Engineering Lattice-based Cryptography

Sujoy Sinha Roy

Solving system of linear equations

- System of linear equations with unknown *s*

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ \vdots \\ b_m \end{pmatrix}$$

- Gaussian elimination solves *s* when number of equations $m \geq n$

System of linear equations with errors

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} + \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \\ \vdots \\ e_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ \vdots \\ b_m \end{pmatrix} \quad \text{mod } q$$

- Search **Learning with Errors** (LWE)-problem:
  Given **(A, b)** → computationally infeasible to solve **(*s*, *e*)**

- Decisional LWE-problem:
  Given **(A, b)** → hard to distinguish from random

Learning with rounding (LWR)

$$\left\lfloor \frac{p}{q} \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} \right\rceil = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ \vdots \\ b_m \end{pmatrix}$$

- Search LWE-problem:
  Given **(A, b)** → computationally infeasible to solve ***(s, e)***

- Decisional LWE-problem:
  Given **(A, b)** → hard to distinguish from random

LWE Diffie-Hellman key-exchange

Public: **A**

Secret **s, e**

$$b = A\,s + e$$

Secret **s', e'**

$$b' = s'^{T} A + e'^{T}$$

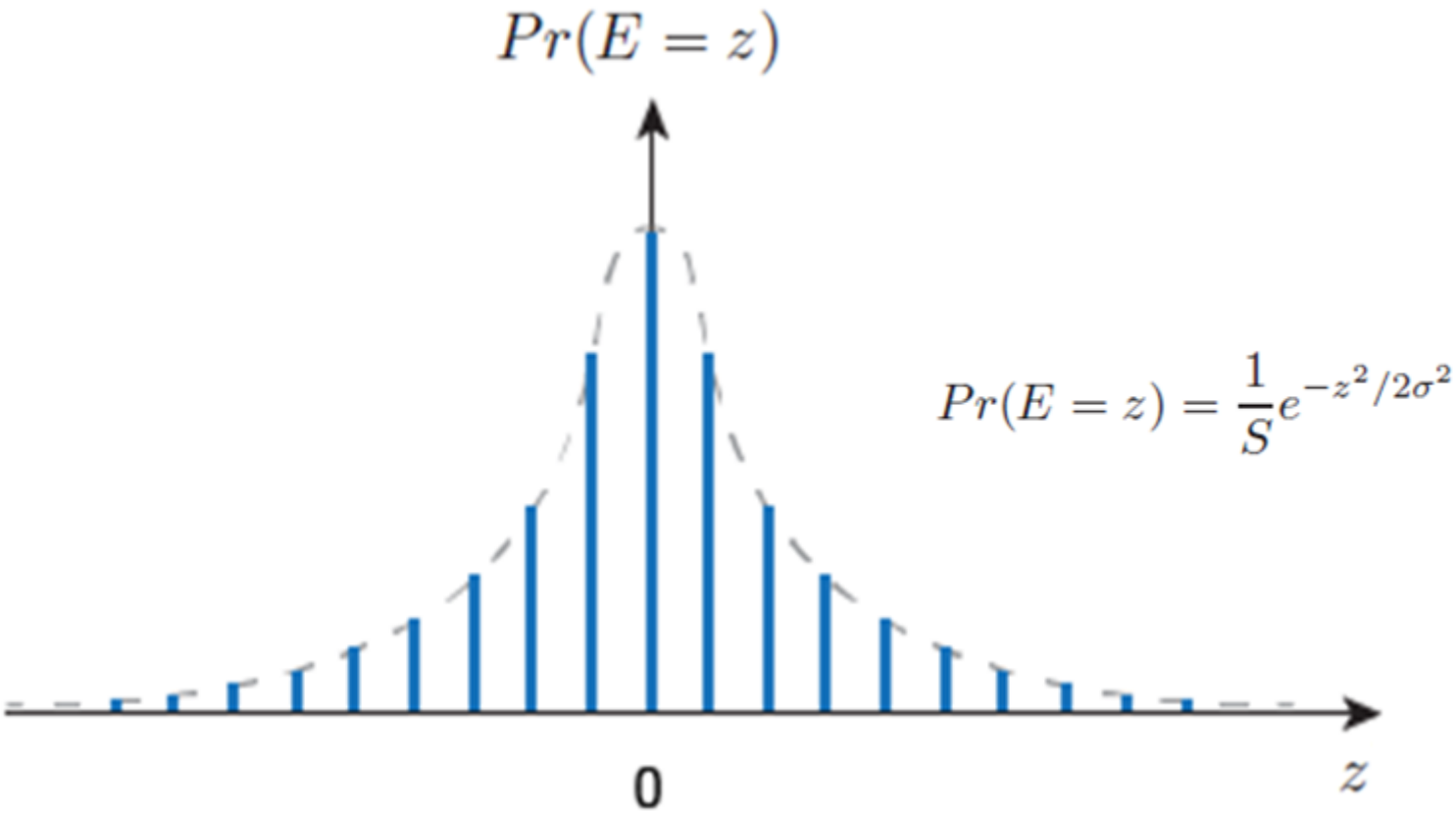
$$v = b'^{T} s$$

$$v' = s'^{T} b$$

Noisy **shared secret**

- **Error sampling**
- Matrix-vector multiplication

Sampling from a known discrete distribution



Knuth-Yao random walk

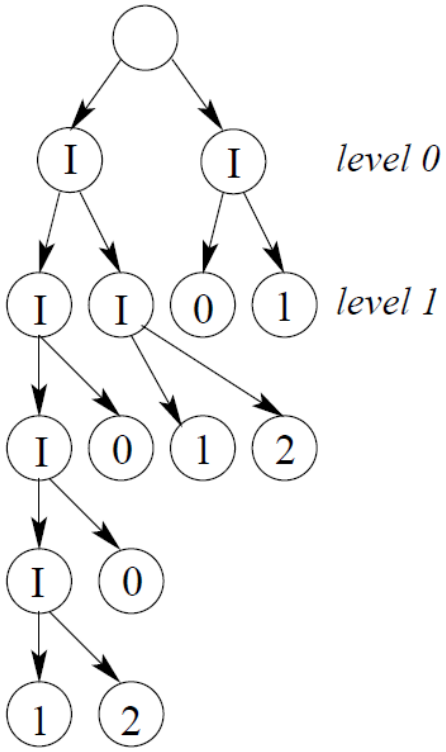
- Example: Let a sample space $S = \{ 0, 1, 2 \}$
  - $p_0 = 0.01110$
  - $p_1 = 0.01101$
  - $p_2 = 0.00101$

- Probability matrix

$$P_{mat} = \begin{array}{c} \\ row\ 0 \rightarrow \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccc} & & column\ 0 & & \\ & & \downarrow & & \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

