05

PQC 최적화 I

양자 컴퓨터 개발 현황 (Google vs IBM)

2019년 구글에서 양자 컴퓨터 supremacy 달성
(난수성 문제 증명: 10,000년 vs 3분 20초)


Article

Quantum supremacy using a programmable superconducting processor

https://doi.org/10.1038/s41586-019-1666-5
Received: 22 July 2019
Accepted: 20 September 2019
Published online: 23 October 2019

Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven & John M. Martinis

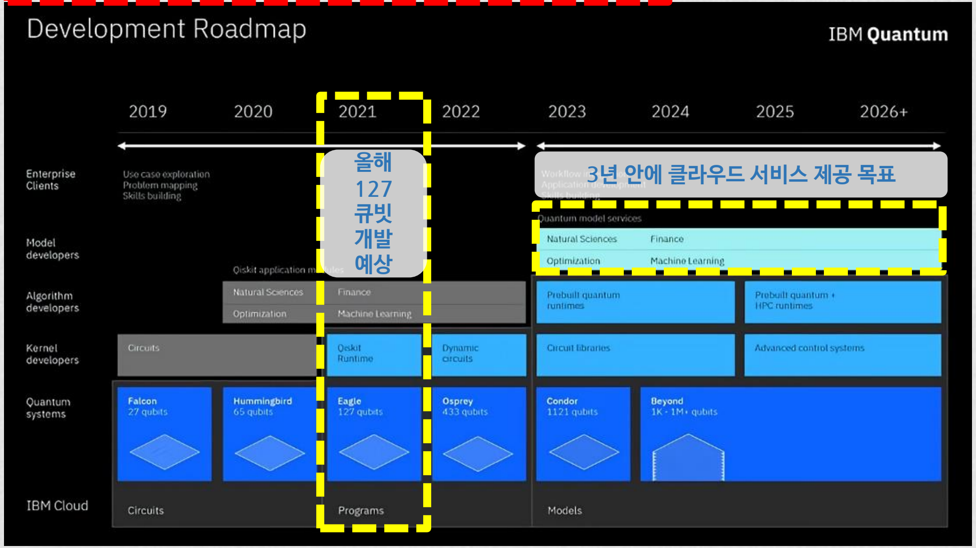
The promise of quantum computers is that certain computational tasks might be executed exponentially faster on a quantum processor than on a classical processor[1]. A fundamental challenge is to build a high-fidelity processor capable of running quantum algorithms in an exponentially large computational space. Here we report the use of a processor with programmable superconducting qubits[2–7] to create quantum states on 53 qubits, corresponding to a computational state-space of dimension 2^{53} (about 10^{16}). Measurements from repeated experiments sample the resulting probability distribution, which we verify using classical simulations. Our Sycamore processor takes about 200 seconds to sample one instance of a quantum circuit a million times—our benchmarks currently indicate that the equivalent task for a state-of-the-art classical supercomputer would take approximately 10,000 years. This dramatic increase in speed compared to all known classical algorithms is an experimental realization of quantum supremacy[8–14] for this specific computational task, heralding a much-anticipated computing paradigm.

In the early 1980s, Richard Feynman proposed that a quantum computer would be an effective tool with which to solve problems in physics and chemistry, given that it is exponentially costly to simulate large quantum systems with classical computers[1]. Realizing Feynman's vision poses substantial experimental and theoretical challenges. First, can a quantum system be engineered to perform a computation in a large enough computational (Hilbert) space and with a low enough error rate to provide a quantum speedup? Second, can we formulate a problem that is hard for a classical computer but easy for a quantum computer? By computing such a benchmark task on our superconducting qubit processor, we tackle both questions. Our experiment achieves quantum supremacy, a milestone on the path to full-scale quantum computing[8–14].

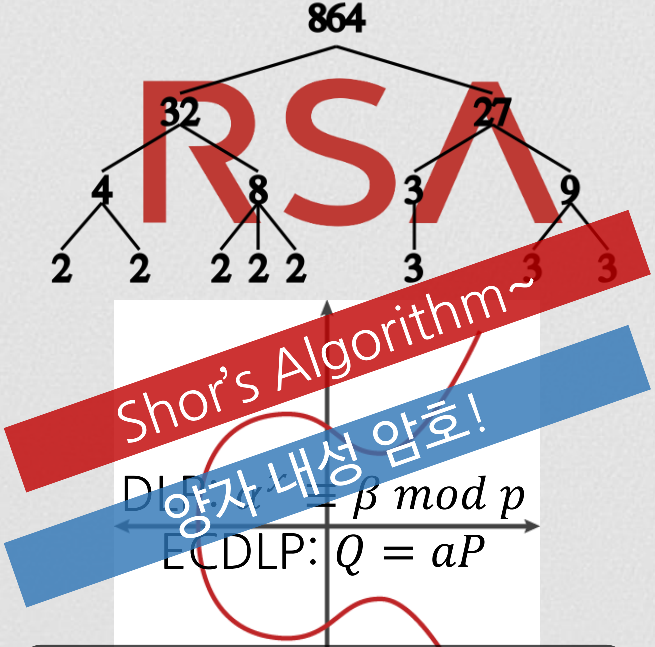
In reaching this milestone, we show that quantum speedup is achievable in a real-world system and is not precluded by any hidden physical laws. Quantum supremacy also heralds the era of noisy intermediate-scale quantum (NISQ) technologies[15]. The benchmark task we demonstrate has an immediate application in generating certifiable random numbers (S. Aaronson, manuscript in preparation); other initial uses for this new computational capability may include optimization[16,17], machine learning[18–21], materials science and chemistry[22–24]. However, realizing the full promise of quantum computing (using Shor's algorithm for factoring, for example) still requires technical leaps to engineer fault-tolerant logical qubits[25–29].

