에서의 다항식 곱셈 시 $X^2 = \zeta^{2i+1}$ 이고, $a_0 + a_1x$ 와 $b_0 + b_1x$ 의 곱셈의 결과는 $(a_1b_1 + a_0b_0\zeta^{2i+1}) + (a_1b_0 + a_0b_1)x$ 이다. 행렬곱셈에서 비밀 벡터는 반복적으로 사용되므로 비밀 벡터에 대응되는 $b_0\zeta^{2i+1}$ 을 저장하면 각 계수별 몽고메리 곱셈을 생략할 수 있다. 하기 그림은 Asymmetric Multiplication 구현 방법론의 아이디어를 보여준다.

```
void basemul
(int16_t r[2], int16_t a[2], int16_t s[2], int16_t zeta)
{
    r[0] = montgomery_reduce(s[1] * zeta); // 생략가능
    r[0] = montgomery_reduce(r[0] * a[1]);
    r[0] += montgomery_reduce(a[0] * s[0]);
    r[1] = montgomery_reduce(a[0] * s[1]);
    r[1] += montgomery_reduce(a[1] * s[0]);
}
```

[그림 10] Asymmetric Multiplication의 아이디어 (Kyber 예시)

- pqm4에서 현재 Kyber의 "speed" 버전 구현물에 해당 아이디어가 탑재되어 있으며, 국내 KpqC 알고리즘 중 Incomplete NTT를 사용하는 격자기반 암호체계에 적용이 가능할 것으로 예상된다. 하기 표는 Asymmetric 구현 방법론이 적용된 pqm4의 Kyber 코드의 예시를 보여준다. skpv_prime가 s_i 의 원소와 ζ^{2i+1} 의 곱셈에 대한 몽고메리 감산의 결과를 저장한 변수이다. skpv_prime은 벡터이기 때문에, 보안레벨 별 저장해야하는 stack 크기가 다르다.

pqm4 Kyber stack version code	pqm4 Kyber speed code (Asymmetric)
<pre> void indcpa_keypair_derand (unsigned char *pk, unsigned char *sk, const unsigned char *coins) { polyvec skpv; poly pkp; unsigned char buf[2 * KYBER_SYMBYTES]; unsigned char *publicseed = buf; unsigned char *noiseseed = buf + KYBER_SYMBYTES; int i; unsigned char nonce = 0; hash_g(buf, coins, KYBER_SYMBYTES); for (i = 0; i < KYBER_K; i++) poly_getnoise(skpv.vec + i, noiseseed, nonce++); polyvec_ntt(&skpv); for (i = 0; i < KYBER_K; i++) { matacc(&pkp, &skpv, i, publicseed, 0); poly_invntt(&pkp); poly_addnoise(&pkp, noiseseed, nonce++); poly_ntt(&pkp); poly_tobytes(pk+i*KYBER_POLYBYTES, &pkp); } polyvec_tobytes(sk, &skpv); memcpy (pk + KYBER_POLYVECBYTES, publicseed, KYBER_SYMBYTES); } </pre>	<pre> void indcpa_keypair_derand(unsigned char *pk, unsigned char *sk, const unsigned char *coins){ polyvec skpv, skpv_prime; poly pkp; unsigned char buf[2 * KYBER_SYMBYTES]; unsigned char *publicseed = buf; unsigned char *noiseseed = buf + KYBER_SYMBYTES; int i; unsigned char nonce = 0; hash_g(buf, coins, KYBER_SYMBYTES); for (i = 0; i < KYBER_K; i++) poly_getnoise(skpv.vec + i, noiseseed, nonce++); polyvec_ntt(&skpv); // i = 0 // 점별 곱셈에서 ζ^{2i+1}에 대한 부분 저장 // skpv_prime에 저장됨 matacc_cache32(&pkp, &skpv, &skpv_prime, 0, publicseed, 0); poly_invntt(&pkp); poly_addnoise(&pkp, noiseseed, nonce++); poly_ntt(&pkp); poly_tobytes(pk, &pkp); for (i = 1; i < KYBER_K; i++) { // 점별 곱셈에서 저장된 ζ^{2i+1} 부분 활용 matacc_opt32(&pkp, &skpv, &skpv_prime, i, publicseed, 0); poly_invntt(&pkp); poly_addnoise(&pkp, noiseseed, nonce++); poly_ntt(&pkp); poly_tobytes(pk+i*KYBER_POLYBYTES, &pkp); } polyvec_tobytes(sk, &skpv); memcpy(pk + KYBER_POLYVECBYTES, publicseed, KYBER_SYMBYTES); // Pack the public seed in the public key } </pre>

[표 9] pqm4 Kyber stack 및 speed 코드 비교 (Asymmetric Multiplication)

제 3.6 절 Merging Strategy

- NTT/iNTT 어셈블리 구현 시 메모리 접근 비용을 줄일 수 있는 Merging 전략이 존재한다. Merging 전략은 NTT/iNTT의 매 Layer마다 메모리 load, store를 하는 것이 아닌, 한 번의 load로 여러 레이어를 계산하고 store 한다. 하기는 Kyber를 기준으로 초기 3 Layer Merging에 대한 예시이다.

Kyber Refrence NTT code (3 Layer)	Kyber NTT Merging 3 Layer code
<pre> // 3 Layer에 대한 NTT 계산 void ntt (int16_t r[256]) { unsigned int len, start, j, k; int16_t t, zeta; k = 1; // 0 Layer : len = 128 // 1 Layer : len = 64 // 2 Layer : len = 32 // 매 Layer 계산 시 메모리 접근이 발생 for (len = 128; len >= 32; len >>= 1) { for (start = 0; start < 256; start = j + len) { zeta = zetas[k++]; for (j = start; j < start + len; j++) { t = fqmul(zeta, r[j + len]); r[j + len] = r[j] - t; r[j] = r[j] + t; } } } } </pre>	<pre> void butterfly(int16_t *a, int16_t *b, int16_t zeta) { int16_t t = fqmul(*b, zeta); *b = *a - t; *a = *a + t; } // 3 Layer merging 기법이 적용된 NTT 코드 void merge3_ntt(int16_t r[256]) { int16_t __m_zetas[8] = {0}; int16_t v[8] = { 0 }; // Twiddle Factor Load for (int i = 0; i < 8; i++) __m_zetas[i] = m_zetas[i]; for (int i = 0; i < 32; i++) { // 8개의 다항식 계수 Load for (int j = 0; j < 8; j++) v[j] = r[32 * j + i]; // Load된 8개 계수에 대한 3 Layer butterfly를 수행 butterfly(&v[0], &v[4], __m_zetas[1]); butterfly(&v[1], &v[5], __m_zetas[1]); butterfly(&v[2], &v[6], __m_zetas[1]); butterfly(&v[3], &v[7], __m_zetas[1]); butterfly(&v[0], &v[2], __m_zetas[2]); butterfly(&v[1], &v[3], __m_zetas[2]); butterfly(&v[4], &v[6], __m_zetas[3]); butterfly(&v[5], &v[7], __m_zetas[3]); butterfly(&v[0], &v[1], __m_zetas[4]); butterfly(&v[2], &v[3], __m_zetas[5]); butterfly(&v[4], &v[5], __m_zetas[6]); butterfly(&v[6], &v[7], __m_zetas[7]); // 8개의 다항식 계수 Store for (int j = 0; j < 8; j++) r[32 * j + i] = v[j]; } } </pre>