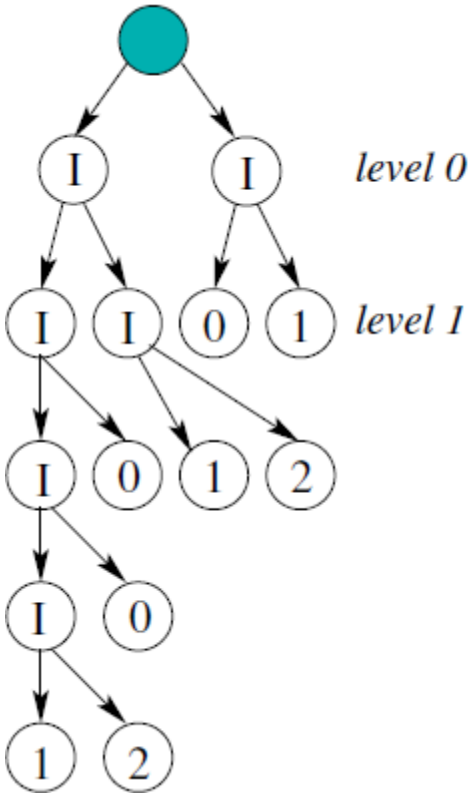
Knuth-Yao random walk: example

Binary tree corresponding to P_{mat}

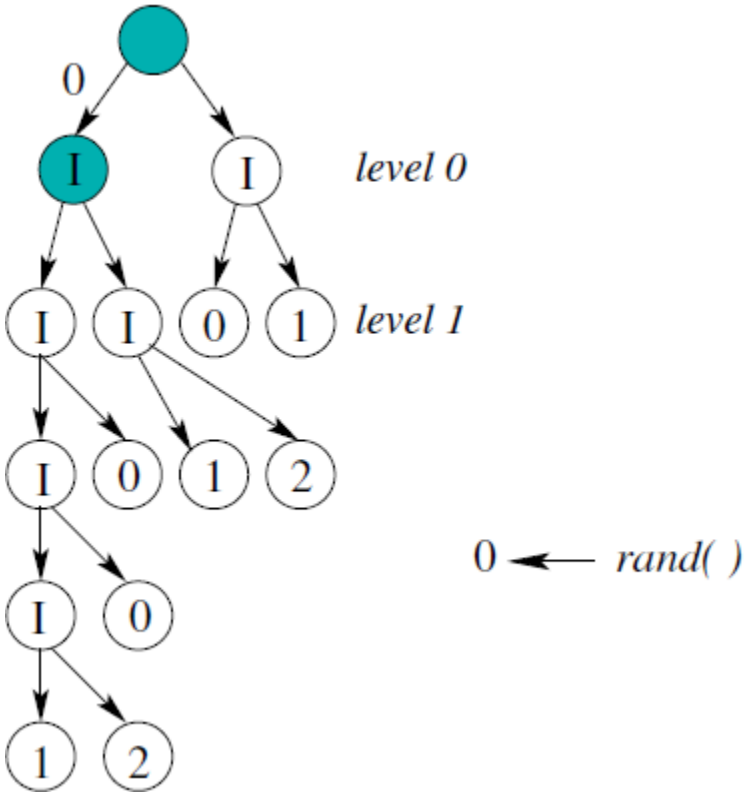


$$P_{mat} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

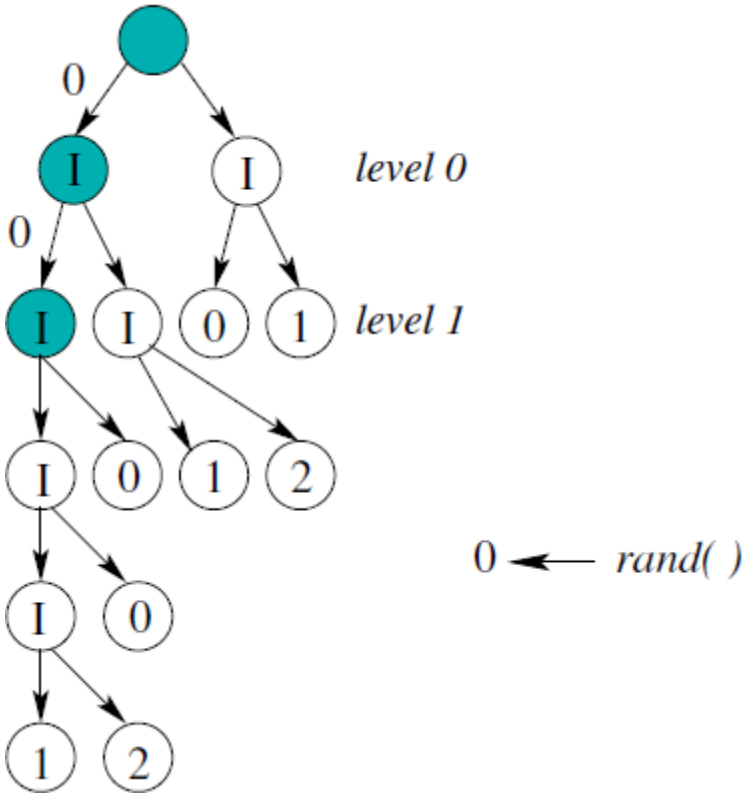
Knuth-Yao random walk: example



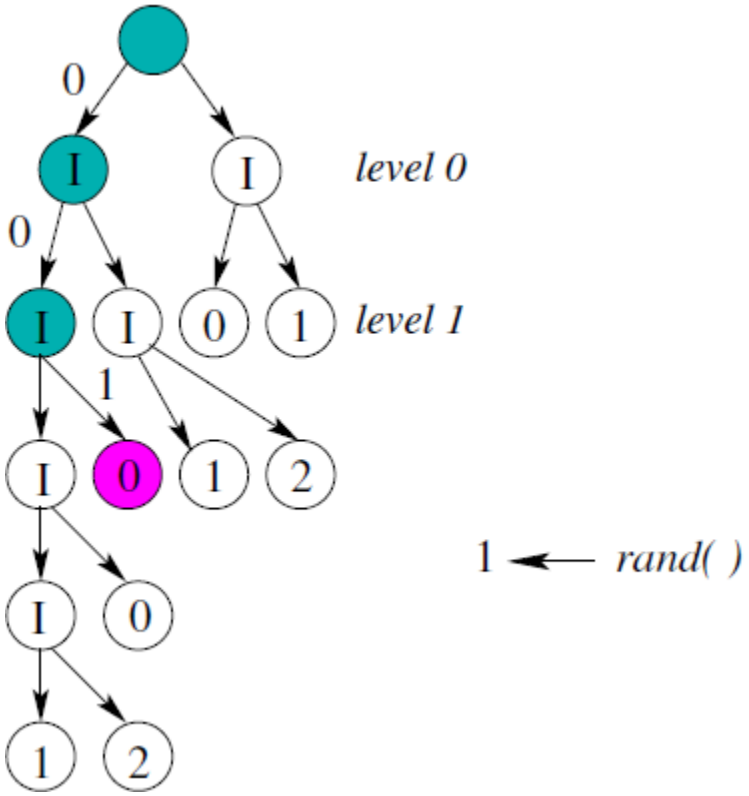
Knuth-Yao random walk: example



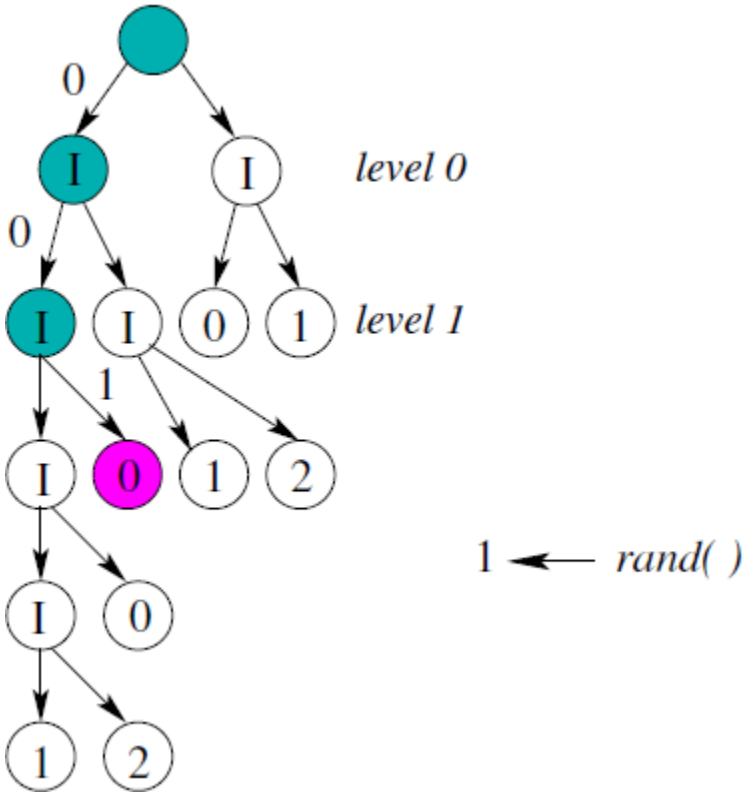
Knuth-Yao random walk: example



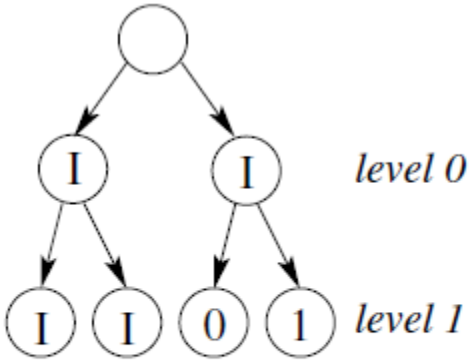
Knuth-Yao random walk: example



Knuth-Yao random walk: example



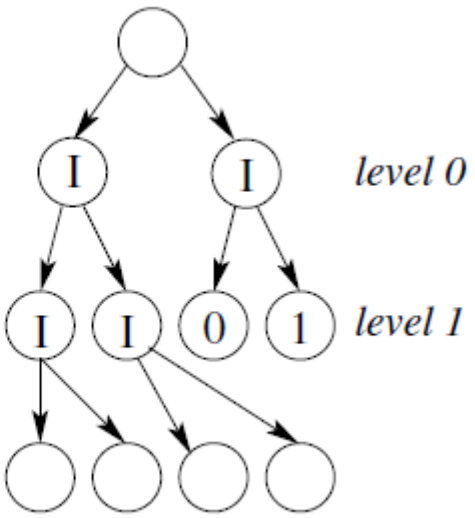
On-the-fly Probability Tree Generation



$$P_{mat} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

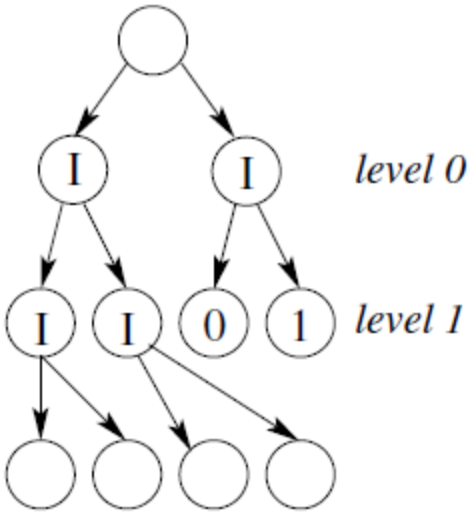
On-the-fly Probability Tree Generation

$$P_{mat} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$



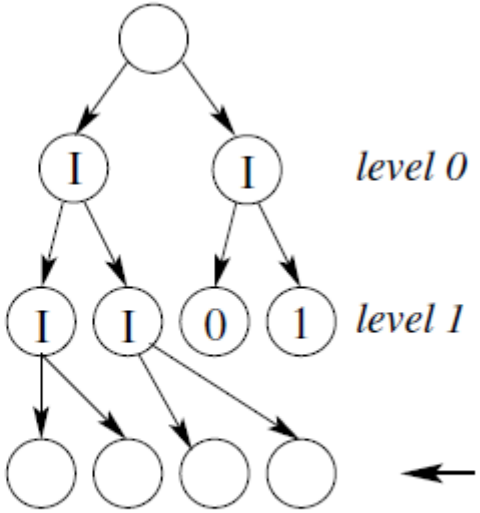
On-the-fly Probability Tree Generation

$$P_{mat} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$



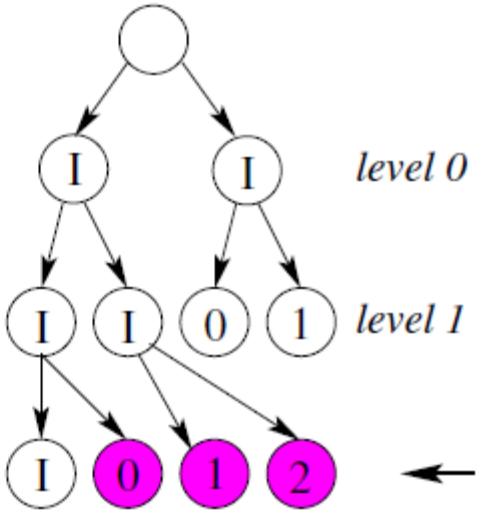
On-the-fly Probability Tree Generation

$$P_{mat} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$



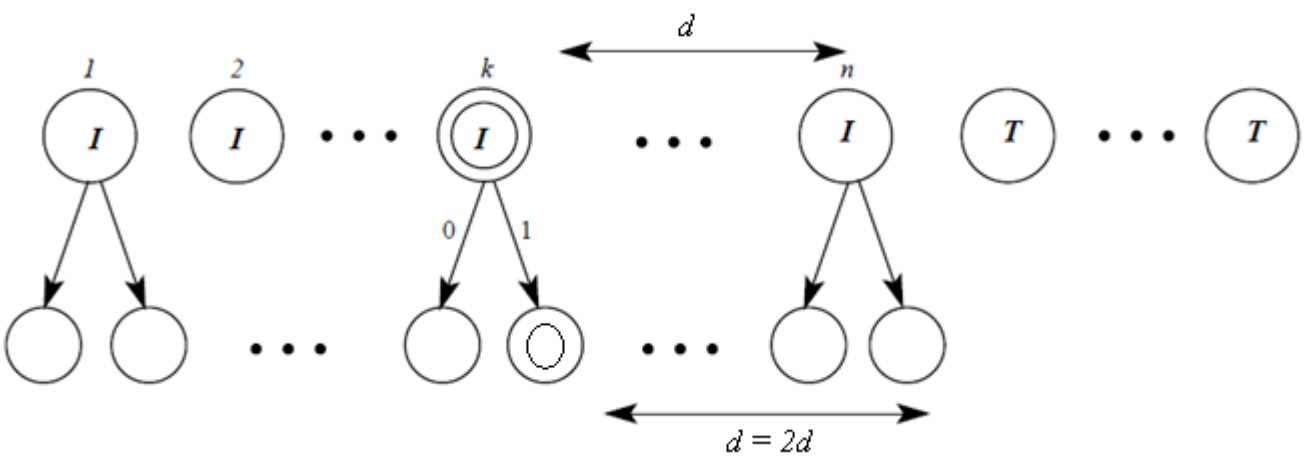
On-the-fly Probability Tree Generation

$$P_{mat} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$



Counter-based algorithm

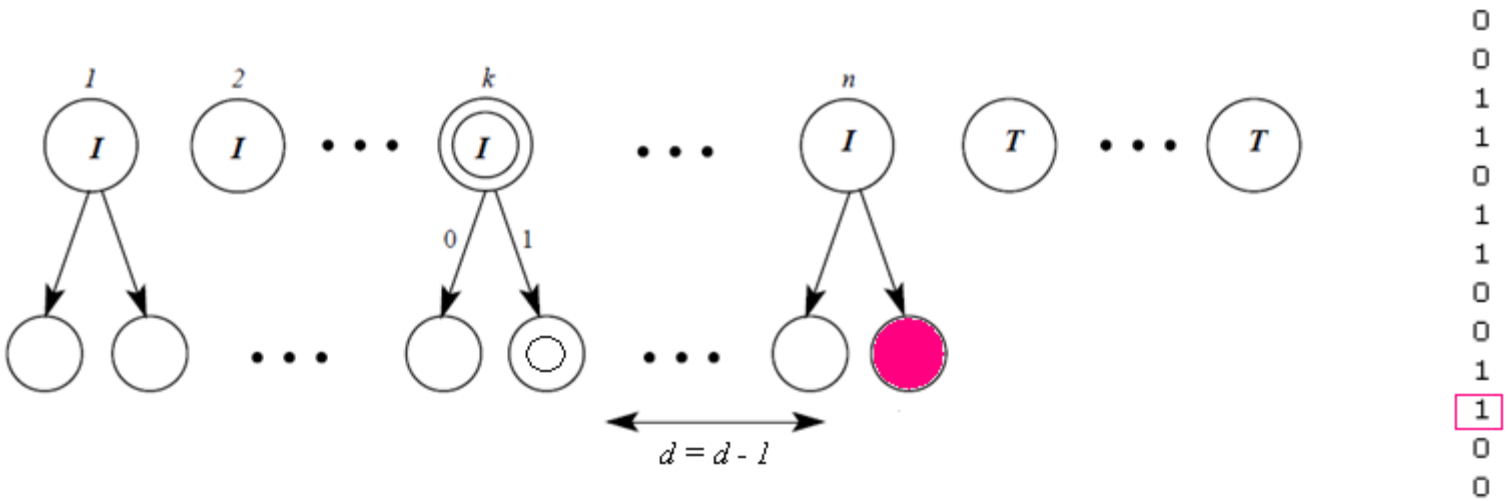
- Construction of *i-th* level during sampling : Counter *d* for distance