To achieve quantum supremacy, we made a number of technical advances which also pave the way towards error correction. We

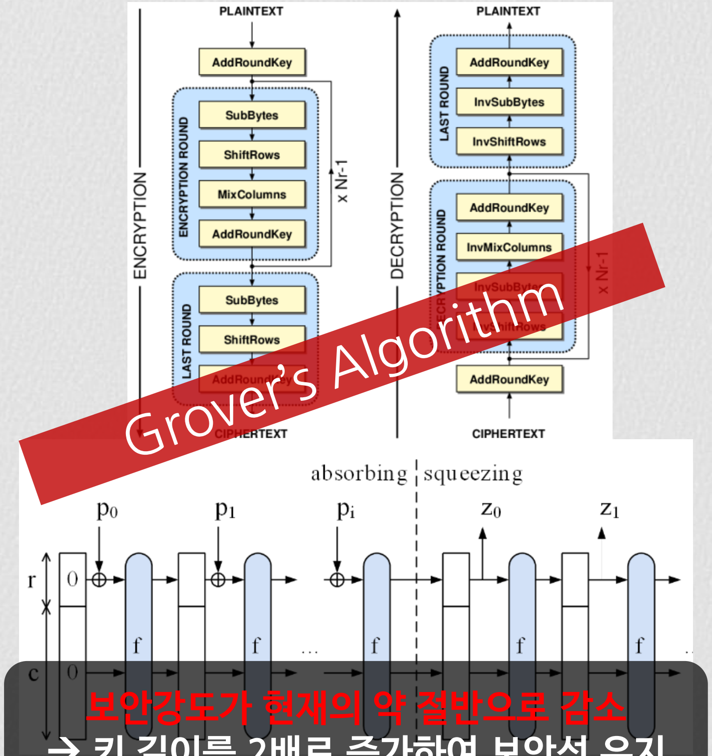
2025년 IBM에서 약 4~5천 큐빗 양자컴퓨터 개발 계획 발표
낮은 보안 강도를 가지는 RSA/ECC의 경우 실제 해킹 가능



현대 암호의 위기와 차세대 암호 알고리즘



보안 강도와 상관없이 취약성 존재
→ 다른 대안 암호로 교체 필요



보안강도가 현재의 약 절반으로 감소
→ 키 길이를 2배로 증가하여 보안성 유지

양자 내성 암호 공모전

양자 알고리즘과 양자 내성 암호

- 양자 알고리즘 → 현대 공개키 암호의 붕괴
- 새로운 공개키 암호의 필요 → 양자 내성 암호 공모전 (NIST)

양자 내성 암호 공모전

양자 내성 암호 표준화 일정

- 초기 계획과 달리 일정이 점차 앞당겨지고 있음

기존 일정	수정된 일정	내용
2017. 11	2017. 11	알고리즘 제안 마감
2018. 04	2018. 04	제안 알고리즘 소개
2018~2019	2018~2019	1차 평가분석 진행 (1차 후보 선정)
2019. 08	2019. 01	1차 선정 결과 발표
2020~2021	2019~2020	2차 평가분석 진행 (2차 후보 선정)
2022~	2020. 07	2차 선정 결과 발표
	2020~2022	3차 평가분석 진행(최종 후보 선정)
	2022~2024	최종 선정 결과 발표 및 표준화

양자 내성 암호 공모전 Round 3 (finalist, alternate)

격자 기반 암호가 가장 큰 관심을 받고 있음

- 현재 격자기반 암호 알고리즘이 가장 많이 후보군에 선정됨

종류	구분	Finalist	Alternate	장점	단점
격자	키교환	KYBER, NTRU, SABER	FrodoKEM, NTRU Prime	고속 구현	파라미터 설정 어려움
	서명	DILITHIUM, FALCON	-		
다변수 다항식	키교환	-	-	작은 서명, 고속 구현	큰 키 사이즈
	서명	Rainbow	GeMSS		
해시	키교환	-	-	안전성 증명 가능	큰 서명 사이즈
	서명	-	SPHINCS+		

양자 내성 암호 공모전 Round 3 (finalist, alternate)