[표 10] Kyber의 Reference 코드의 3 Layer NTT 계산 코드 및 3 Layer Merging 기법이 적용된 NTT 코드

- 하기는 pqm4에 적용된 Kyber의 3 Layer Merging CT Butterfly이의 코드 분석 내용이다.

pqm4 Kyber speed version code
<p>// 32-bit 레지스터 2개(a0, a1)에 저장된 총 4개의 계수에 대한 Butterfly, 즉 doublebutterfly는 연속된 두개의 계수 쌍에 대한 Butterfly를 의미</p> <pre>.macro doublebutterfly_plant a0, a1, twiddle, tmp, q, qa smulwb Wtmp, Wtwiddle, Wa1 smulwt Wa1, Wtwiddle, Wa1 smlabt Wtmp, Wtmp, Wq, Wqa smlabt Wa1, Wa1, Wq, Wqa pkhtb Wtmp, Wa1, Wtmp, asr#16 usub16 Wa1, Wa0, Wtmp uadd16 Wa0, Wa0, Wtmp .endm</pre> <p>// 4개의 레지스터 즉, 총 8개의 계수에 대한 Butterfly</p> <pre>.macro two_doublebutterfly_plant a0, a1, a2, a3, twiddle0, twiddle1, tmp, q, qa doublebutterfly_plant Wa0, Wa1, Wtwiddle0, Wtmp, Wq, Wqa doublebutterfly_plant Wa2, Wa3, Wtwiddle1, Wtmp, Wq, Wqa .endm</pre> <p>// 3 Layer Merging을 위해서는 2^3개의 레지스터가 필요함 (c0, c1, ~, c7)</p> <pre>.macro _3_layer_double_CT_16_plant c0, c1, c2, c3, c4, c5, c6, c7, twiddle1, twiddle2, twiddle_ptr, q, qa, tmp // 첫번째 레이어로 CT{ (c0, c1, c2, c3), (c4, c5, c6, c7), t1 }에 대한 계산을 수행. twiddle factor(t1)은 한개 필요 // 따라서, 세부적으로 동일한 twiddle에 대해 CT(c0,c4, t1), CT(c1,c5, t1), CT(c2,c6, t1), CT(c3,c7, t1)를 계산 ldr.w Wtwiddle1, [Wtwiddle_ptr], #4 two_doublebutterfly_plant Wc0, Wc4, Wc1, Wc5, Wtwiddle1, Wtwiddle1, Wtmp, Wq, Wqa two_doublebutterfly_plant Wc2, Wc6, Wc3, Wc7, Wtwiddle1, Wtwiddle1, Wtmp, Wq, Wqa ----- // 두번째 레이어로 twiddle factor는 두개가 필요 (t2, t3). CT{ (c0, c1), (c2, c3), t2} 및 CT{ (c4, c5), (c6, c7), t3},에 대한 계산을 수행. // 따라서, 세부적으로 두개의 twiddle에 대해 CT(c0, c2, t2), CT(c1, c3, t2), CT(c4, c6, t3), CT(c5, c7, t3)를 계산 ldrd Wtwiddle1, Wtwiddle2, [Wtwiddle_ptr], #8 two_doublebutterfly_plant Wc0, Wc2, Wc1, Wc3, Wtwiddle1, Wtwiddle1, Wtmp, Wq, Wqa two_doublebutterfly_plant Wc4, Wc6, Wc5, Wc7, Wtwiddle2, Wtwiddle2, Wtmp, Wq, Wqa ----- // 두번째 레이어로 twiddle factor는 4개가 필요 (t4, t5, t6, t7). // 4개의 twiddle에 대해 CT(c0, c1, t4), CT(c2, c3, t5), CT(c4, c5, t6), CT(c6, c7, t7)를 계산 ldrd Wtwiddle1, Wtwiddle2, [Wtwiddle_ptr], #8 two_doublebutterfly_plant Wc0, Wc1, Wc2, Wc3, Wtwiddle1, Wtwiddle2, Wtmp, Wq, Wqa ldrd Wtwiddle1, Wtwiddle2, [Wtwiddle_ptr], #8 two_doublebutterfly_plant Wc4, Wc5, Wc6, Wc7, Wtwiddle1, Wtwiddle2, Wtmp, Wq, Wqa .endm</pre> <p>// Merging 전략을 사용하지 않으면, 파란색 구분선에서 다항식 계수를 추가로 메모리에 저장해야함</p>

[표 11] pqm4 Kyber의 NTT 코드

(https://github.com/mupq/pqm4/blob/master/crypto_kem/kyber768/m4fspeed/fastntt.S)

제 3.7 절 Binary Field Multiplication

□ Binary Field 상에서의 연산

- 앞서 설명한 Kyber와 같이 Prime Field 상에서 연산이 이루어지는 알고리즘과 달리 Binary Field 상에서의 연산을 통해 설계한 PQC 알고리즘 역시 존재한다. 현재 NIST PQC Round 4에서 심사가 진행 중인 알고리즘 3종(Classic McEliece, HQC, BIKE) 모두 코드 기반 암호가 이에 해당한다. 이런 Binary Field 상에서 다항식의 계수는 0과 1뿐이므로 덧셈과 뺄셈은 XOR로 이루어지며 곱셈의 경우 이런 덧셈의 성질로 인해 carry가 발생하지 않는 carry-less 곱셈이 수행된다.
- 코드 기반 암호뿐만 아니라 Binary Field 상에서 연산이 구성되는 대부분의 PQC 역시 다항식 곱셈에서 주요 연산 부하가 발생하므로 여전히 다항식 곱셈이 가장 우선시 되는 최적화 지점이다. 따라서 본 절에서는 대표적인 Binary Field 곱셈 방법을 소개하고, Cortex-M4 환경에서 최적화하는 방법론을 설명한다.

□ Binary Field Multiplication Algorithm

- 기본적인 Carry-less 곱셈 방법은 아래 [그림 17]과 같이 한 Bit씩 검사하여 1인 경우 곱셈 결과를 더해주는 방식으로 구현 시 [그림 16]과 같이 조건문을 통해 비트 연산만으로 구현할 수 있다. 다만 이는 조건문에 따라 달라지는 연산에 의해 부채널 분석(Constant-Time)에 취약하다는 단점이 있다.

Ordinary multiplication	Carryless multiplication
$\begin{array}{r} 1011 \\ 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ +) 1011 \\ \hline 10001111 \end{array}$	$\begin{array}{r} 1011 \\ 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ xor) 1011 \\ \hline 1111111 \end{array}$

[그림 11] Ordinary multiplication과 Carry-less multiplication의 연산 차이

Algorithm 7 Multiplication in $GF(2^{128})$. Computes the value of $Z = X \cdot Y$, where X, Y and $Z \in GF(2^{128})$.

```

1:  $Z \leftarrow 0, V \leftarrow X$ 
2: for  $i = 0$  to  $127$  do
3:   if  $Y_i = 1$  then
4:      $Z \leftarrow Z \oplus V$ 
5:   end if
6:   if  $V_{127} = 0$  then
7:      $V \leftarrow \text{rightshift}(V)$ 
8:   else
9:      $V \leftarrow \text{rightshift}(V) \oplus R$ 
10:  end if
11: end for
12: return  $Z$ 

```

[표 12] Binary Field에서의 School Book Multiplication

- 위의 취약점을 보완하고자 같이 Bit-Level에서 조건문을 제거한 아래 [그림 17]와 같은 Bit Width(Word size) 단위로 연산하는 곱셈 방법을 사용하며, 각각의 PQC 알고리즘에서 사용한 다항식 ring의 irreducible polynomial을 통해 modular 연산까지 통합한 곱셈 방법을 사용한다. 해당 곱셈 방법을 채택한 알고리즘은 대표적으로 Classic McEliece, MCBits가 있으며 곱셈 방법은 동일하지만 각각의 알고리즘이 선택한 irreducible polynomial이 달라 modular 연산 과정만 달라짐을 [표 4]에서 확인할 수 있다.