0
0
1
1
0
1
1
0
0
1
1
0
0

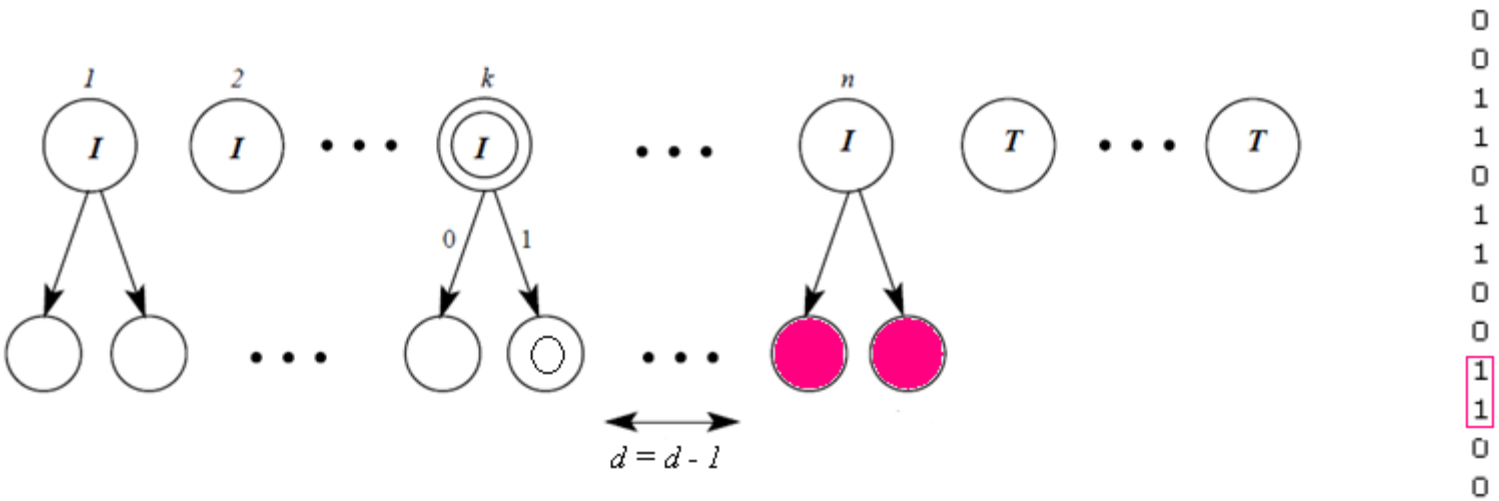
Counter-based algorithm

- Construction of *i-th* level during sampling : Counter *d* for distance



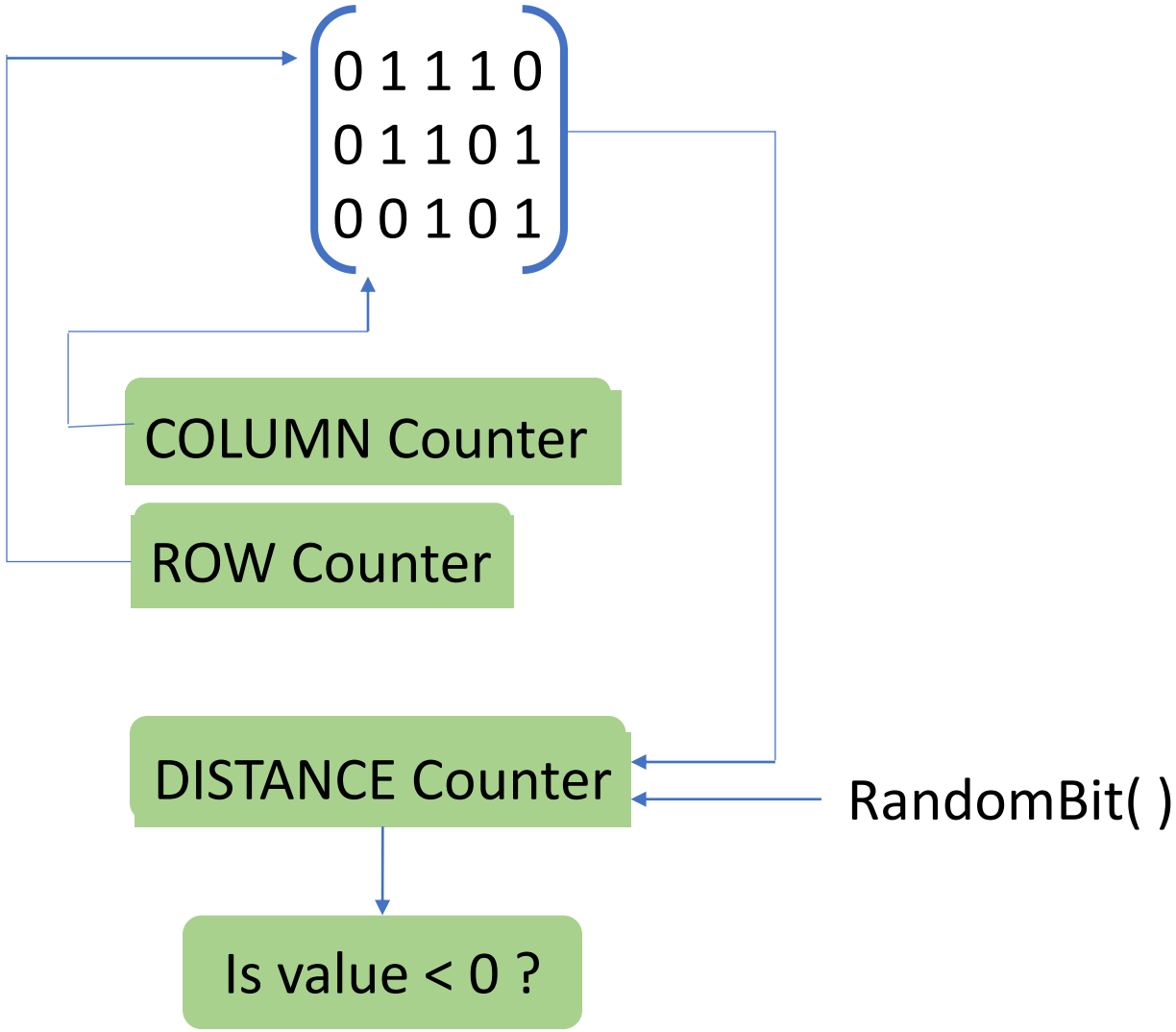
Counter-based algorithm

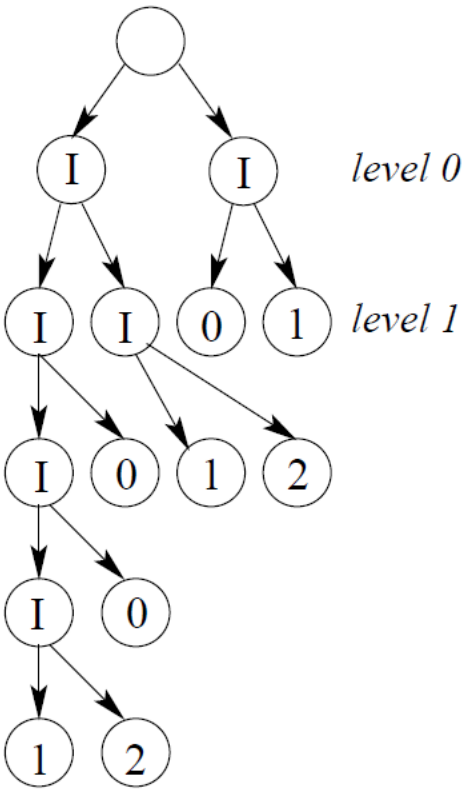
- Construction of *i-th* level during sampling : Counter *d* for distance



- When $d < 0$ for the first time, the visited node is a terminal node
- We need counters for *d* and row-number

Tree Traversal: Simple Algorithm





Memory optimization: Probability Matrix

```
000111111111101011100010111010101001011000110110010101010000001110000011111111101011100010111010101001011000
001111001101110110011011001101000010101001100000001111000011011110111011100111100110000000000011100101110111
001101001000110011101100011010110000000110111001000111100011011110011001011101001110011110001000010011000001
001010010010001110000011001110001110000011001101001001010101110001111011001000011011110110010101011111111111
000111010011001101100110100000100001001100011010000010101010101000011100111011000010001001000010111111101000
000100101100101100100011010010101110101011111110000101100111100010100101010000000111110111110000100011110001
000010101111011110010010001110000001101100100100100010101011010101111100000110111011101101101111011110011010
000000101110011011000100101100011011000100100100100110000110101011010110001110111100111010101001010001100101
000000010110010001011001010110111111001111001101111001011101010110111010111011101101001101101100001001111010
000000001001101100000011010001010000100010110100011100101000010110111000110111011111011010111011010100110101 1
000000000011110100100011111101111100000110101001001110000000101100110000101011111101010111111110010110001000 0
000000000001010111011101100100111011010010110110101010111011010000001000000110001110010101111111101011111110011
000000000000011100010111001100111100101101110110110011001000011011101111000101010100100101010010011110110100100111100100
000000000000001000010101101010110000011100101011000000100110110111111010010101100111110110111010000101011111
000000000000000100011100100010011110010000000101101000001101010101111110011010011000111111011011100011111100
000000000000000001000100110001110011100100010100110101111110011010010110000100001000010000111110001001010010000 0
000000000000000000001111000100101111010101111000010111011100111010110001001011001001010100111001101101000000
000000000000000000000010111111101101110110000001111011011001100001011100111011010101111101010011011110001 00
000000000000000000000000100010100010110101110110100011001101110000000011100100010110111100001000001100100001
000000000000000000000000000101101001001100101100100000010100100111111010110001000000111111101100000100000 11
000000000000000000000000000000110101100000011001001000011100110010011110011000100010001000001011001111110111
000000000000000000000000000000001110010111111001101001101100100001010011011101010001001111000110000001 1100
000000000000000000000000000000000011100000000011111101101001111011010011001110101001000111001010011111100
000000000000000000000000000000000000001100010111101111100111001100101101000110100011001100110110110011001 0
000000000000000000000000000000000000000010011111010001011011000011111111110010000100001111101011001100000
000000000000000000000000000000000000000000001110011001001010100000010100101010010010001111101100011001111
000000000000000000000000000000000000000000000100101111010011001101011000011001110011001101000110011001 0001
000000000000000000000000000000000000000000000001011010100011000011101110110000100100010100000000111010
000000000000000000000000000000000000000000000000001100010000010101100100000100111110110000010011101 0
000000000000000000000000000000000000000000000000000110000000100000011101110011011101000100000010111
000000000000000000000000000000000000000000000000000000101010110101110101011010001010011010101011
000000000000000000000000000000000000000000000000000000000100010100101000001001010010110111111001
000000000000000000000000000000000000000000000000000000000000110010100111001000010011010110010
000000000000000000000000000000000000000000000000000000000000000100001100101010111110111010111
000000000000000000000000000000000000000000000000000000000000000000101000011010011000111100
000000000000000000000000000000000000000000000000000000000000000000000101100000101111 0111
000000000000000000000000000000000000000000000000000000000000000000000000101011100111 0
000000000000000000000000000000000000000000000000000000000000000000000000000010011100 1
000000000000000000000000000000000000000000000000000000000000000000000000000000000000111
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Probability Matrix : Column-wise Optimization

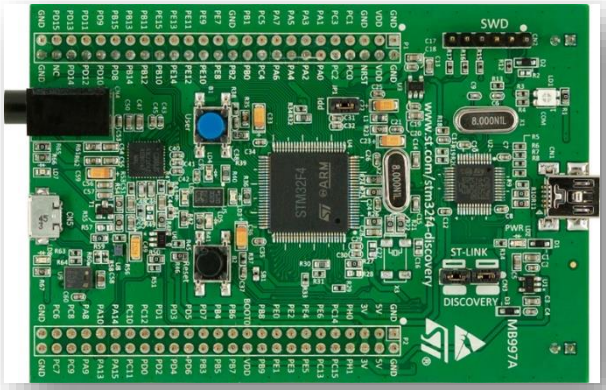
- Observation
  - Difference in length is 1 for most consecutive columns

- One-step difference in column length
  - One bit per differential column-length
    - 1 for increment
    - 0 for no-increment