종류	구분	Finalist	Alternate	장점	단점
아이소지니	키교환	–	SIKE	작은 키 사이즈	느린 속도
	서명	–	–		
코드	키교환	Classic McEliece	BIKE, HQC	고속 구현	큰 키 사이즈
	서명	–	–		
영지식	키교환	–	–	number-theoretic 혹은 structured hardness에 기반하지 않음	큰 서명 사이즈
	서명	–	Picnic		

타겟 프로세서

일정한 제한을 두고 양자내성암호 벤치마크를 수행 중에 있음

- STM32F4DISCOVERY
  - ARM Cortex-M4
  - 32-bit ARMv7E-M
  - 192KB RAM, 168 MHz

What NIST wants

2 라운드 주요 관심사

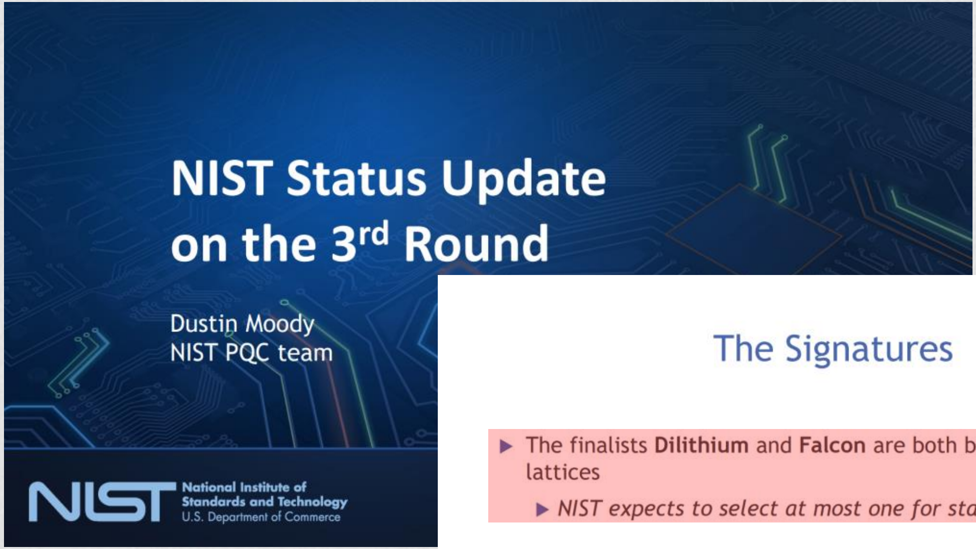
- Performance (hardware+software) will play more of a role
  - More benchmarks
  - For hardware, NIST asks to focus on Cortex M4 (with all options) and Artix-7
    - pqc-hardware-forum
- Continued research and analysis on **ALL** of the 2nd round candidates

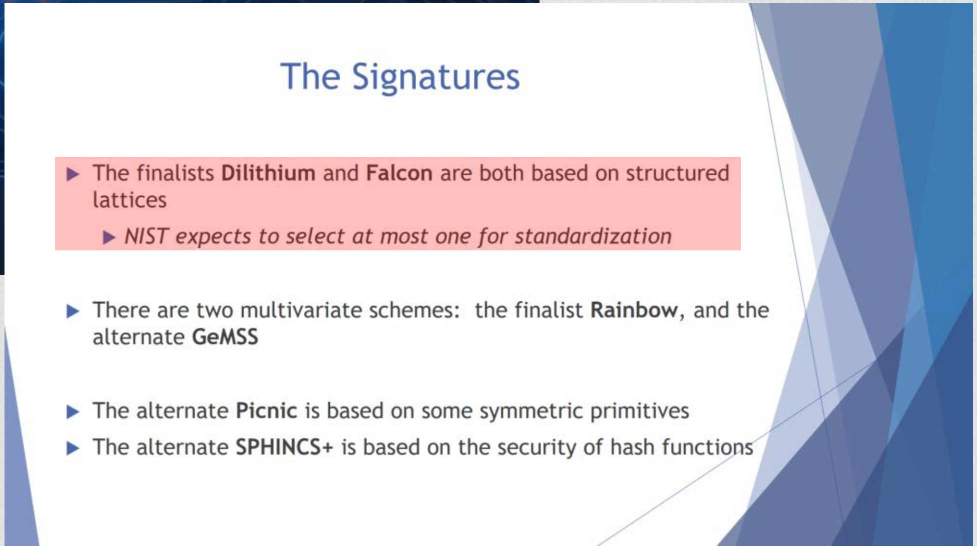
2019년, NIST에서는 임베디드 장비에 대한 벤치마크를 요구

- See how submissions fit into applications/procotols. Any constraints?