Algorithm 8 Bit-Level Combinational Multiplication. A and B Are BW -Bit Digits, R Is $(2BW - 1)$ -Bit Long. $A[i], B[i]$ and $R[i]$ Indicate Single Bits

```

1: function COMBINATIONALMUL( $A, B$ ) ▷ Returns  $[R]$ 
2:   for  $i = 0$  to  $BW - 1$  do
3:     for  $j = 0$  to  $BW - 1$  do
4:        $R[i + j] = R[i + j] \oplus (A[i] \cdot B[j])$ 
5:     end for
6:   end for
7:   return  $R$ 
8: end function

```

[표 13] Bit-Level Combinational Multiplication

Classic McEliece	McBits
<pre> 24 gf gf_mul(gf in0, gf in1) 25 { 26 int i; 27 28 uint32_t tmp; 29 uint32_t t0; 30 uint32_t t1; 31 uint32_t t; 32 33 t0 = in0; 34 t1 = in1; 35 36 tmp = t0 * (t1 & 1); 37 38 for (i = 1; i < GFBITS; i++) 39 tmp ^= (t0 * (t1 & (1 << i))); 40 41 t = tmp & 0x7FC000; 42 tmp ^= t >> 9; 43 tmp ^= t >> 12; 44 45 t = tmp & 0x3000; 46 tmp ^= t >> 9; 47 tmp ^= t >> 12; 48 49 return tmp & ((1 << GFBITS)-1); 50 } </pre>	<pre> 7 gf gf_mul(gf in0, gf in1) 8 { 9 int i; 10 11 uint64_t tmp; 12 uint64_t t0; 13 uint64_t t1; 14 uint64_t t; 15 16 t0 = in0; 17 t1 = in1; 18 19 tmp = t0 * (t1 & 1); 20 21 for (i = 1; i < GFBITS; i++) 22 tmp ^= (t0 * (t1 & (1 << i))); 23 24 // 25 26 t = tmp & 0x1FF0000; 27 tmp ^= (t >> 9) ^ (t >> 10) ^ (t >> 12) ^ (t >> 13); 28 29 t = tmp & 0x000E000; 30 tmp ^= (t >> 9) ^ (t >> 10) ^ (t >> 12) ^ (t >> 13); 31 32 return tmp & GFMASK; 33 } </pre>

[표 14] PQC 3종에서 사용한 Binary Field Multiplication 구현 방법

- 앞서 설명한 모든 비트를 하나씩 검사하여 연산하는 Bit serial 방식과 달리 여러 비트를 한 번에 처리할 수 있도록 Table을 기반으로 연산하는 방식이 존재한다. 두 다항식에 대해 특정 단위의 크기인 window를 설정하여 작은 다항식의 배수를 미리 테이블에 저장하고, 이를 참조하여 연산하는 방식이다. 해당 곱셈 방법은 [그림 22]과 같이 3단계로 나눌 수 있다. 먼저 window(s) 크기에 맞는 테이블을 생성하는 Step 1과 해당 데이터를 통해 곱셈을 수행하는 Step 2 (w는 연산 word 크기), 마지막으로 고차(high part) 부분에 대한 보정(repair)과정인 Step 3로 이루어져 있다. 보정 과정이 필요한 이유는 우선 Step 1에서 테이블을 생성할 때 b의 하위 비트(window에 따라)에 해당하는 값들만 저장되어 h 파트가 고려되지 않을 수 있고, 기존 h 파트에 비트가 설정되어 있다면 곱셈의 결과가 해당 위치에 영향을 주게 되므로 마스킹을 통한 보정 과정이 필요하다.

```

mul1(ulong a, ulong b)
multiplies polynomials a and b. The result goes in l (low part) and h (high part).
    ulong u[2s] = { 0, b, 0, ... };                                /* Step 1 (tabulate) */
    for(int i = 2 ; i < 2s ; i += 2)
        u[i] = u[i >> 1] << 1; u[i + 1] = u[i] ^ b;
    ulong g = u[a & (2s - 1)], l = g, h = 0;                        /* Step 2 (multiply) */
    for(int i = s ; i < w ; i += s)
        g = u[a >> i & (2s - 1)]; l ^= g << i; h ^= g >> (w - i);
    ulong m = (2s - 2) × (1 + 2s + 22s + 23s + ...) mod 2w; /* Step 3 (repair) */
    for(int j = 1 ; j < s ; j++)
        a = (a << 1) & m;
        if (bit w - j of b is set) h ^= a;
    return l, h;

```

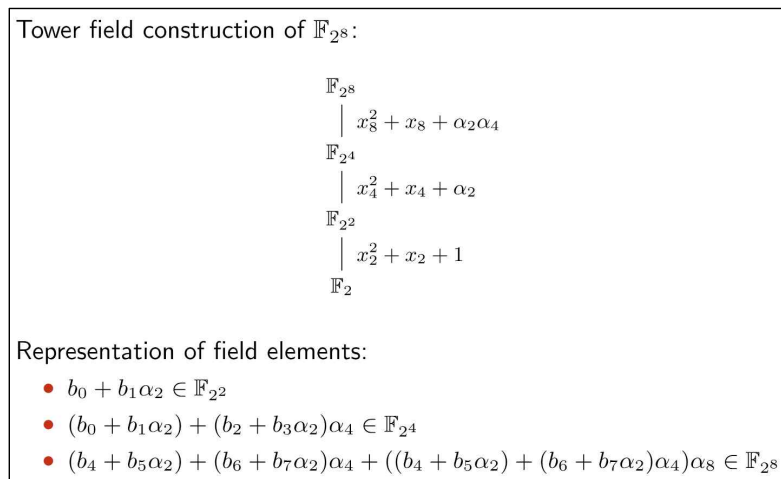
[그림 12] Window-Table 기반 Binary Field Multiplication

해당 구현은 오픈 Binary Field 곱셈 라이브러리인 NTL (Number Theory Library)와 gf2x(Galois Field 2 multiplication)에서 채택하였고, 이는 대표적으로 HQC, BIKE 알고리즘에서 채택되어 활용되고 있다.