```
000111111111010111000101110101
001111001101110110011011001101
001101001000110011101100011010
001010010010001110000011001110
000111010011001101100110100000
000100101100101100100011010010
000010101111011110010010001110
000001011100110110001001011000
000000101100100010110010101101
000000010011011000000110100010
000000000111101001000111111011
000000000010101110111011001001
000000000000011100010111000110 0
000000000000001000010101101010 1
000000000000000100011100100010
000000000000000010001001100 01
000000000000000000001111000100
000000000000000000000010111111
```

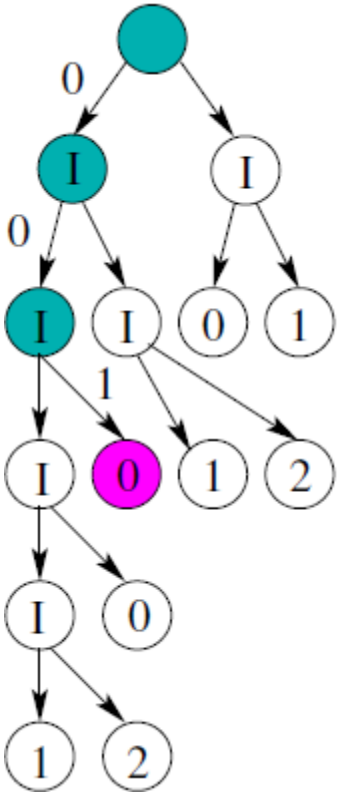
Easy to implement in both HW and SW

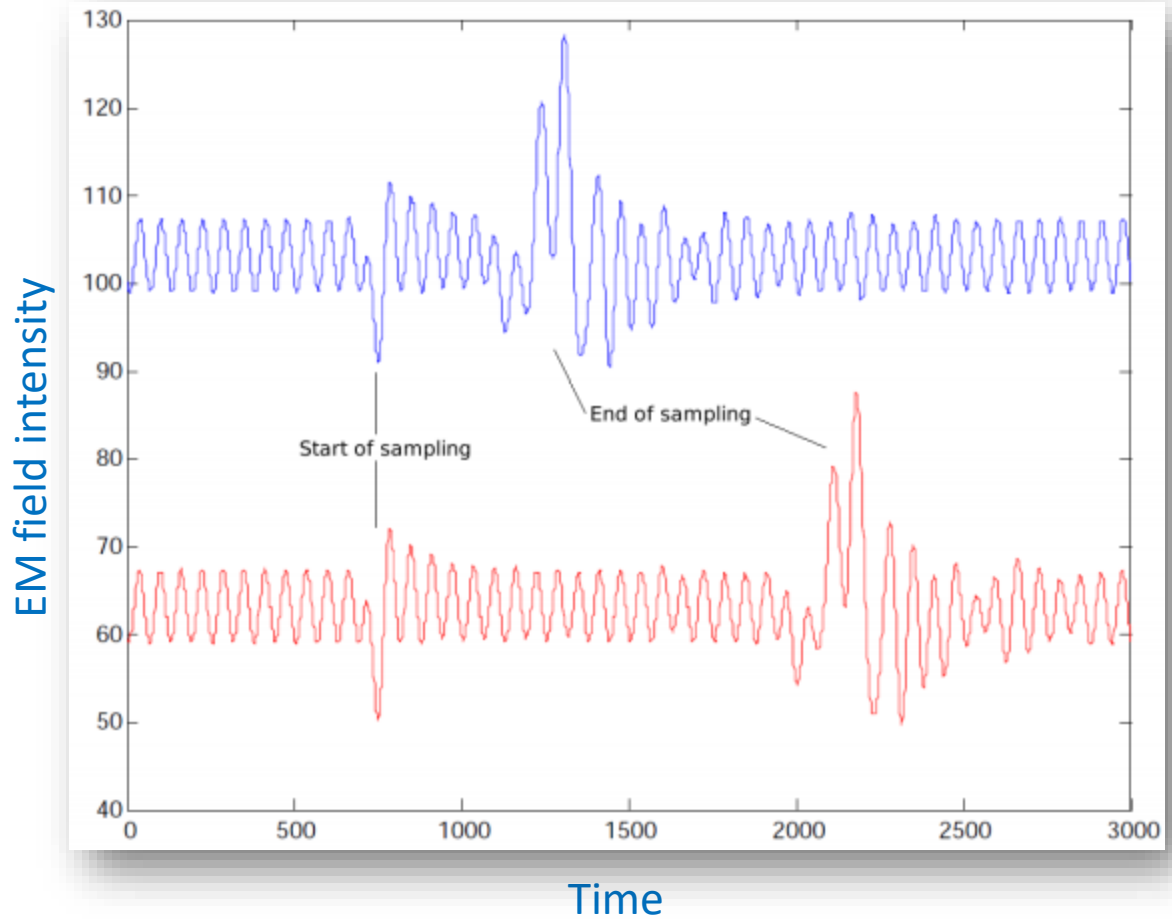
Resources: 1% of smallest FPGA
Cycles per sample (avg.) ~1.5



Memory 0.7 KB
Cycles per sample (avg.) ~28

Variation in execution time



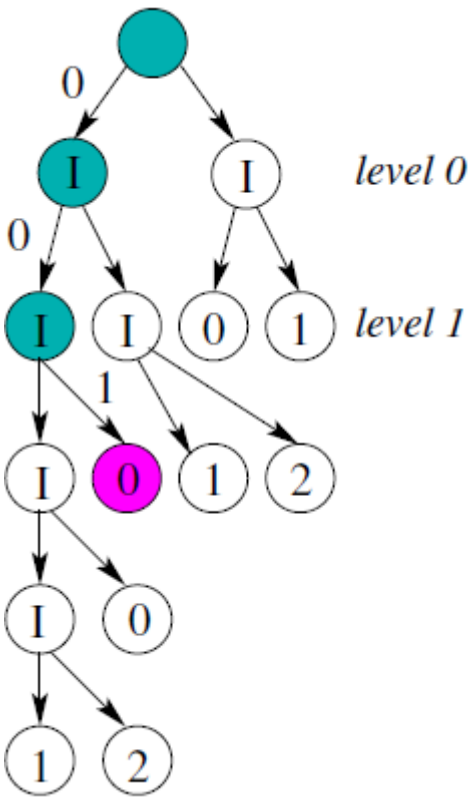


Ideal world: Hard to solve s when **b** = **A**·**s** + **e**

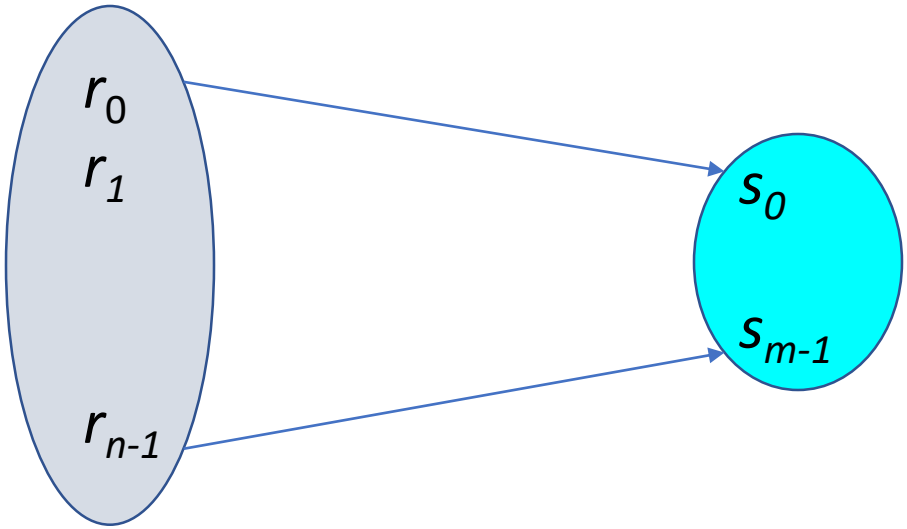
Timing leak → Easy to solve s when **b** = **A**·**s** + **e**

Constant-time **Discrete Gaussian sampling**

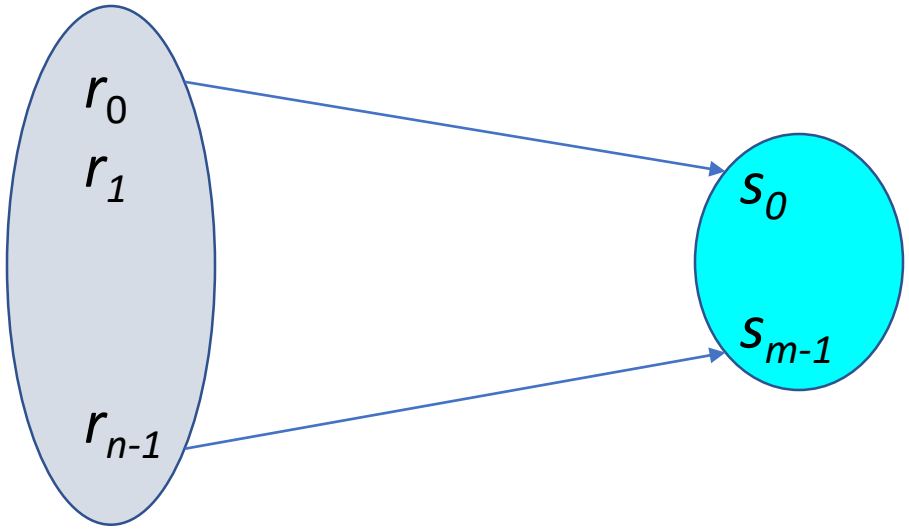
Knuth-Yao random walk: as a mapping



Random string	→	Sample value
001	→	0
010	→	1
011	→	2
...		



Mapping → Boolean equations



$$s_0 = f^0(r_0, r_1, \ldots, r_{n-1})$$
$$\ldots$$
$$s_{m-1} = f^{m-1}(r_0, r_1, \ldots, r_{n-1})$$

Needs to be evaluated in constant time

Constant-time evaluation

E.g. $s_0 = r_0 \wedge (r_1 \vee r_2)$.

Result is 0 when $r_0 = 0$ → Non-constant time

Constant-time evaluation

E.g. $s_0 = r_0 \wedge (r_1 \vee r_2)$.

Result is 0 when $r_0 = 0$ → Non-constant time

Bit-slicing

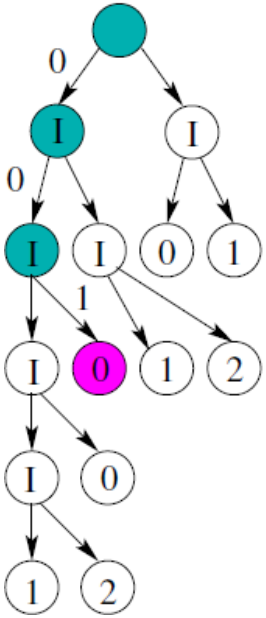
- constant time execution and
- performance

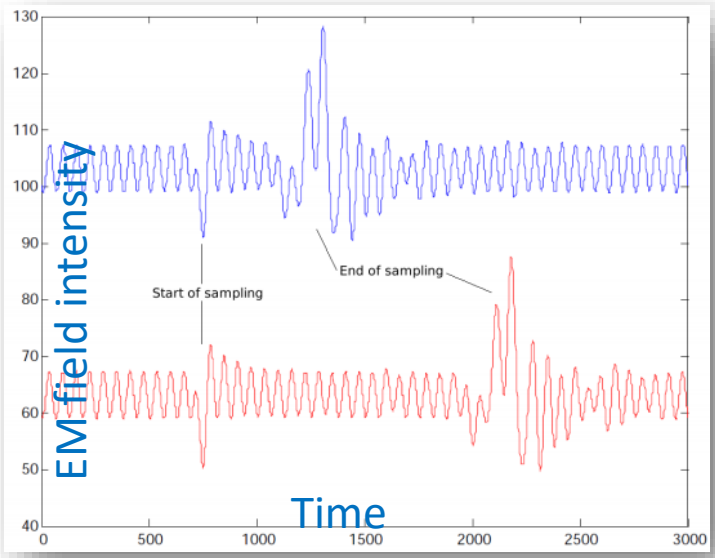
- 64-bit computer architecture
- 64-bit random integers R_0, R_1, and R_2

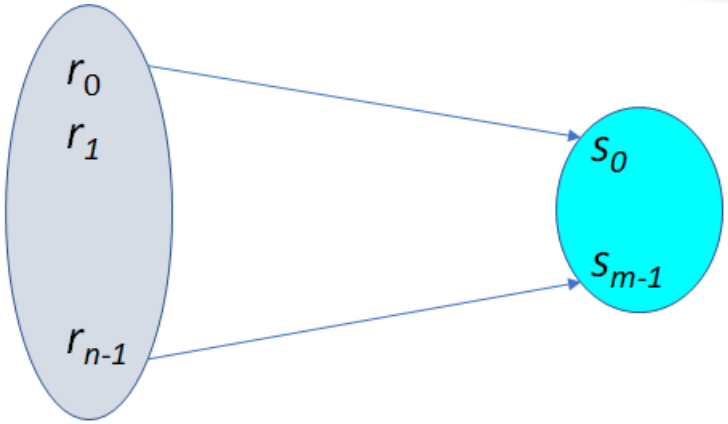
$$S_0 = R_0 \wedge (R_1 \vee R_2).$$

Parallel computation of 64 output bits in constant-time

Constant-time Gaussian Sampling





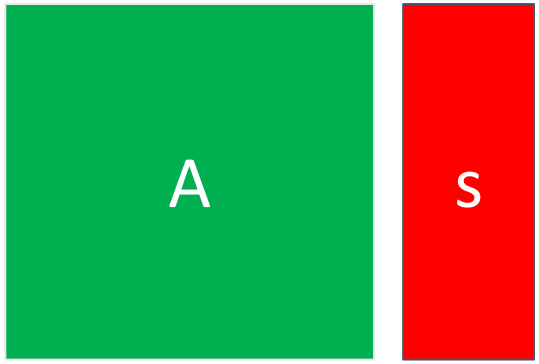


$$s_0 = f^0(r_0, r_1, \ldots, r_{n-1})$$
$$\ldots$$
$$s_{m-1} = f^{m-1}(r_0, r_1, \ldots, r_{n-1})$$

Convolution + Boolean Decomp + Bitslicing

Followup → Const. time *Falcon* signature becomes 33% slower

- Error sampling ✓
- **Matrix-vector multiplication**



Complexity $O(n^2)$

Performance of FRODO KEM

- FRODO KEM uses matrix dimension
  - n=640 for 100 bit quantum security
  - n=976 for 150 bit quantum security

FRODO KEM (n=640) on ARM Cortex M4

Key gen	Encapsulation	Decapsulation
81 M	86 M	87 M

- Roughly 3.3 sec at 24 MHz
- Slow due to expensive matrix-vector multiplications

Standard LWE

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} * \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{pmatrix} \approx \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \pmod q$$

Uniformly random matrix

Special LWE

$$\begin{pmatrix} a_0 & -a_3 & -a_2 & -a_1 \\ a_1 & a_0 & -a_3 & -a_2 \\ a_2 & a_1 & a_0 & -a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} * \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{pmatrix} \approx \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \pmod q$$

Matrix from uniformly random vector

Special LWE: known as Ring-LWE

$$\begin{pmatrix} a_0 & -a_3 & -a_2 & -a_1 \\ a_1 & a_0 & -a_3 & -a_2 \\ a_2 & a_1 & a_0 & -a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} * \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{pmatrix} \approx \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \pmod q$$

Matrix from uniformly random vector

$$a(x) * s(x) + e(x) \approx b(x) \pmod q \pmod{x^4 + 1}$$

where

$a(x) = (a_0 + a_1x + a_2x^2 + a_3x^3)$

$s(x) = (s_0 + s_1x + s_2x^2 + s_3x^3)$

$e(x) = (e_0 + e_1x + e_2x^2 + e_3x^3)$

$b(x) = (b_0 + b_1x + b_2x^2 + b_3x^3)$

Efficient Polynomial multiplication

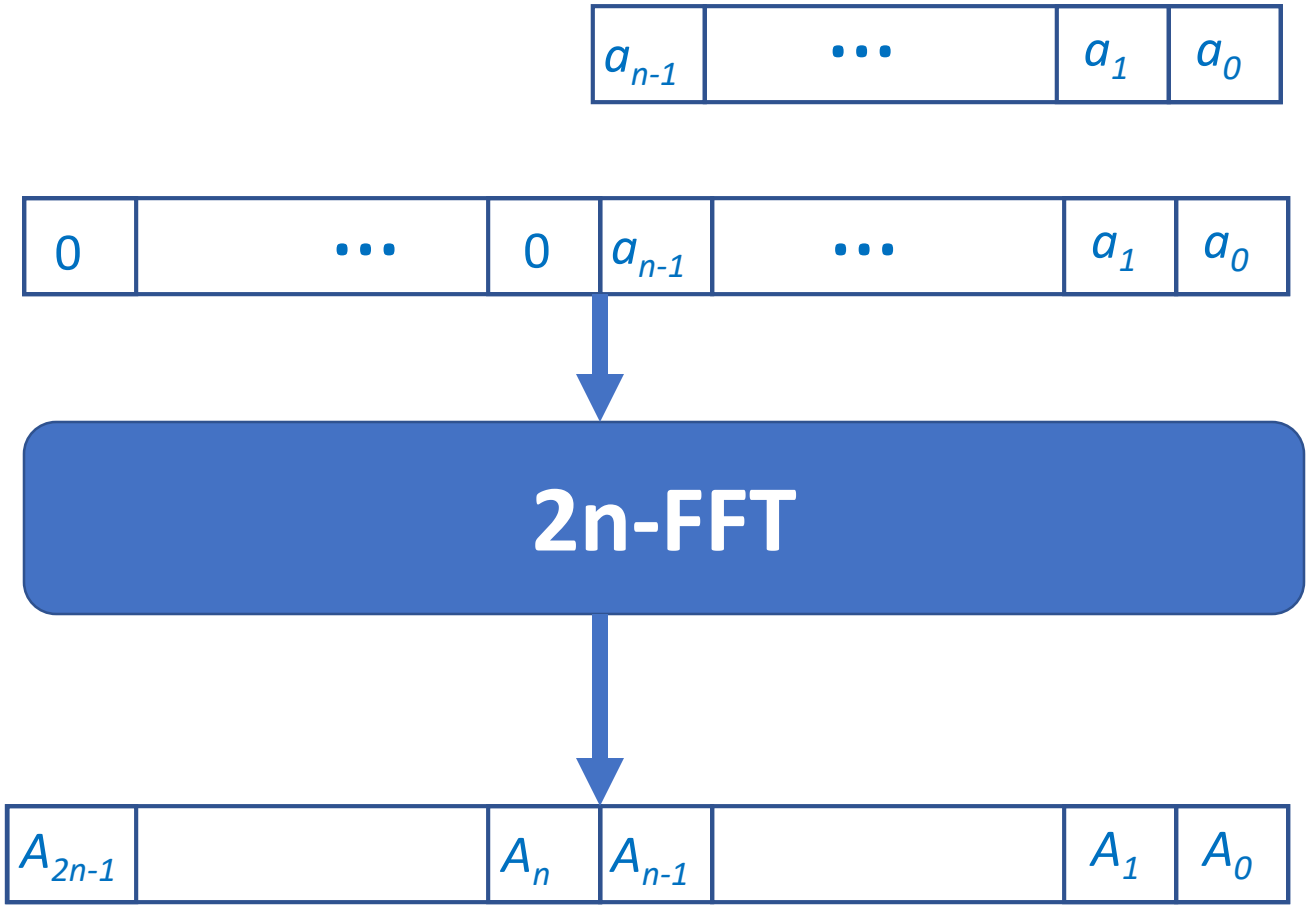
Polynomial multiplication: methods

- Schoolbook multiplication: complexity n^2
- Karatsuba multiplication: complexity $n^{1.585}$
- FFT-based multiplication: **complexity (*n log n*)**

FFT = Fast Fourier Transform

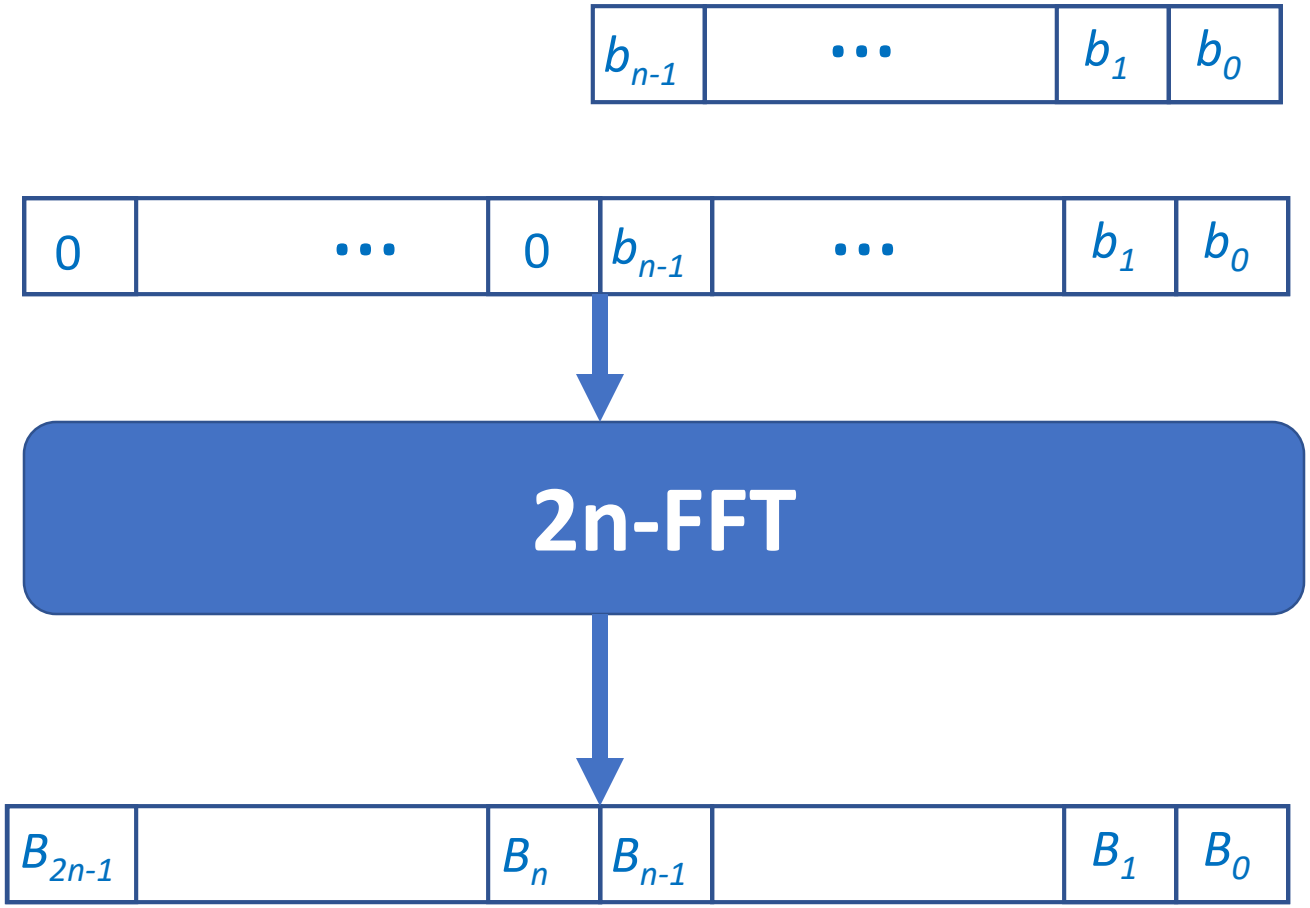
Polynomial multiplication: FFT-based

$a(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$

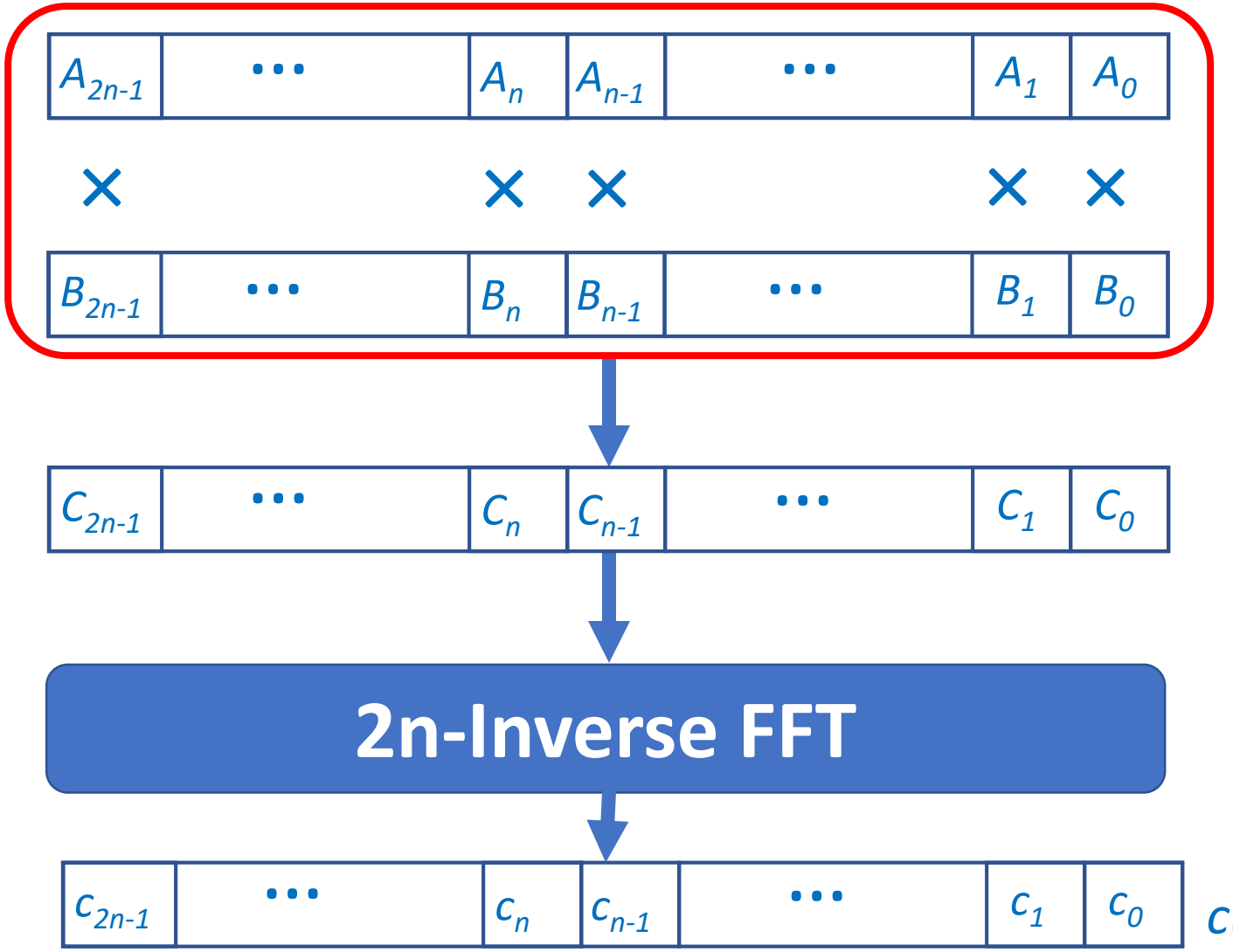


Polynomial multiplication: FFT-based

$b(x) = b_{n-1}x^{n-1} + \ldots + b_1 x + b_0$



Polynomial multiplication: FFT-based



$c(x)=a(x)\cdot b(x)$

FFT to NTT

- FFT involves real numbers
- Number Theoretic Transform (**NTT**)
  - is a generalization of FFT
  - Only integer arithmetic modulo q

Requirements: $\mathbb{Z}_q[x]/(x^n + 1)$

- $n = 2^k$
- Prime q with $q \equiv 1 \bmod n$

FFT to NTT

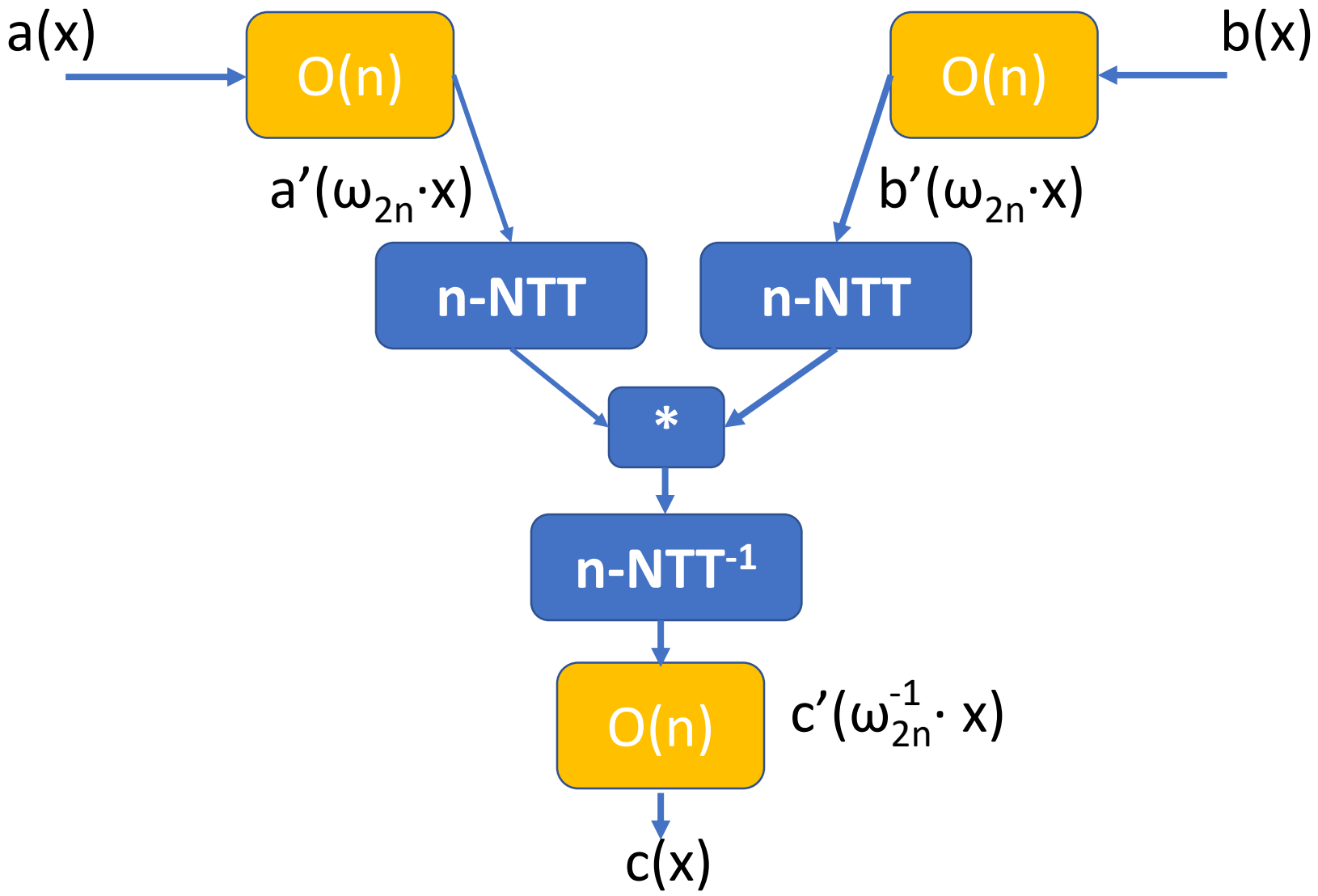
- FFT involves real numbers
- Number Theoretic Transform (**NTT**)
  - is a generalization of FFT
  - Only integer arithmetic modulo q

Requirements: $\mathbb{Z}_q[x]/(x^n + 1)$

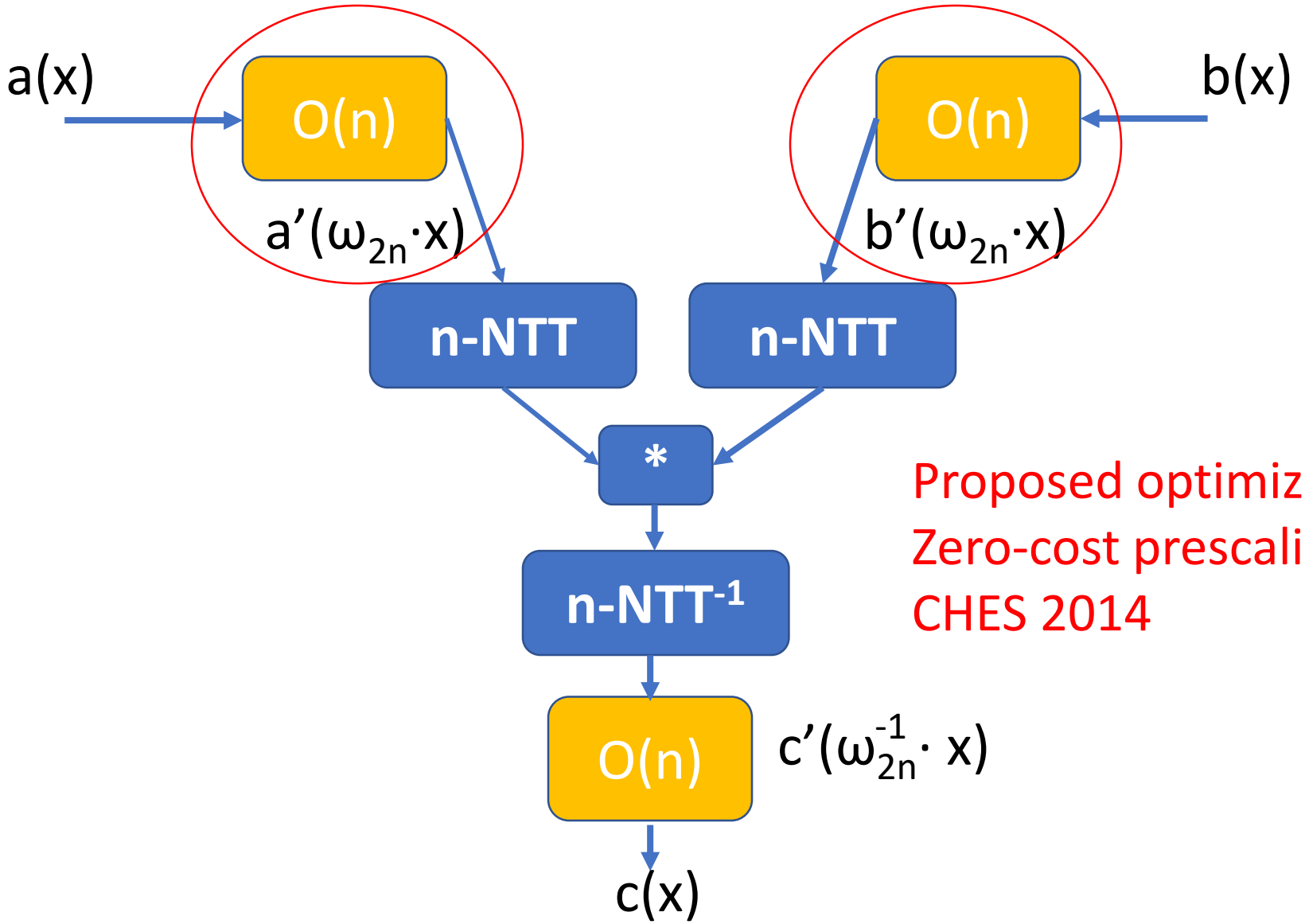
- $n = 2^k$
- Prime q with $q \equiv 1 \bmod 2n$

'Negative-wrapped convolution'
Requires n-NTT
instead of 2n-NTT

Special optimization: scaling overhead

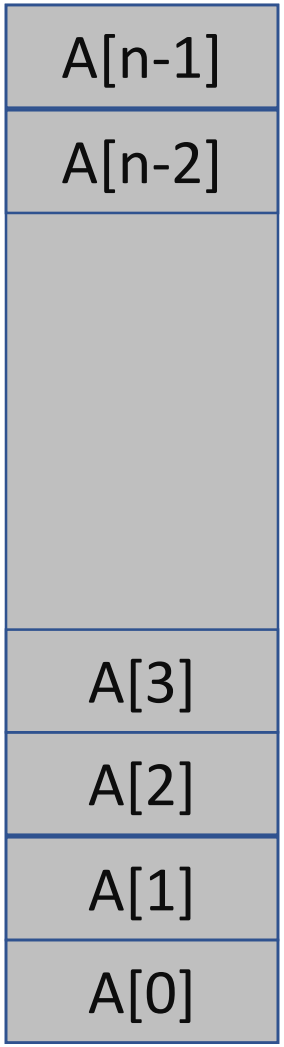


Special optimization: scaling overhead



Proposed optimization:
Zero-cost prescaling
CHES 2014

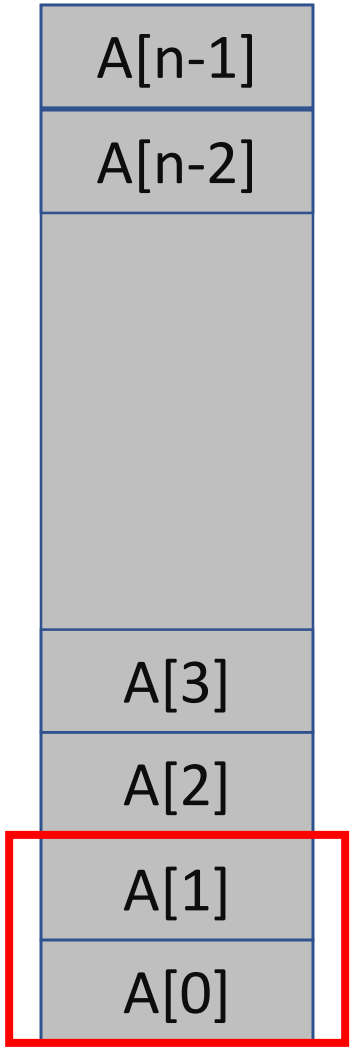
Memory



Simplified NTT loops