격자 기반 양자 내성 암호

격자 기반 서명 알고리즘인 Dilithium과 Falcon 중 하나 선정 예정





Dilithium 서명 알고리즘 (NIST 3 라운드 finalist)

- 작은 키와 서명 크기를 가지고 있음
- **Polynomial ring 상에서 연산:** $R_q = Z_q[X]/(X^{256}+1)$, $q = 8380417$
- **NTT 연산을 통해 최적화 구현:** $a \cdot b = INTT(NTT(a) \bullet NTT(b))$

알고리즘	NIST Level	Public Key [bytes]	Signature [bytes]	Rejection sampling의 예상 반복 횟수
Dilithium2	1	1,184	2,044	5.9
Dilithium3	2	1,472	2,701	6.6
Dilithium4	3	1,760	3,366	4.3

Algorithm 1 Dilithium key generation

Output: Secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
Output: Public key $pk = (\rho, \mathbf{t}_1)$
$\rho \leftarrow \{0,1\}^{256}$
$K \leftarrow \{0,1\}^{256}$
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$
$\hat{\mathbf{A}} \in R_q^{k\times\ell} := \mathtt{ExpandA}(\rho)$
$\mathbf{t} := \mathtt{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathtt{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$
$(\mathbf{t}_1, \mathbf{t}_0) := \mathtt{Power2Round}(\mathbf{t})$
$tr \in \{0,1\}^{384} := \mathcal{H}(\rho || \mathbf{t}_1)$

Algorithm 2 Dilithium signature generation

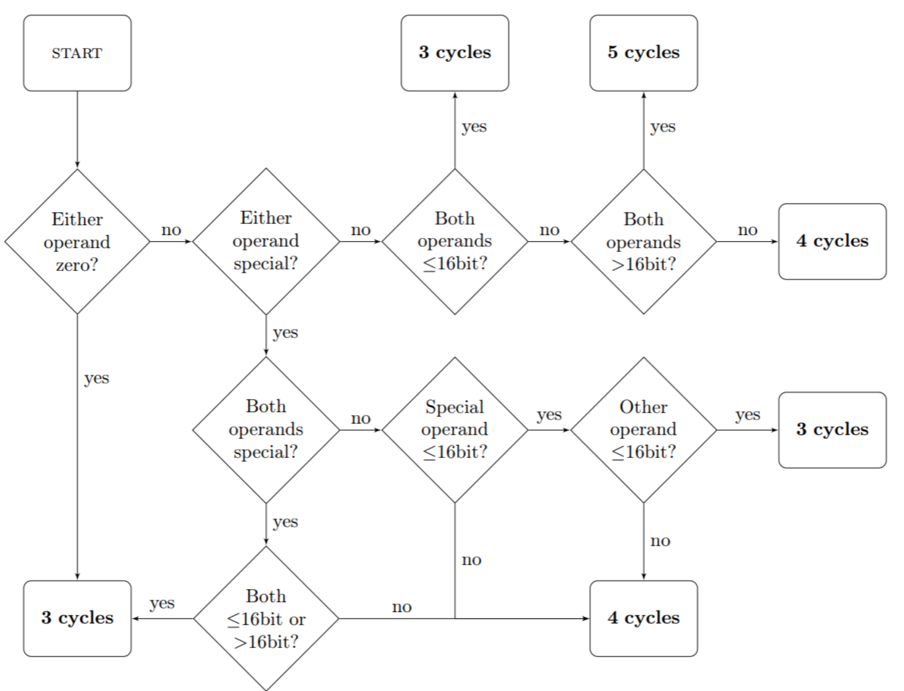
Input: Secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
Input: Message $M \in \{0,1\}^*$
Output: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
$\hat{\mathbf{A}} \in R_q^{k\times\ell} := \mathtt{ExpandA}(\rho)$
$\mu \in \{0,1\}^{384} := \mathcal{H}(tr || M)$
$\kappa := 0; (\mathbf{z}, \mathbf{h}) = \perp$
$\rho' \in \{0,1\}^{384} := \mathcal{H}(K || \mu)$
$\hat{\mathbf{s}}_1 := \mathtt{NTT}(\mathbf{s}_1); \hat{\mathbf{s}}_2 := \mathtt{NTT}(\mathbf{s}_2); \hat{\mathbf{t}}_0 := \mathtt{NTT}(\mathbf{t}_0)$
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  $\mathbf{y} \in S_{\gamma_1 - 1}^\ell := \mathtt{ExpandMask}(\rho', \kappa)$
  $\mathbf{w} := \mathtt{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathtt{NTT}(\mathbf{y}))$
  $\mathbf{w}_1 := \mathtt{HighBits}(\mathbf{w})$
  $\hat{c} := \mathtt{NTT}(\mathcal{H}_B(\mu || \mathbf{w}_1))$
  $\mathbf{z} := \mathbf{y} + \mathtt{NTT}^{-1}(\hat{c} \circ \hat{s}_1)$
  $(\mathbf{r}_1, \mathbf{r}_0) := \mathtt{Decompose}(\mathbf{w} - \mathtt{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_2))$
  if $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ **or** $||\mathbf{r}_0||_\infty \geq \gamma_2 - \beta$ **or** $\mathbf{r}_1 \neq \mathbf{w}_1$ **then**
    $(\mathbf{z}, \mathbf{h}) = \perp$
  else
    $\mathbf{h} := \mathtt{MakeHint}\left(-\mathtt{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0), \mathbf{w} - \mathtt{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_2) + \mathtt{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0)\right)$
    if $||\mathtt{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0)||_\infty \geq \gamma_2$ **or** $\#$ 1's in $\mathbf{h} > \omega$ **then**
      $(\mathbf{z}, \mathbf{h}) = \perp$