HQC, BIKE (NTL 활용)	
(Case 1) w=32, s=3	(Case 2) w=64, s=4
<pre> 443 #define NTL_ALT1_BB_MUL_CODE0 \ 444 _ntl_ulong hi, lo, t;\ 445 _ntl_ulong A[8];\ 446 A[0] = 0;\ 447 A[1] = a;\ 448 A[2] = A[1] << 1;\ 449 A[3] = A[2] ^ A[1];\ 450 A[4] = A[2] << 1;\ 451 A[5] = A[4] ^ A[1];\ 452 A[6] = A[3] << 1;\ 453 A[7] = A[6] ^ A[1];\ 454 lo = A[b >> 7]; t = A[(b >> 3) & 7]; hi = t >> 29; lo ^= t << 3;\ 455 t = A[(b >> 6) & 7]; hi ^= t >> 26; lo ^= t << 6;\ 456 t = A[(b >> 9) & 7]; hi ^= t >> 23; lo ^= t << 9;\ 457 t = A[(b >> 12) & 7]; hi ^= t >> 20; lo ^= t << 12;\ 458 t = A[(b >> 15) & 7]; hi ^= t >> 17; lo ^= t << 15;\ 459 t = A[(b >> 18) & 7]; hi ^= t >> 14; lo ^= t << 18;\ 460 t = A[(b >> 21) & 7]; hi ^= t >> 11; lo ^= t << 21;\ 461 t = A[(b >> 24) & 7]; hi ^= t >> 8; lo ^= t << 24;\ 462 t = A[(b >> 27) & 7]; hi ^= t >> 5; lo ^= t << 27;\ 463 t = A[b >> 30]; hi ^= t >> 2; lo ^= t << 30;\ 464 hi ^= (((b & 0xb6db6db6UL) >> 1) & ~(a >> 31)))\ 465 ^ (((b & 0x24924924UL) >> 2) & ~(a >> 30) & 1UL));\ 466 c[0] = lo; c[1] = hi;\ </pre>	<pre> 1998 #define NTL_ALT1_BB_MUL_CODE0 \ 1999 uint64_t hi, lo, t;\ 2000 uint64_t A[16];\ 2001 A[0] = 0;\ 2002 A[1] = a;\ 2003 A[2] = A[1] << 1;\ 2004 A[3] = A[2] ^ A[1];\ 2005 A[4] = A[2] << 1;\ 2006 A[5] = A[4] ^ A[1];\ 2007 A[6] = A[3] << 1;\ 2008 A[7] = A[6] ^ A[1];\ 2009 A[8] = A[4] << 1;\ 2010 A[9] = A[8] ^ A[1];\ 2011 A[10] = A[5] << 1;\ 2012 A[11] = A[10] ^ A[1];\ 2013 A[12] = A[6] << 1;\ 2014 A[13] = A[12] ^ A[1];\ 2015 A[14] = A[7] << 1;\ 2016 A[15] = A[14] ^ A[1];\ 2017 lo = A[b & 15]; t = A[(b >> 4) & 15]; hi = t >> 60; lo ^= t << 4;\ 2018 t = A[(b >> 8) & 15]; hi ^= t >> 56; lo ^= t << 8;\ 2019 t = A[(b >> 12) & 15]; hi ^= t >> 52; lo ^= t << 12;\ 2020 t = A[(b >> 16) & 15]; hi ^= t >> 48; lo ^= t << 16;\ 2021 t = A[(b >> 20) & 15]; hi ^= t >> 44; lo ^= t << 20;\ 2022 t = A[(b >> 24) & 15]; hi ^= t >> 40; lo ^= t << 24;\ 2023 t = A[(b >> 28) & 15]; hi ^= t >> 36; lo ^= t << 28;\ 2024 t = A[(b >> 32) & 15]; hi ^= t >> 32; lo ^= t << 32;\ 2025 t = A[(b >> 36) & 15]; hi ^= t >> 28; lo ^= t << 36;\ 2026 t = A[(b >> 40) & 15]; hi ^= t >> 24; lo ^= t << 40;\ 2027 t = A[(b >> 44) & 15]; hi ^= t >> 20; lo ^= t << 44;\ 2028 t = A[(b >> 48) & 15]; hi ^= t >> 16; lo ^= t << 48;\ 2029 t = A[(b >> 52) & 15]; hi ^= t >> 12; lo ^= t << 52;\ 2030 t = A[(b >> 56) & 15]; hi ^= t >> 8; lo ^= t << 56;\ 2031 t = A[b >> 60]; hi ^= t >> 4; lo ^= t << 60;\ 2032 hi ^= (((b & 0xffffffffUL) >> 1) & ~(a >> 63)))\ 2033 ^ (((b & 0xffffffffUL) >> 2) & ~(a >> 62) & 1UL))\ 2034 ^ (((b & 0x8888888888888888UL) >> 3) & ~(a >> 61) & 1UL));\ 2035 c[0] = lo; c[1] = hi;\ </pre>

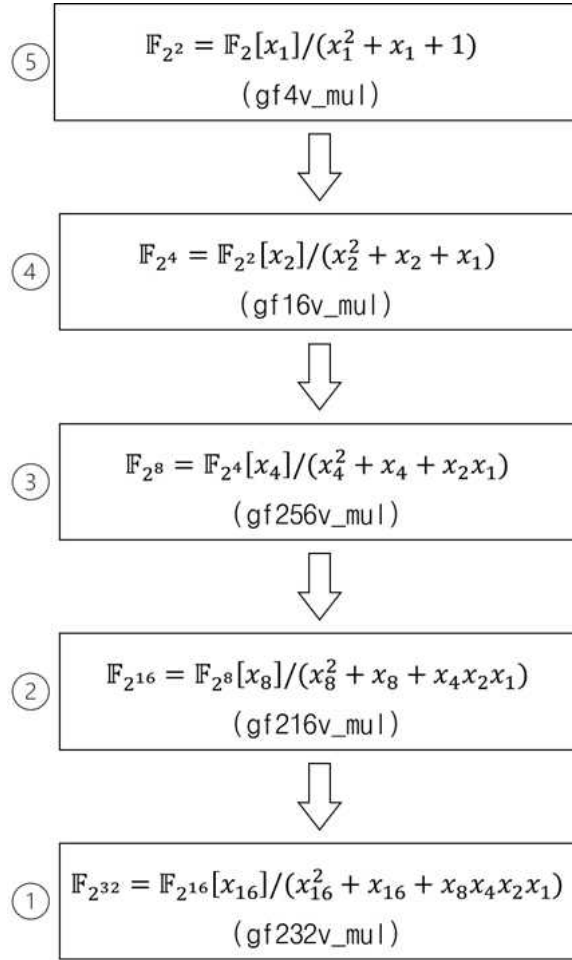
[표 15] PQC 3종에서 사용한 Binary Field Multiplication 구현 방법

- 앞서 설명한 두 방법처럼 곱셈 이후에 기약 다항식 modular를 통해 연산을 수행하는 방법과 달리 Binary Field의 Extension Field를 이용하는 곱셈 방법인 Tower of Extension Field Multiplication이 존재한다. 이는 F_{2^m} 상에서 $m = 2^n$ 꼴로 이루어졌을 때 주로 사용하는 방법으로, 작은 field에서의 연산 결과를 점차 확장하면서 더 큰 field의 연산 결과로 단계적으로 구성하는 방법이다.



[그림 13] F_{2^8} 에서의 Tower Field Arithmetic

[그림 25]와 같이 F_{2^8} 의 예시를 통해 확인하면, F_2 (0과 1로 이루어진 Binary Field)의 원소를 이용한 irreducible polynomial인 $x_2^2 + x_2 + 1$ 을 이용하여 상위 Extension Field인 F_{2^2} 를 구성한다. 여기서 해당 Field는 1과 x_2 의 선형 조합으로 표현된다. 따라서 F_{2^2} 의 임의의 원소는 $b_0 + b_1\alpha_2$ 로 표현할 수 있다. 이와 같은 논리로 상위 필드의 원소를 표현하면 결국 상대적으로 작은 필드의 원소를 계산하여 가장 큰 필드의 원소를 연산할 수 있다. 이런 Tower Field Arithmetic의 경우 F_{2^m} 에서의 역원/곱셈을 연산하거나 FFT 방법을 활용하여 고차 다항식의 곱셈을 수행할 때 주로 활용된다. Cortex-M4 PQC 구현물 중에서는 [표 29]과 같이 BIKE 최적화 연구에서 채택되었다.