```
for(m=2; m<=n; m=2m)
{
  for(j=0; j<=m/2-1; j++)
  {
    for(k=0; j<n; k=k+m)
    {
      index = f(m, j, k);
      Butterfly(A[index],A[index+m/2]);
    }
  }
}
```

Memory



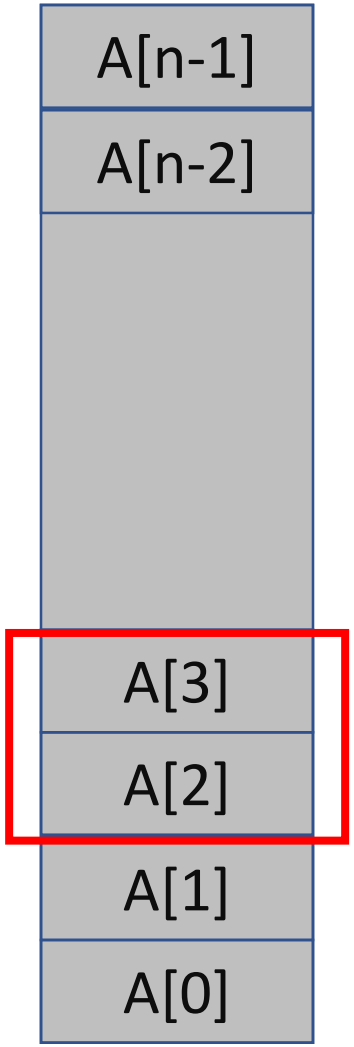
Simplified NTT loops