Algorithm 3 Dilithium verification

Input: Public key $pk = (\rho, \mathbf{t}_1)$
Input: Message $M \in \{0,1\}^*$
Input: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
Output: `Valid` or `Invalid`
$\hat{\mathbf{A}} \in R_q^{k\times\ell} := \mathtt{ExpandA}(\rho)$
$\mu \in \{0,1\}^{384} := \mathcal{H}(\mathcal{H}(\rho || \mathbf{t}_1) || M)$
$\mathbf{w}'_1 := \mathtt{UseHint}(\mathbf{h}, \mathtt{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathtt{NTT}(\mathbf{z}) - \mathtt{NTT}(c) \circ \mathtt{NTT}(2^d \cdot \mathbf{t}_1)))$
if $c = \mathcal{H}_B(\mu || w'_1)$ **and** $||\mathbf{z}||_\infty < \gamma_1 - \beta$ **and** $\#$ 1's in $\mathbf{h} \leq \omega$ **then**
  return `Valid`
else
  return `Invalid`

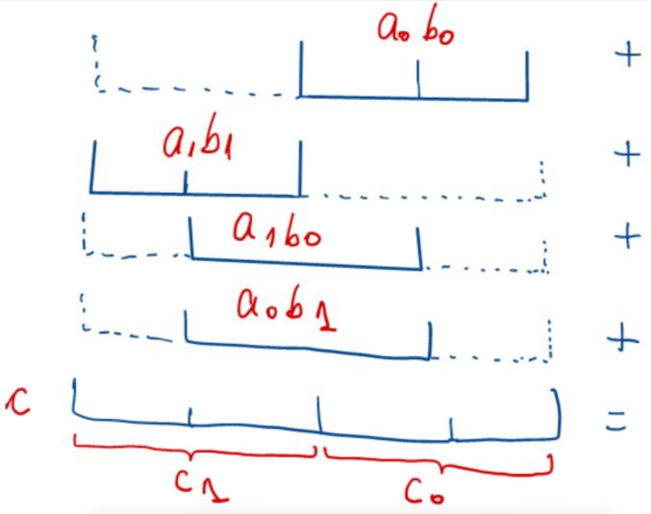
ARM Cortex-M3/M4 상에서의 Constant Timing 곱셈 구현

- UMULL RdHi, RdLo, Rm, Rs→ RdHi : RdLo := Rm * Rs
  - unsigned 64-비트 곱셈기 명령어
  - M4에서는 single cycle
  - M3에서는 경우에 따라 다름 → Timing Attack 가능



M3 상에서의 Constant Timing 구현

- **64-비트 곱셈기는 variable timing**
- **대신 32-비트 곱셈기는 constant timing**
  - Radix 2^{16}을 이용하여 64-비트 값 표현
  - $a = 2^{16}a_1 + a_0$ 그리고 $b = 2^{16}b_1 + b_0$
    $(0 \leq a_0, b_0 < 2^{16}, -2^{15} \leq a_1, b_1 < 2^{15})$
  - $ab = 2^{32}a_1b_1 + 2^{16}(a_0b_1 + a_1b_0) + a_0b_0 \; (-2^{31} \leq a_ib_j < 2^{31})$



M3/M4 상에서의 Constant Timing 구현

Listing 1 CT butterfly from [GKOS18]

```
; q=8380417, qinv=4236238847
; Input: p0, p1, twiddle
; Output: p0, p1
umull tmp0, tmp1, p1, twiddle
mul   pol1, tmp0, qinv
umlal tmp0, tmp1, p1, q
add   p1, p0, q, lsl#1
sub   p1, p1, tmp1
add   p0, p0, tmp1
```

Listing 2 Our CT butterfly

```
; 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
```

Listing 3 GS butterfly in [GKOS18]

```
; q=8380417, qinv=4236238847
; Input: p0, p1, twiddle
; Output: p0, p1
add   tmp0, p0, q, lsl#8
sub   tmp0, tmp0, p1
add   p0, p0, p1
umull tmp1, p1, tmp0, twiddle
mul   tmp0, tmp1, qinv
umlal tmp1, p1, tmp0, q
```

Listing 4 Our GS butterfly

```
; 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
```