[그림 14] 곱셈을 위한 Tower Field 연산 순서

[그림 10]과 같이 BIKE에서는 FAFFT (Frobenius Additive Fast Fourier Transform) forward FFT 마지막 과정에서 점별 곱셈 과정 중 위 $F_{2^{32}}$ 필드 상에서의 다항식 곱셈을 위해 Tower Field Arithmetic을 사용한다. Extension field의 실제 연산이 수행되는 방향은 화살표 방향이지만, 하위 field에서의 연산 결과를 활용하기 위해 함수 호출 순서는 왼쪽 번호 순서로 호출하게 된다.

BIKE : Tower of Extension Field Multiplication ($F_{2^{32}}$)	
$\begin{aligned} \mathbb{F}_{2^2} &= \mathbb{F}_2[x_1]/(x_1^2 + x_1 + 1), \\ \mathbb{F}_{2^4} &= \mathbb{F}_{2^2}[x_2]/(x_2^2 + x_2 + x_1), \\ \mathbb{F}_{2^8} &= \mathbb{F}_{2^4}[x_4]/(x_4^2 + x_4 + x_2x_1), \\ \mathbb{F}_{2^{16}} &= \mathbb{F}_{2^8}[x_8]/(x_8^2 + x_8 + x_4x_2x_1), \\ \mathbb{F}_{2^{32}} &= \mathbb{F}_{2^{16}}[x_{16}]/(x_{16}^2 + x_{16} + x_8x_4x_2x_1). \end{aligned}$	
<pre> 228 // gf232 := gf216[Z]/Z^2+Z+YXxy 229 static inline void gf232v_mul(sto_t *c, const sto_t *a, const sto_t *b) { 230 sto_t c1[16]; for(int i=0;i<16;i++) c1[i] = a[i]^a[16+i]; 231 sto_t c2[16]; for(int i=0;i<16;i++) c2[i] = b[i]^b[16+i]; 232 233 gf216v_mul(c , a , b); 234 gf216v_mul(c+16 , c1 , c2); 235 gf216v_mul(c2 , a+16 , b+16); 236 for(int i=0;i<16;i++) c[16+i] ^= c[i]; 237 238 gf216v_mul_0x8000(c1 , c2); 239 for(int i=0;i<16;i++) c[i] ^= c1[i]; 240 } </pre>	<pre> 202 // gf216 := gf256[Y]/Y^2+Y+Xxy 203 static inline void gf216v_mul(sto_t *c, const sto_t *a, const sto_t *b) { 204 sto_t c1[8]; for(int i=0;i<8;i++) c1[i] = a[i]^a[8+i]; 205 sto_t c2[8]; for(int i=0;i<8;i++) c2[i] = b[i]^b[8+i]; 206 207 gf256v_mul(c , a , b); 208 gf256v_mul(c+8 , c1 , c2); 209 gf256v_mul(c2 , a+8 , b+8); 210 for(int i=0;i<8;i++) c[8+i] ^= c[i]; 211 212 gf256v_mul_0x80(c1 , c2); 213 for(int i=0;i<8;i++) c[i] ^= c1[i]; 214 } </pre>
<pre> 133 // gf256 := gf16[X]/X^2+X+xy 134 static inline void gf256v_mul(sto_t *c, const sto_t *a, const sto_t *b) { 135 sto_t c1[4]; 136 sto_t c2[4]; 137 c1[0] = a[0]^a[4]; 138 c1[1] = a[1]^a[5]; 139 c1[2] = a[2]^a[6]; 140 c1[3] = a[3]^a[7]; 141 c2[0] = b[0]^b[4]; 142 c2[1] = b[1]^b[5]; 143 c2[2] = b[2]^b[6]; 144 c2[3] = b[3]^b[7]; 145 146 gf16v_mul(c , a , b); 147 gf16v_mul(c+4 , c1 , c2); 148 gf16v_mul(c2 , a+4 , b+4); 149 c[4] ^= c[0]; 150 c[5] ^= c[1]; 151 c[6] ^= c[2]; 152 c[7] ^= c[3]; 153 154 gf16v_mul_8(c1 , c2); 155 c[0] ^= c1[0]; 156 c[1] ^= c1[1]; 157 c[2] ^= c1[2]; 158 c[3] ^= c1[3]; 159 } </pre>	<pre> 72 // gf16 := gf4[y]/y^2+y+x 73 static inline void gf16v_mul(sto_t *c, const sto_t *a, const sto_t *b) { 74 sto_t c2[2]; 75 sto_t c1[2]; 76 c1[0] = a[0]^a[2]; 77 c1[1] = a[1]^a[3]; 78 c2[0] = b[0]^b[2]; 79 c2[1] = b[1]^b[3]; 80 gf4v_mul(c , a , b); 81 gf4v_mul(c+2 , c1 , c2); 82 gf4v_mul(c2 , a+2 , b+2); 83 c[2] ^= c[0]; 84 c[3] ^= c[1]; 85 86 gf4v_mul_2(c1 , c2); 87 c[0] ^= c1[0]; 88 c[1] ^= c1[1]; 89 } </pre>
<pre> 56 static inline void gf4v_mul(sto_t *c, const sto_t *a, const sto_t *b) { 57 c[0] = (a[0]&b[0])^(a[1]&b[1]); 58 c[1] = (a[0]&b[1])^(a[1]&b[0]); 59 } </pre>	<pre> 49 // gf4 := gf2[x]/x^2+x+1 50 static inline void gf4v_mul_2(sto_t *c, const sto_t *a) { 51 sto_t tmp = a[0]; 52 c[0] = a[1]; 53 c[1] = a[1]^tmp; 54 } </pre>

[표 16] BIKE Cortex-M4 최적화 방안에서 사용된 Tower of Extension Field Multiplication

□ Binary Field Multiplication Algorithm에 대한 Cortex-M4 Assembly 최적화

■ Cortex-M4 Assembly를 이용하여 32비트에 대한 Binary Multiplication을 최적화한 코드이다.