```
for(m=2; m<=n; m=2m)
{
  for(j=0; j<=m/2-1; j++)
  {
    for(k=0; j<n; k=k+m)
    {
      index = f(m, j, k);
      Butterfly(A[index],A[index+m/2]);
    }
  }
}
```

NTT starts with m=2

Butterfly(A[0], A[1])

Memory



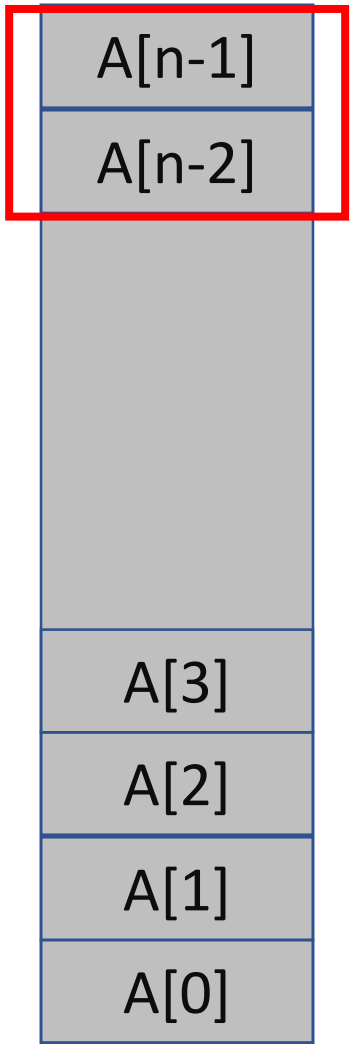
Simplified NTT loops

```
for(m=2; m<=n; m=2m)
{
  for(j=0; j<=m/2-1; j++)
  {
    for(k=0; j<n; k=k+m)
    {
      index = f(m, j, k);
      Butterfly(A[index],A[index+m/2]);
    }
  }
}
```

NTT starts with m=2

Butterfly(A[2], A[3])

Memory



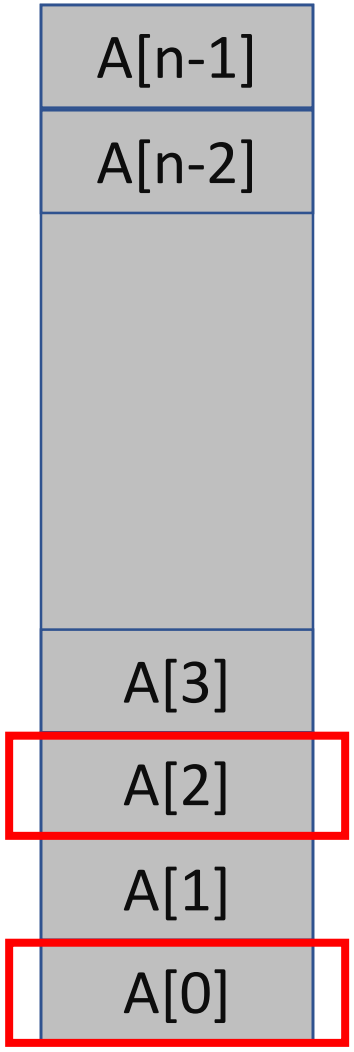
Simplified NTT loops

```
for(m=2; m<=n; m=2m)
{
  for(j=0; j<=m/2-1; j++)
  {
    for(k=0; j<n; k=k+m)
    {
      index = f(m, j, k);
      Butterfly(A[index],A[index+m/2]);
    }
  }
}
```

NTT starts with m=2

Butterfly(A[n-2], A[n-1])

Memory

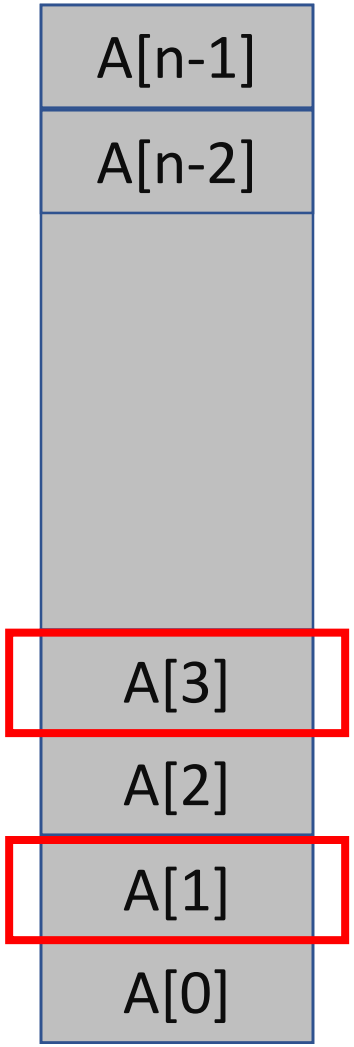


Simplified NTT loops