기존 unsigned를 signed로 변경

Listing 5 Schoolbook SMULL (SBSMULL)

```
; Input:  a = a0 + a1*2^16
;         b = b0 + b1*2^16
; Output: c = a*b = c0 + c1*2^32
mul  c0, a0, b0
mul  c1, a1, b1
mul  tmp, a1, b0
mla  tmp, a0, b1, tmp
adds c0, c0, tmp, lsl #16
adc  c1, c1, tmp, asr #16
```

Listing 6 Schoolbook SMLAL (SBSMLAL)

```
; Input:  a = a0 + a1*2^16
;         b = b0 + b1*2^16
;         c = c0 + c1*2^32
; Output: c = c + a*b
;           = c0 + c1*2^32
mul  tmp, a0, b0
adds c0, c0, tmp
mul  tmp, a1, b1
adc  c1, c1, tmp
mul  tmp, a1, b0
mla  tmp, a0, b1, tmp
adds c0, c0, tmp, lsl #16
adc  c1, c1, tmp, asr #16
```

M3의 경우 smull/smlal이 variable timing

→ mul과 mla을 통해 constant timing으로 재 구현

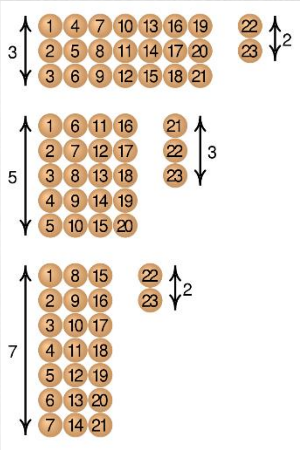
NTT

- **두 polynomial (*a* 그리고 *b*) 에 대한 연산 결과 (*c*):**
  1) 입력인자 (*a* 그리고 *b*) 를 *N*개의 degree-0 polynomials 으로 변환 (Forward transform; NTT).

  2) Pointwise multiplication 을 수행

  3) Chinese remainder theorem을 통해 본래의 값 복구 (*c*) (Backward transform; INTT).

Chinese remainder theorem 예시
$x \equiv 2 \ (mod\ 3) \equiv 3 \ (mod\ 5) \equiv 2 \ (mod\ 7)$
해답 $x = 23 + 105k$, k는 integer



NTT 수학 (1 / 2)

- **Case study (reduction polynomial을 나누기)**
  1) $R_q = Z_q[X]/(X^{256}+1)$
  2) $(X^{128}-\alpha)(X^{128}+\alpha) = (X^{256}+1)$

$$(X^{256}+1) = (X^{128}-\alpha)(X^{128}+\alpha)$$

$$(X^{256}+1) = X^{256} + (-\alpha+\alpha)X^{128} - \alpha^2$$

$$1 = -\alpha^2$$

$$\alpha^2 = -1$$

$$\alpha^4 = 1, \alpha = \sqrt[4]{1}$$

(Fourth primitive root of 1)

NTT 수학 (2 / 2)

- **Case study (reduction polynomial을 나누기)**

1) $R_q = Z_q[X]/(X^{256} + 1)$
2) $(X^{128} - \alpha)(X^{128} + \alpha) = (X^{256} + 1)$
3) $(X^{128} + \alpha) = (X^{64} - \gamma)(X^{64} + \gamma)$

$$(X^{128} + \alpha) = X^{128} + (-\gamma + \gamma)X^{64} - \gamma^2$$

$$\alpha = -\gamma^2$$

$$\gamma = \sqrt{-\alpha} = \sqrt{(-1) \cdot \alpha} = \sqrt{\alpha^2 \cdot \alpha} = (\sqrt{\alpha})^3$$
$$= (\sqrt{\zeta_4})^3 = \zeta_8^3$$

$$(X^{128} - \alpha) = (X^{64} - \beta)(X^{64} + \beta)$$

$$(X^{128} - \alpha) = X^{128} + (-\beta + \beta)X^{64} - \beta^2$$

$$-\alpha = -\beta^2$$

$$\beta = \sqrt{\alpha} = \sqrt{\zeta_4} = \zeta_8$$

ζ_k 는 k-th primitive root of 1, 혹은 수학식은 다음과 같음 ($\zeta_k = \sqrt[k]{1}$).

NTT 예시 (1 / 3)

- **Case study (reduction polynomial을 나누기)**

1) $R_q = Z_q[X]/(X^{256}+1)$

2) $(X^{128}-\alpha)(X^{128}+\alpha) = (X^{256}+1)$

3) $(X^{128}+\alpha) = (X^{64}-\gamma)(X^{64}+\gamma)$ $\quad (X^{128}-\alpha) = (X^{64}-\beta)(X^{64}+\beta)$

4) $(X^{256}+1) = (X^{64}-\zeta_8)(X^{64}+\zeta_8)(X^{64}-\zeta_8^3)(X^{64}+\zeta_8^3)$

Degree-0 polynomials에 도달할 때까지 나누기

5) $(X^{256}+1) = (X-\zeta_{512})(X+\zeta_{512})(X-\zeta_{512}^{129})(X+\zeta_{512}^{129})\cdots(X-\zeta_{512}^{127})(X+\zeta_{512}^{127})(X-\zeta_{512}^{255})(X+\zeta_{512}^{255})$