C language	Cortex-M4 ASM
<pre> 23 gfu32 gfu32_2_12_mul(gfu32 a, gfu32 b) 24 #endif 25 { 26 const uint32_t mask32 = 0x11111111; 27 uint32_t a0 = a&mask32; uint32_t a1 = (a>>1)&mask32; 28 uint32_t b0 = b&mask32; uint32_t b1 = (b>>1)&mask32; 29 30 union{ uint64_t v64; uint32_t v32[2]; } t0, t1; 31 uint32_t t2; 32 33 // multiplication 34 t0.v64 = ((uint64_t)a0) * ((uint64_t)b0); 35 t2 = a1 * b1; 36 37 t1.v32[0] = t0.v32[1]; 38 t1.v32[1] = t2; 39 40 t1.v64 += ((uint64_t)a1) * ((uint64_t)b0); 41 t1.v64 &= 0x1111111111111111ULL; 42 t1.v64 += ((uint64_t)a0) * ((uint64_t)b1); 43 44 t0.v32[1] = t1.v32[0]; 45 t2 = t1.v32[1]; 46 47 // reduction 48 const uint32_t rd_x12 = 0x1001; // x^12 = x^3 + 1 49 const uint32_t rd_x16 = rd_x12<<16; // x^16 = x^7 + x^4 50 t0.v64 += ((uint64_t)(t2&mask32)) * ((uint64_t)(rd_x16)); 51 t0.v64 &= 0x1111111111111111ULL; 52 t0.v64 += (((uint64_t)rd_x12) * ((uint64_t)(t0.v32[1]>>16))); 53 54 // output 55 t0.v32[0] &= mask32; // # 6 56 t0.v32[1] &= (mask32>>16); // # 7 57 58 return t0.v32[0]^(t0.v32[1]<<1); 59 } </pre>	<pre> 23 gfu32_2_12_mul: 24 @ args = 0, pretend = 0, frame = 0 25 @ frame_needed = 0, uses_anonymous_args = 0 26 push {r4, r5, lr} 27 @ unpack inputs 28 mov ip, #286331153 29 and r2, ip, r0, lsr #1 30 and r3, ip, r1, lsr #1 31 and r0, ip, r0 32 and r1, ip, r1 33 @ mult start 34 umull lr, r4, r2, r3 35 umull r4, r5, r0, r1 36 umlal r5, lr, r0, r3 37 and r5, ip 38 and lr, ip 39 umlal r5, lr, r1, r2 40 @ reduce 41 movw r2, #4097 42 and lr, ip 43 lsl r1, r2, #16 44 umlal r4, r5, lr, r1 45 and r5, ip 46 and r4, ip 47 lsr r3, r5, #16 48 umlal r4, r5, r2, r3 49 @ output 50 and r0, r4, ip 51 and r1, r5, ip, lsr #16 52 eor r0, r0, r1, lsl #1 53 54 pop {r4, r5, pc} </pre>

[표 17] Classic McEliece Cortex-M4 구현

■ Cortex-M4 Assembly를 이용하여 Tower of Extension Field Multiplication을 최적화한 코드이다.

C language	
<pre> 202 // gf216 := gf256[Y]/Y^2+Y+Xxy 203 static inline void gf216v_mul(sto_t *c, const sto_t *a, const sto_t *b) { 204 sto_t c1[8]; for(int i=0;i<8;i++) c1[i] = a[i]^a[8+i]; 205 sto_t c2[8]; for(int i=0;i<8;i++) c2[i] = b[i]^b[8+i]; 206 207 gf256v_mul(c , a , b); 208 gf256v_mul(c+8 , c1 , c2); 209 gf256v_mul(c2 , a+8 , b+8); 210 for(int i=0;i<8;i++) c[8+i] ^= c[i]; 211 212 gf256v_mul_0x80(c1 , c2); 213 for(int i=0;i<8;i++) c[i] ^= c1[i]; 214 } </pre>	<pre> 72 // gf16 := gf4[y]/y^2+y+x 73 static inline void gf16v_mul(sto_t *c, const sto_t *a, const sto_t *b) { 74 sto_t c2[2]; 75 sto_t c1[2]; 76 c1[0] = a[0]^a[2]; 77 c1[1] = a[1]^a[3]; 78 c2[0] = b[0]^b[2]; 79 c2[1] = b[1]^b[3]; 80 gf4v_mul(c , a , b); 81 gf4v_mul(c+2 , c1 , c2); 82 gf4v_mul(c2 , a+2 , b+2); 83 c[2] ^= c[0]; 84 c[3] ^= c[1]; 85 86 gf4v_mul_2(c1 , c2); 87 c[0] ^= c1[0]; 88 c[1] ^= c1[1]; 89 } </pre>
Cortex-M4 ASM	
<pre> 671 .macro m_gf216v_mul 672 sub sp, #32 //c1 673 mov r11, sp 674 sub sp, #32 //c2 675 mov r12, sp 676 vmov s10, r0 677 vmov s11, r1 678 vmov s12, r2 679 vmov s13, r11 680 vmov s14, r12 681 ldm r1!, {r3-r10} //a[8] 682 ldm r1!, {r0, r11, r12, r14} 683 m_eor4 r3, r4, r5, r6, r0, r11, r12, r14 684 ldm r1!, {r0, r11, r12, r14} 685 m_eor4 r7, r8, r9, r10, r0, r11, r12, r14 686 vmov r0, s13 //ptr_c1 687 stm r0, {r3-r10} //c1[8] 688 689 ldm r2!, {r3-r10} //b[8] 690 ldm r2!, {r0, r11, r12, r14} 691 m_eor4 r3, r4, r5, r6, r0, r11, r12, r14 692 ldm r2!, {r0, r11, r12, r14} 693 m_eor4 r7, r8, r9, r10, r0, r11, r12, r14 694 vmov r0, s14 //ptr_c2 695 stm r0, {r3-r10} //c2[8] 696 697 vmov r0, s10 //ptr_c 698 vmov r1, s11 //ptr_a 699 vmov r2, s12 //ptr_b 700 m_gf256v_mul 701 add r0, #32 //ptr_c+8 702 vmov r1, s13 //ptr_c1 703 vmov r2, s14 //ptr_c2 704 m_gf256v_mul 705 vmov r1, s10 //ptr_c 706 mov r2, r0 //ptr_c+8 707 ldm r2, {r3-r10} //c+8[8] 708 ldm r1!, {r0, r11, r12, r14} //c0-c3 709 m_eor4 r3, r4, r5, r6, r0, r11, r12, r14 710 ldm r1!, {r0, r11, r12, r14} //c4-c7 711 m_eor4 r7, r8, r9, r10, r0, r11, r12, r14 712 stm r2, {r3-r10} //c+8[8] 713 714 vmov r0, s14 //ptr_c2 715 vmov r1, s11 716 add r1, #32 //ptr_a+8 717 vmov r2, s12 718 add r2, #32 //ptr_b+8 719 m_gf256v_mul 720 vmov r1, s14 //ptr_c2 721 m_gf256v_mul_0x80 load r1, r3 r4, r5, r6, r7, r8, r9, r10 722 vmov r2, s10 //ptr_c 723 ldm r2, {r0, r11, r12, r14} //c0-c3 724 m_eor4 r0, r11, r12, r14, r3, r4, r5, r6 725 stm r2, {r0, r11, r12, r14} //c0-c3 726 add r2, #16 727 ldm r2, {r0, r11, r12, r14} //c4-c7 728 m_eor4 r0, r11, r12, r14, r7, r8, r9, r10 729 stm r2, {r0, r11, r12, r14} //c4-c7 730 add sp, #64 731 .endm </pre>	<pre> 179 //void gf16v_mul_asm(sto_t *c, const sto_t *a, const sto_t *b) 180 // does not preserve pointers, uses all registers 181 .global gf16v_mul_asm 182 gf16v_mul_asm: 183 push {r4-r11, lr} 184 ptr_c .req r0 185 ptr_a .req r1 186 c_0 .req r1 187 ptr_b .req r2 188 c_1 .req r2 189 a_0 .req r3 190 a_1 .req r4 191 a_2 .req r5 192 a_3 .req r6 193 b_0 .req r7 194 b_1 .req r8 195 b_2 .req r9 196 b_3 .req r10 197 c_2 .req r11 198 c_3 .req r12 199 buf0 .req r14 200 //ldm ptr_a, {a_0-a_3} 201 //ldm ptr_b, {b_0-b_3} 202 ldr a_0, [ptr_a, #0] 203 ldr a_1, [ptr_a, #4] 204 ldr a_2, [ptr_a, #8] 205 ldr a_3, [ptr_a, #12] 206 ldr b_0, [ptr_b, #0] 207 ldr b_1, [ptr_b, #4] 208 ldr b_2, [ptr_b, #8] 209 ldr b_3, [ptr_b, #12] 210 m_gf16v_mul c_0, c_1, c_2, c_3, a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3, buf0 211 //stm ptr_c, {c_0, c_1, c_2, c_3} 212 str c_0, [ptr_c, #0] 213 str c_1, [ptr_c, #4] 214 str c_2, [ptr_c, #8] 215 str c_3, [ptr_c, #12] 216 .unreq ptr_c 217 .unreq ptr_a 218 .unreq c_0 219 .unreq ptr_b 220 .unreq c_1 221 .unreq a_0 222 .unreq a_1 223 .unreq a_2 224 .unreq a_3 225 .unreq b_0 226 .unreq b_1 227 .unreq b_2 228 .unreq b_3 229 .unreq c_2 230 .unreq c_3 231 .unreq buf0 232 pop {r4-r11, pc} </pre>