```
for(m=2; m<=n; m=2m)
{
  for(j=0; j<=m/2-1; j++)
  {
    for(k=0; j<n; k=k+m)
    {
      index = f(m, j, k);
      Butterfly(A[index],A[index+m/2]);
    }
  }
}
```

Next, m increments to m=4.
Butterfly(A[0], A[2]), Butterfly(A[4], A[6]) …

Memory



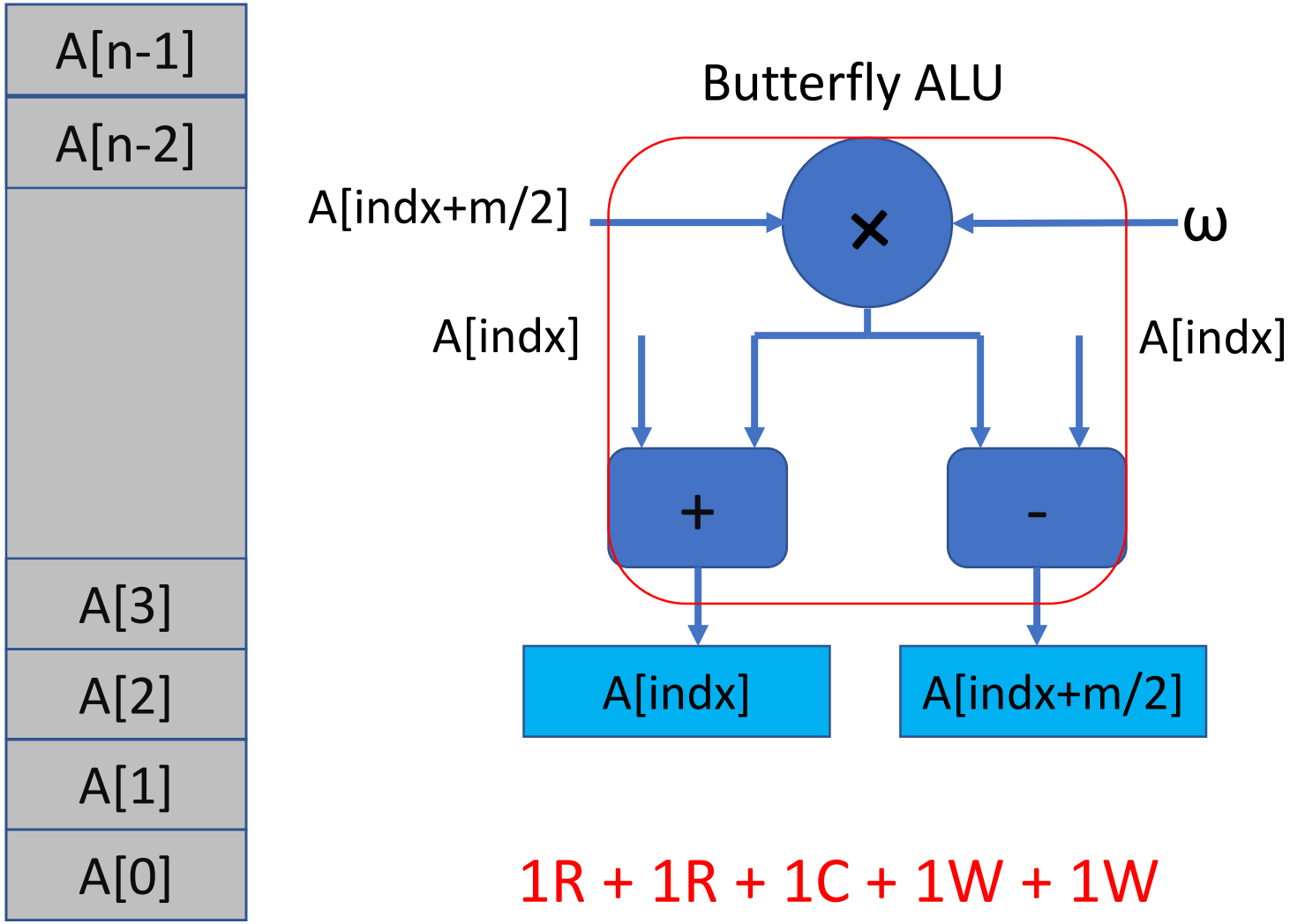
Simplified NTT loops