NTT 예시 (2 / 3)

- **Case study (reduction polynomial을 나누기)**

1) $a \in Z_q[X]/(X^{256}+1)$

$$(X^{128}-\alpha)(X^{128}+\alpha) = (X^{256}+1)$$

a_L a_R

2) $a_L \in Z_q[X]/(X^{128}-\alpha)$

$$X^{128} = \alpha$$

$$a_i\ X^i = a_i\ \alpha X^{i-128}$$

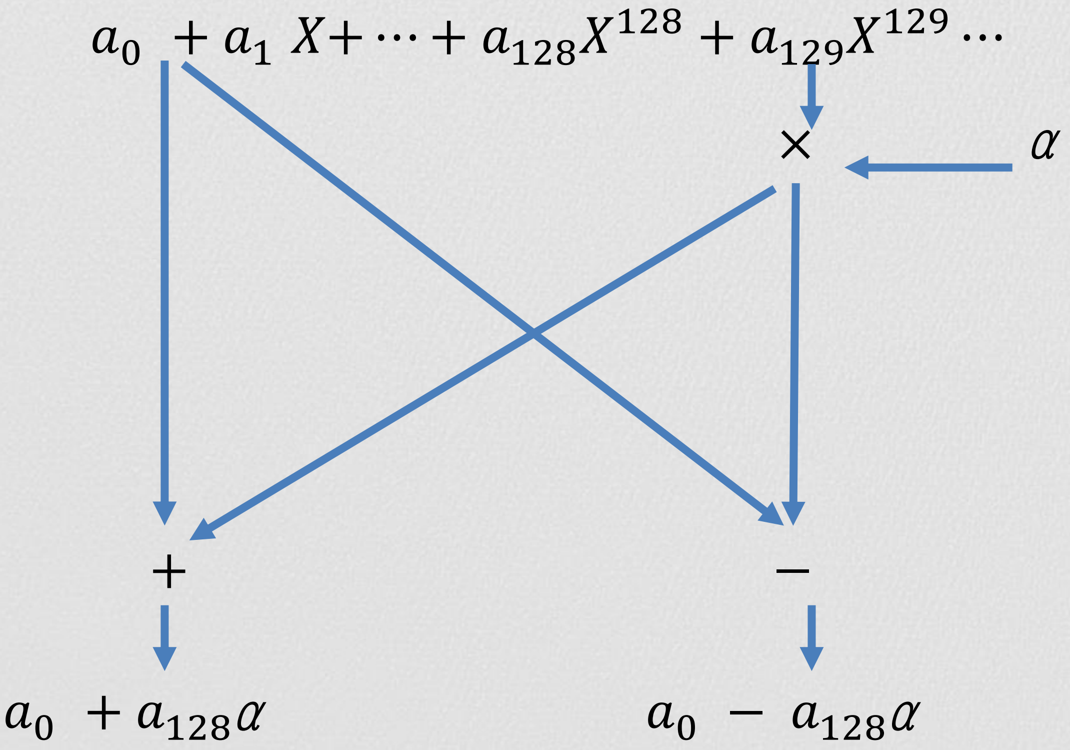
3) $a_L = (a_0 + a_{128}\alpha) + (a_1 + a_{129}\alpha)X + (a_2 + a_{130}\alpha)X^2 + \cdots$

유사하게 a_R 에 적용 가능~

4) $a_R = (a_0 - a_{128}\alpha) + (a_1 - a_{129}\alpha)X + (a_2 - a_{130}\alpha)X^2 + \cdots$