C language	
<pre> 56 static inline void gf4v_mul(sto_t *c, const sto_t *a , const sto_t *b) { 57 c[0] = (a[0]&b[0])^(a[1]&b[1]); 58 c[1] = (a[0]&b[1])^(a[1]&(b[0]^b[1])); 59 } </pre>	<pre> 49 // gf4 := gf2[x]/x^2+x+1 50 static inline void gf4v_mul_2(sto_t *c, const sto_t *a) { 51 sto_t tmp = a[0]; 52 c[0] = a[1]; 53 c[1] = a[1]^tmp; 54 } </pre>
Cortex-M4 ASM	
<pre> 56 //void gf4v_mul_asm(sto_t *c, const sto_t *a , const sto_t *b) 57 .global gf4v_mul_asm 58 gf4v_mul_asm: 59 push {r4-r5} 60 ptr_c .req r0 61 ptr_a .req r1 62 buf0 .req r1 63 ptr_b .req r2 64 buf1 .req r2 65 a_0 .req r3 66 a_1 .req r4 67 buf2 .req r4 68 b_0 .req r5 69 b_1 .req r12 70 //ldm ptr_a, {a_0, a_1} 71 //ldm ptr_b, {b_0, b_1} 72 ldr a_0, [ptr_a, #0] 73 ldr a_1, [ptr_a, #4] 74 ldr b_0, [ptr_b, #0] 75 ldr b_1, [ptr_b, #4] 76 and buf0, a_0, b_0 77 and buf1, a_1, b_1 78 eor buf0, buf0, buf1 //c_0 79 eor buf1, b_0, b_1 80 and a_1, a_1, buf1 81 and buf1, a_0, b_1 82 eor buf1, buf1, buf2 //c_1 83 //stm ptr_c, {buf0, buf1} 84 str buf0, [ptr_c, #0] 85 str buf1, [ptr_c, #4] 86 .unreq a_1 87 .unreq ptr_a 88 .unreq ptr_b 89 .unreq ptr_c 90 .unreq buf0 91 .unreq buf1 92 .unreq a_0 93 .unreq buf2 94 .unreq b_0 95 .unreq b_1 96 pop {r4-r5} 97 bx lr </pre>	<pre> 45 //void gf4v_mul_2_asm(sto_t *c, const sto_t *a) 46 .global gf4v_mul_2_asm 47 gf4v_mul_2_asm: 48 ldr r3, [r1] //load a0 49 ldr r2, [r1, #4] //load a1 50 eor r3, r3, r2 //c1= a0 xor a1 51 //stm r0, {r2, r3} 52 str r2, [r0, #0] 53 str r3, [r0, #4] 54 bx lr </pre>

[표 18] BIKE Cortex-M4 구현

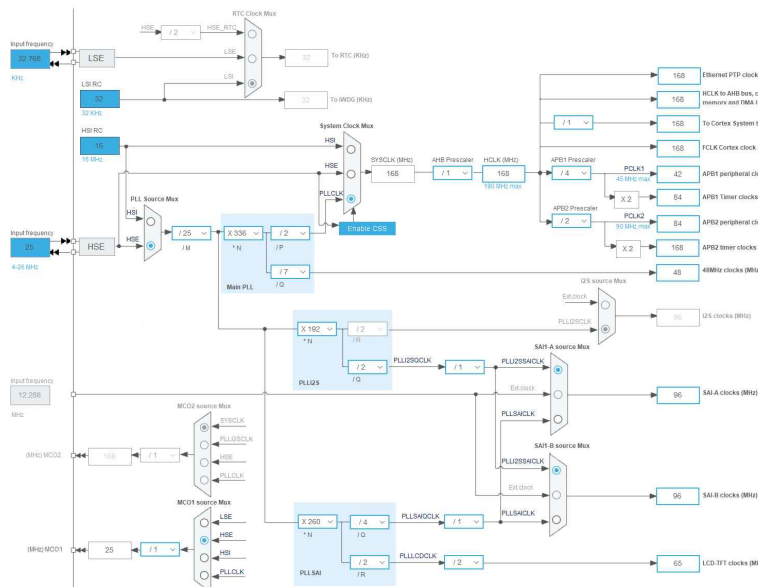
- Window-Table 기반 Binary Field Multiplication의 경우 Cortex-M4 환경에서 현재 최적화된 연구는 없다. 다만 해당 연산을 대해서는 연산 부하 감소를 위해 Barrel Shifter를 통해 Shift 연산을 최적화하는 것이 중요하다는 것을 알 수 있다.

제 4 장 Cortex-M4 구현 기술 Tip 및 가이드

제 4.1 절 주파수 및 Cycles

□ CubeIDE 동작 주파수 설정

- CubeIDE에는 각 프로젝트마다 보드의 동작 주파수와 함께 클럭 설정을 할 수 있는 기능을 지원한다. 해당 설정을 위해서는 프로젝트의 .ioc 파일을 연 뒤, 상단의 Clock Configuration으로 진입한다. 진입한 뒤에 아래와 같은 클럭 부분의 회로도가 나오게 된다.



제 4.2 절 malloc

□ 임베디드 환경에서의 동적 할당

- 임베디드 환경에서는 일반적으로 동적 할당을 사용하지 않고, 전부 정적인 배열을 스택에 저장하여 사용한다.
- 동적 할당을 정적 배열로 바꾸기 위해서는 먼저 해당 배열이 항상 고정된 값을 할당하는지, 아니면 할당할 때마다 크기가 바뀌는지 알아야 할 필요가 있다. 만약 항상 고정된 값이 할당된다면 해당 크기만큼 정적 배열로 바꾸면 된다. 그러나 할당할 때마다 할당되는 크기가 바뀐다면 할당되는 크기의 최댓값을 알아야 한다. 해당 최댓값으로 배열을 크기를 정적 할당하면 기존에 동적 할당한 크기보다는 메모리 사용량이 증가하게 되지만, 동적 할당을 지양하는 임베디드 환경에서는 필요한 과정이다.
- KpqC 알고리즘 중 SMAUG-T를 예시로 들어, 동적 할당을 어떻게 정적 할당으로 바꾸는지 설명한다.
- SMAUG-T의 경우, sppoly 구조체의 sx 변수를 포인터로 두어 동적할당을 수행한다. 해당 변수는 동적 할당 할 때 SMAUG-T1 기준 최대 97바이트를 할당하며, 호출될 때마다 할당하는 크기가 다르다. 따라서 sx 변수의 최대 할당량인 97바이트만큼 할당하도록 정적으로 수정하였으며, sx 변수에 동적 할당 하는 함수가 호출되는 부분을 삭제함으로써 정적 배열로 변환 가능하다. sx 변수의 변경된 모습과 동적 할당 함수가 삭제된 것은 아래 그림에서 확인할 수 있다.