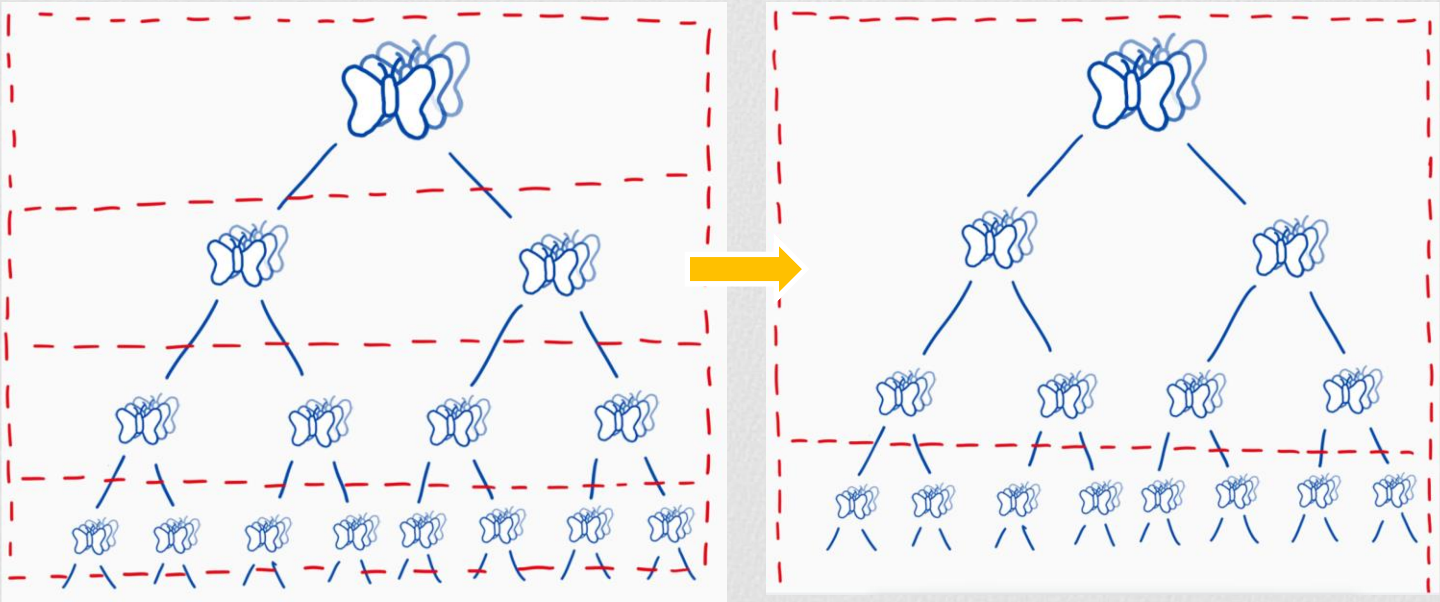
NTT 예시 (3 / 3)

- Case study (reduction polynomial을 나누기)
  - Butterfly diagram



Inverse NTT 는 NTT 연산과 유사!

Merging layers (NTT) for M3/M4



프로세서 별 constant-time 구현상세
- M4: 2 layers를 merging 함
- M3: layer들을 merging 하지 않음

Modular Operation: Barret reduction

$$\frac{1}{b} = \frac{(2^k)/b}{b \cdot (2^k)/b} = \frac{(2^k)/b}{2^k} \approx \frac{x}{2^k}$$

$$x = \lfloor 2^k/b \rfloor$$

$$k = \log_2(a)$$

근사치를 이용한 modular operation

Input: Two numbers a and b, parameter k, $x = \left\lfloor \frac{2^k}{b} \right\rfloor$

Output: $a \pmod b$

```
q ← (a × x) >> k;
r ← a − q × b;
if r ≥ b then
  |  r ← r − b
end
return r
```

Modular Operation: Montgomery reduction

- 2의 배수의 값으로 나누어지도록 숫자를 변환하여 계산
- z는 R^2 승수를 가지는 임의의 수를 의미
- 연산 복잡도는 n 워드 연산 시 $2n^2 + n$ 곱셈 필요

$$\tilde{x} = xR$$

$$Mont(\tilde{x}, \tilde{y}) = \tilde{x}\tilde{y}R^{-1}\, mod\, p = xyR\, mod\, p.$$

$$c \leftarrow \frac{z + (zp' mod\, R)p}{R}, p' = -p^{-1}\, mod\, R$$

만약 $c \geq p$ 라면 $c \leftarrow c - p$ 수행

z는 곱셈 결과값, p는 modulus 값, c는 reduction 이후 값

Modular Operation: *K*−Montgomery reduction

- $q = 12289 = 3 \cdot 2^{12} + 1$, 여기서 3을 k로 설정
- 모듈러가 $q = k \cdot 2^{m} + 1$이고 입력 인자 $0 \leq a, b \leq q$인 경우
- $C = a \cdot b \rightarrow 0 \leq C \leq q^2 = k^2 2^{2m} + k2^{m+1} + 1$
- $C = C_0 + 2^m C_1 \ (0 \leq C_0 \leq 2^m)$
- $0 \leq C_1 = \frac{C - C_0}{2^m} < k^2 2^m + 2k + \frac{1}{2^m} = kq + k + \frac{1}{2^m}$
- $kC \equiv kC_0 - C_1 \ (mod\ q), |kC_0 - C_1| < \left(k + \frac{1}{2^m}\right) q$

function K-RED(C)
    $C_0 \leftarrow C \bmod 2^m$
    $C_1 \leftarrow C / 2^m$
    return $kC_0 - C_1$
end function

Modular Operation: *K*–Montgomery reduction

- 몽고메리 reduction을 통해 격자기반 modulus에 대한 reduction 효율적으로 수행

- $\tilde{x} = x \cdot k^{-1}\ mod\ p\ , \tilde{y} = y \cdot k^{-1}\ mod\ p$

- $K - RED(\tilde{x} \cdot \tilde{y}) \equiv \tilde{x}\tilde{y}k \equiv xy \cdot k^{-1}\ (mod\ p)$

```
function K-RED(C)
    C_0 ← C mod 2^m
    C_1 ← C/2^m
    return kC_0 − C_1
end function
```