<pre>typedef struct { uint8_t* sx; uint8_t neg_start; uint8_t cnt; } sppoly; // sparse poly</pre>	<pre>typedef struct { uint8_t sx[97]; uint8_t neg_start; uint8_t cnt; } sppoly; // sparse poly</pre>
---	--

[그림 17] 기존 sppoly 구조체(좌)와 정적 배열로 바꾼 sppoly 구조체(우)

```
for (unsigned long long i = 0; i < MODULE_RANK; ++i) {
    r[i].cnt = cnt_arr[i];
    r[i].sx = (uint8_t *)malloc(cnt_arr[i] * sizeof(uint8_t));
    r[i].neg_start = convToIdx(r[i].sx, r[i].cnt, res + (i * LWE_N), LWE_N);
}

for (unsigned long long i = 0; i < MODULE_RANK; ++i) {
    r[i].cnt = cnt_arr[i];
    r[i].neg_start = convToIdx(r[i].sx, r[i].cnt, res + (i * LWE_N), LWE_N);
}
```

[그림 18] 기존의 동적할당이 있는 코드(위)와 동적할당을 제거한 코드(아래)

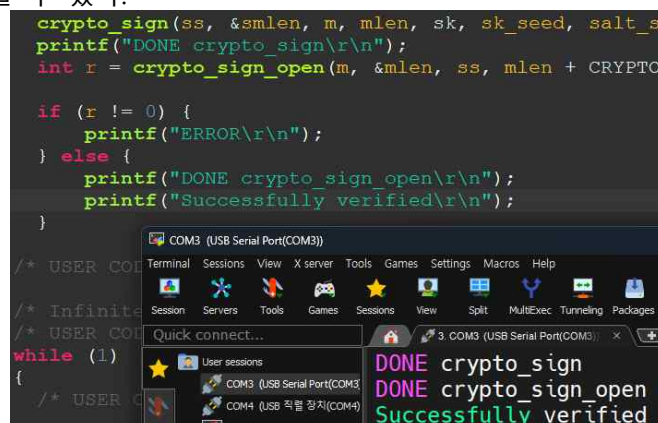
제 4.3 절 flash-memory Writing and Read

□ Flash-memory에 읽고 쓰기

- STM 보드에서는 플래시 메모리 영역에 데이터를 읽고 쓸 수 있다. 플래시 메모리는 메모리 영역의 일부로, 프로그램이 동작하기 전에 저장된 데이터이며, 프로그램이 실행되면서 저장된 데이터를 읽거나 해당 영역에 다시 쓰는 것이 가능하다.
- CubeIDE에서는 플래시 메모리에 데이터를 올리기 위해서 해당 변수에 다음과 같은 attribute를 부여한다.
 - 예를 들어, unsigned char로 선언된 a 배열을 플래시 메모리에 올리기 위해서는 다음과 같이 작성한다.: `__attribute__((section(".text"))) unsigned char a[] = { /* 데이터 */ };`

`__attribute__((section(".text"))) <type> <variable_name>`

- 이렇게 attribute가 부여된 변수는 CubeIDE에서 프로젝트를 빌드하고 실행할 때, 먼저 .text 영역에 데이터가 플래시 된다. .text 영역은 플래시 메모리의 일부로, 결과적으로 플래시 메모리에 데이터를 쓴 것이다.
- KpqC 알고리즘을 통한 예시를 설명한다. 대상 알고리즘은 MQ-Sign으로, STM32429I-EVAL1 보드의 SRAM 크기를 초과하는 키 쌍 크기를 가진다. 그러나 해당 보드의 플래시 메모리는 2MB이기 때문에 키 쌍을 플래시 메모리에 저장한다면 충분히 구동이 가능하다.
- MQ-Sign의 공개키와 개인키를 사전에 생성한다. 이는 PC에서 직접 생성하며, 키 생성 함수를 통해 생성된 공개키와 개인키를 출력하여 사전에 만들 수 있다. 이렇게 생성한 키 쌍을 key.h 헤더를 만들어 추가한 뒤, 위와 같은 attribute를 부여한다. 이렇게 하여 키 쌍을 플래시 메모리에 탑재되도록 한다.
- 그 뒤, 키 생성 함수를 제외한 서명 생성, 서명 검증 함수를 호출하여 서명의 유효성을 확인한다. 이를 위해서는 main 함수가 호출되는 .c 파일에 key.h 헤더를 include 해야 한다. 키 쌍을 저장하는 변수는 전역 변수로 사용할 수 있기 때문에 서명 생성 및 검증 과정에서 변수를 바로 사용할 수 있다.
- 마지막으로 프로젝트를 실행하면 키 쌍이 플래시 메모리에 탑재되며, 플래시 메모리에 탑재된 키를 읽어 서명을 생성 및 검증한다. 아래 그림과 같이 검증에 성공하며, 실제로 플래시 메모리에 저장된 키가 유효함을 확인할 수 있다.



The image shows a terminal window with a dark background. The top part displays C code for the MQ-Sign process, including functions for signing and verification. The bottom part shows the execution output, which includes the text 'DONE crypto_sign', 'DONE crypto_sign_open', and 'Successfully verified'. The terminal window has a title bar that reads 'COM3 (USB Serial Port(COM3))' and a menu bar with options like 'Terminal', 'Sessions', 'View', 'X server', 'Tools', 'Games', 'Settings', 'Macros', and 'Help'.

```
crypto_sign(ss, &smplen, m, mlen, sk, sk_seed, salt_seed);
printf("DONE crypto_sign\r\n");
int r = crypto_sign_open(m, &mlen, ss, mlen + CRYPTO_

if (r != 0) {
    printf("ERROR\r\n");
} else {
    printf("DONE crypto_sign_open\r\n");
    printf("Successfully verified\r\n");
}

/* USER CODE HERE */
/* Infinite loop */
while (1)
{
    /* USER CODE HERE */
}
```

[그림 19] MQ-Sign의 서명 생성 및 검증

Reference

- 1) Huang, Junhao, et al. "Revisiting Keccak and Dilithium Implementations on ARMv7-M." IACR Transactions on Cryptographic Hardware and Embedded Systems 2024.2 (2024): 1-24.
- 2) Huang, Junhao, et al. "Improved plantard arithmetic for lattice-based cryptography." IACR Transactions on Cryptographic Hardware and Embedded Systems 2022.4 (2022): 614-636.
- 3) Huang, Junhao, et al. "Improved plantard arithmetic for lattice-based cryptography." IACR Transactions on Cryptographic Hardware and Embedded Systems 2022.4 (2022): 614-636.
- 4) Abdulrahman, Amin, et al. "Faster kyber and dilithium on the cortex-m4." International Conference on Applied Cryptography and Network Security. Cham: Springer International Publishing, 2022.
- 5) Bos, Joppe W., Joost Renes, and Amber Sprenkels. "Dilithium for memory constrained devices." International Conference on Cryptology in Africa. Cham: Springer Nature Switzerland, 2022.
- 6) Abdulrahman, A., Hwang, V., Kannwischer, M. J., & Sprenkels, D. (2022). Faster Kyber and Dilithium on the Cortex-M4. Cryptology ePrint Archive.
- 7) Becker, H., Hwang, V., Kannwischer, M. J., Yang, B. Y., & Yang, S. Y. (2021). Neon NTT: faster dilithium, kyber, and saber on cortex-a72 and apple M1. Cryptology ePrint Archive.