```
for(m=2; m<=n; m=2m)
{
  for(j=0; j<=m/2-1; j++)
  {
    for(k=0; j<n; k=k+m)
    {
      index = f(m, j, k);
      Butterfly(A[index],A[index+m/2]);
    }
  }
}
```

Next, m increments to m=4.
Butterfly(A[1], A[3]), Butterfly(A[5], A[7]) …

Memory access problem

Memory is sequential

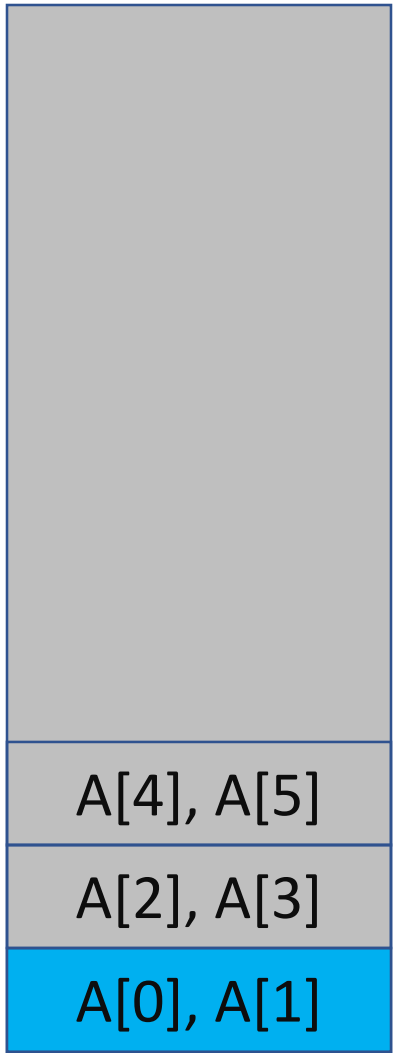


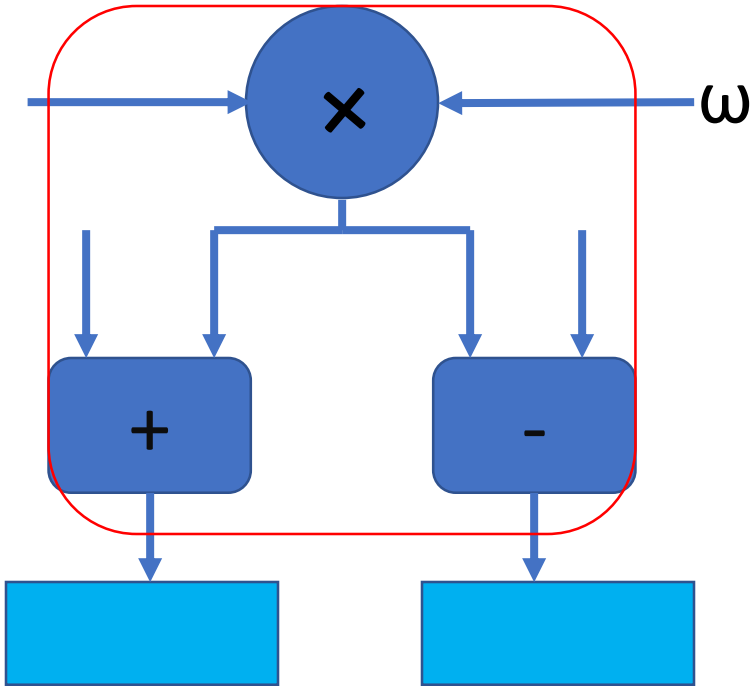
1R + 1R + 1C + 1W + 1W

A[4], A[5]
A[2], A[3]
A[0], A[1]

- If we have small coefficients
- Two coefficients in a single word?

Observation

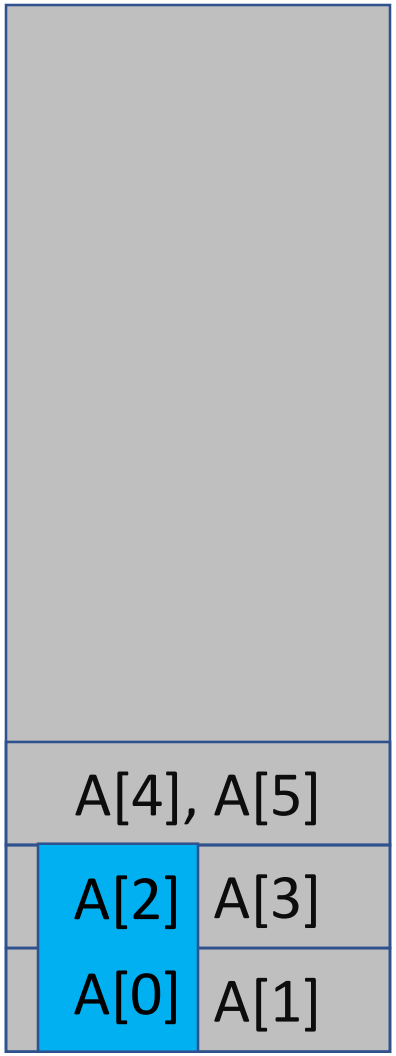


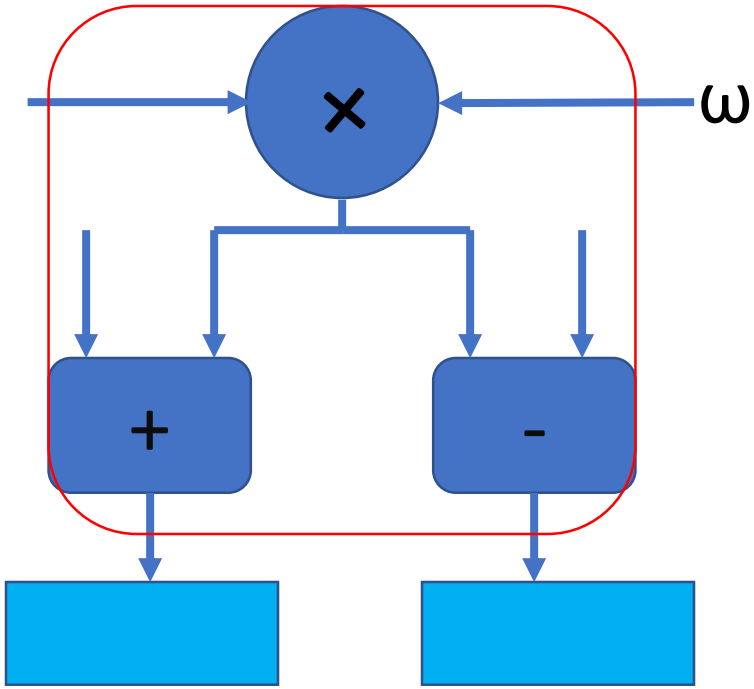


During NTT loop m = 2

1. Read {A[0], A[1]} [1 Cycle]
2. Butterfly(A[0], A[1]) [1 Cycle]
3. Write {A[0], A[1]} [1 Cycle]

Observation



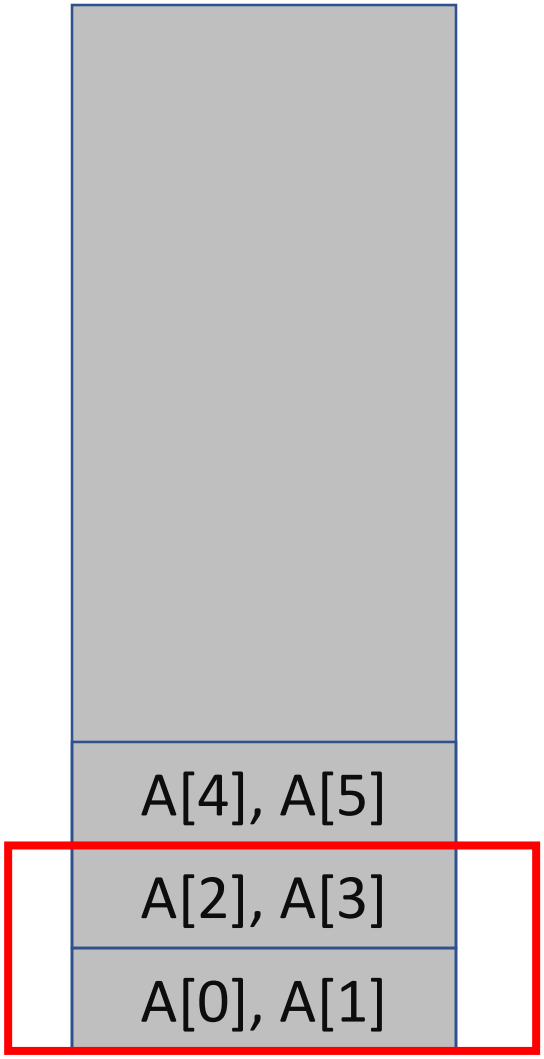


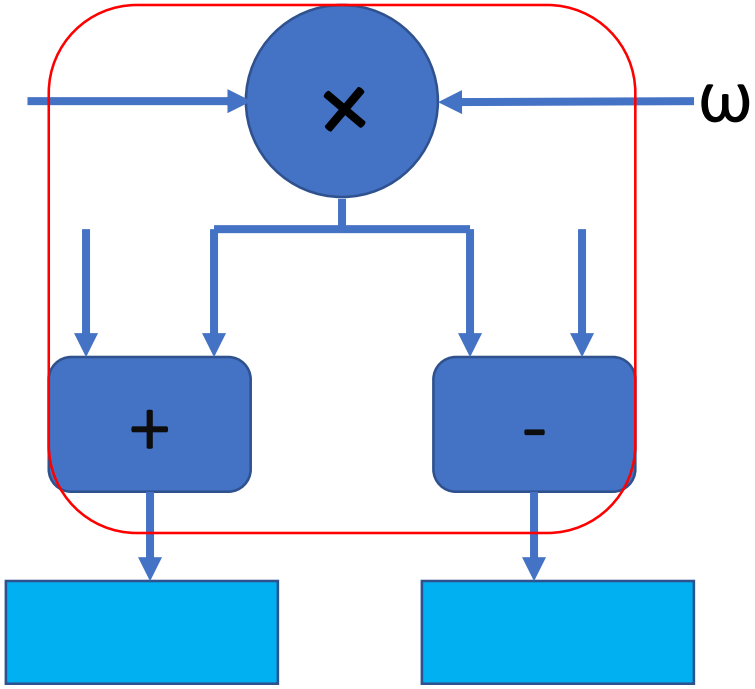
Problem happens next, when m=4.

Butterfly(A[0], A[2]), Butterfly(A[4], A[6]) …

Solution

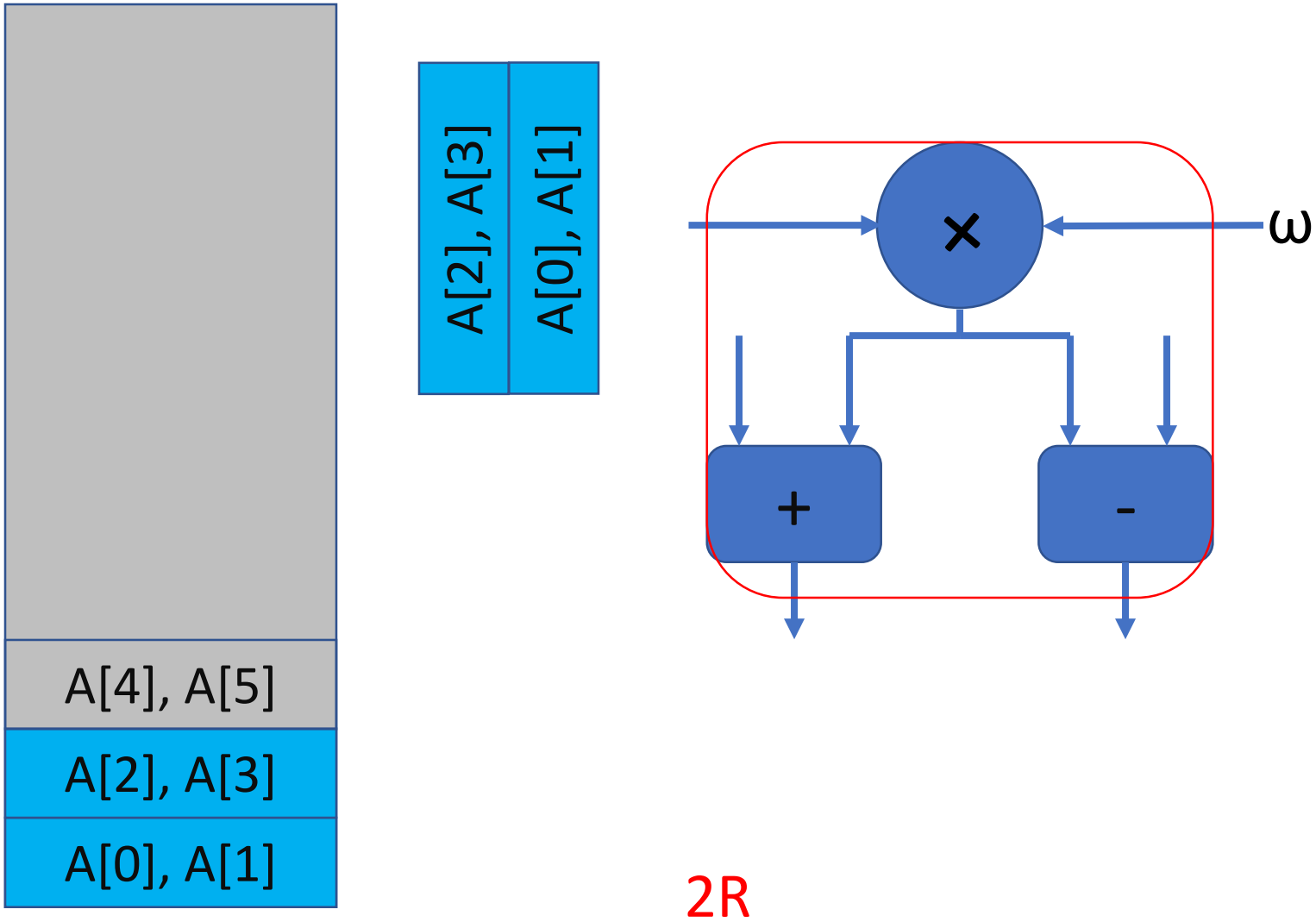
Process 4 coefficients together





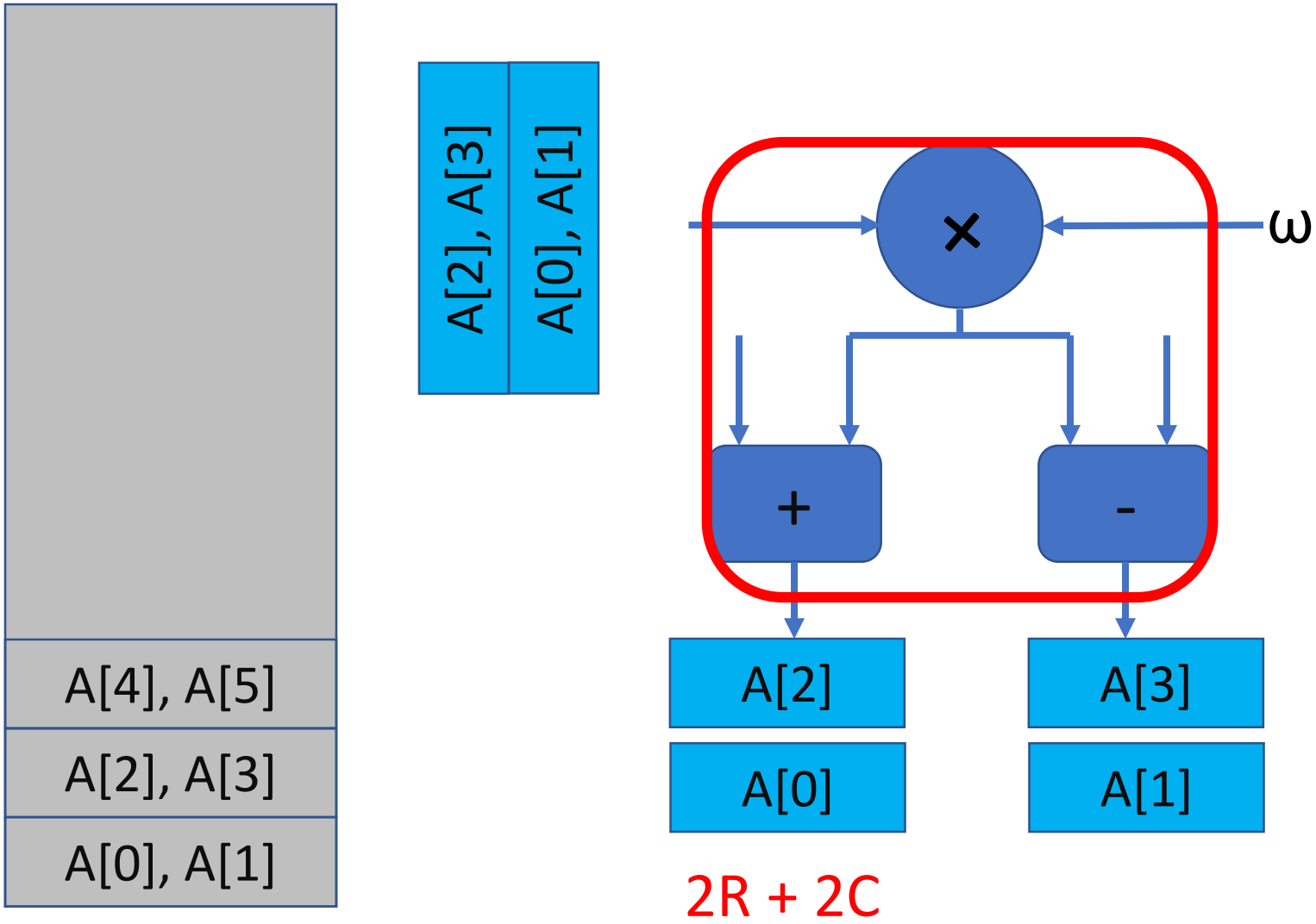
Solution

Process 4 coefficients together



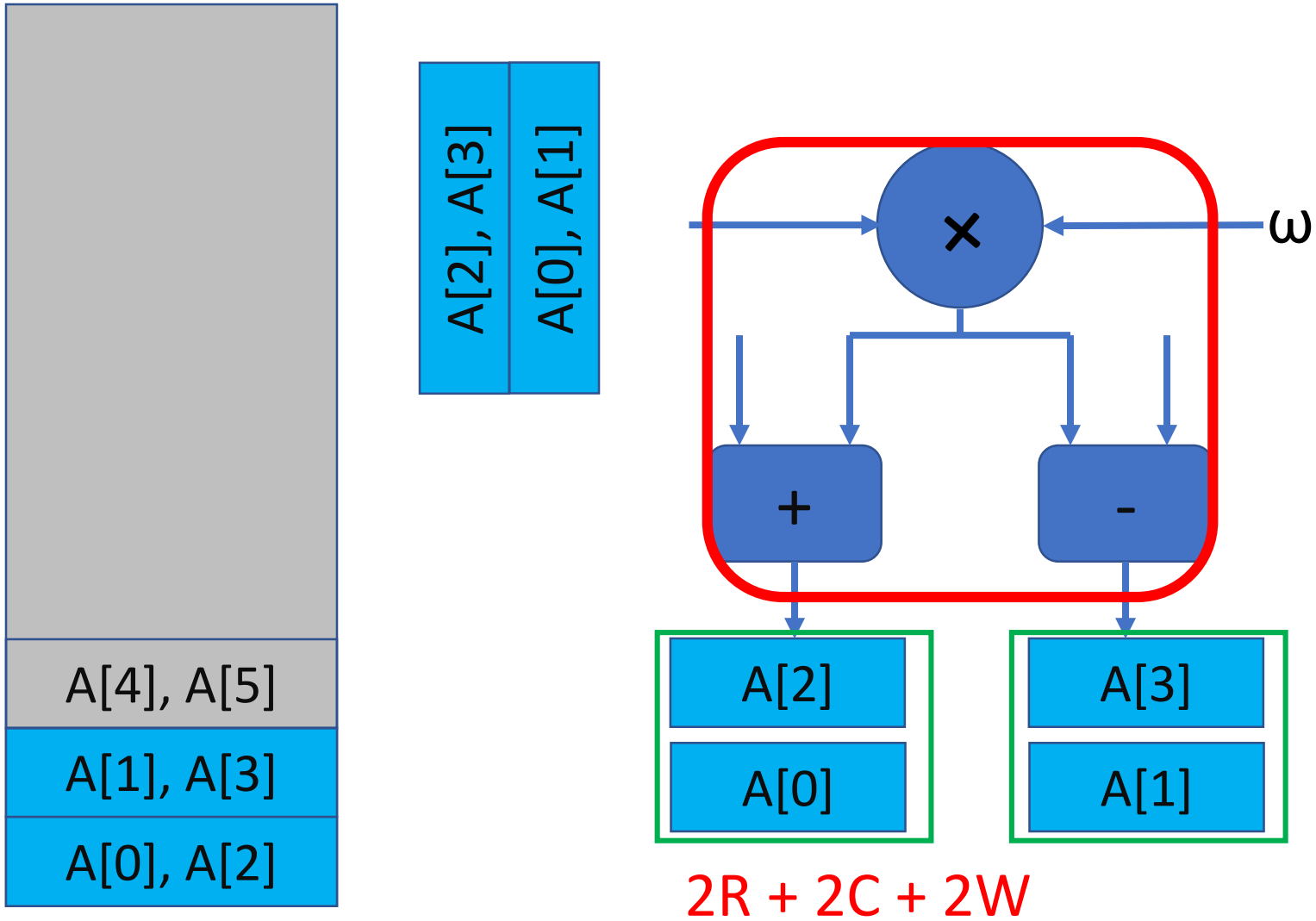
Solution

Process 4 coefficients together



Solution

Process 4 coefficients together

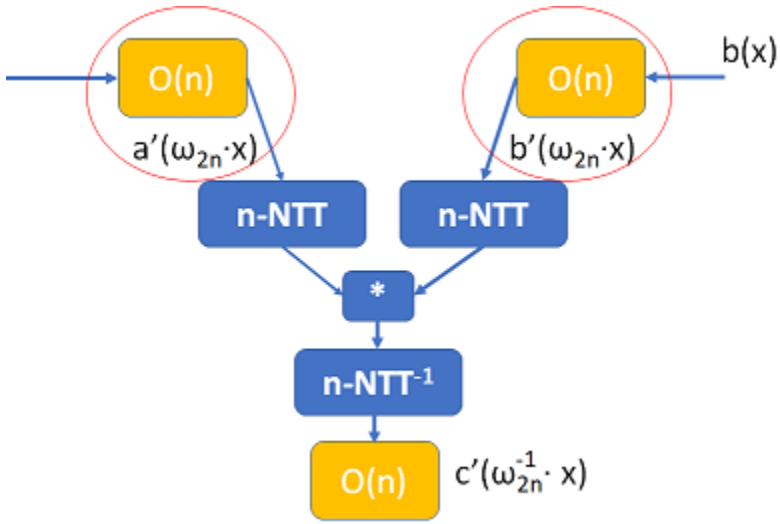


Results

Before $O(n) + O(n) + O(n \log n) + O(n)$

Results

Before O(n) + O(n) + O(n log n) + O(n)

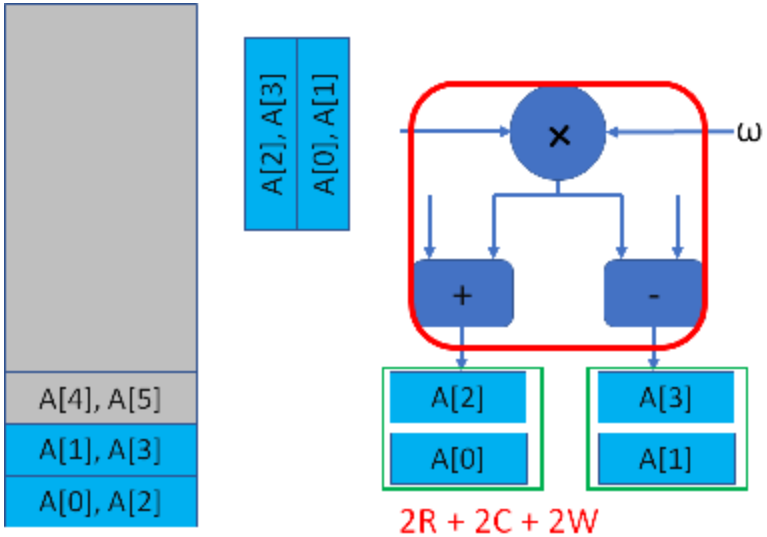


Now ~~O(n)~~ + ~~O(n)~~ + O(n log n) + O(n)

Optimization 1
Zero-cost
prescaling

Results

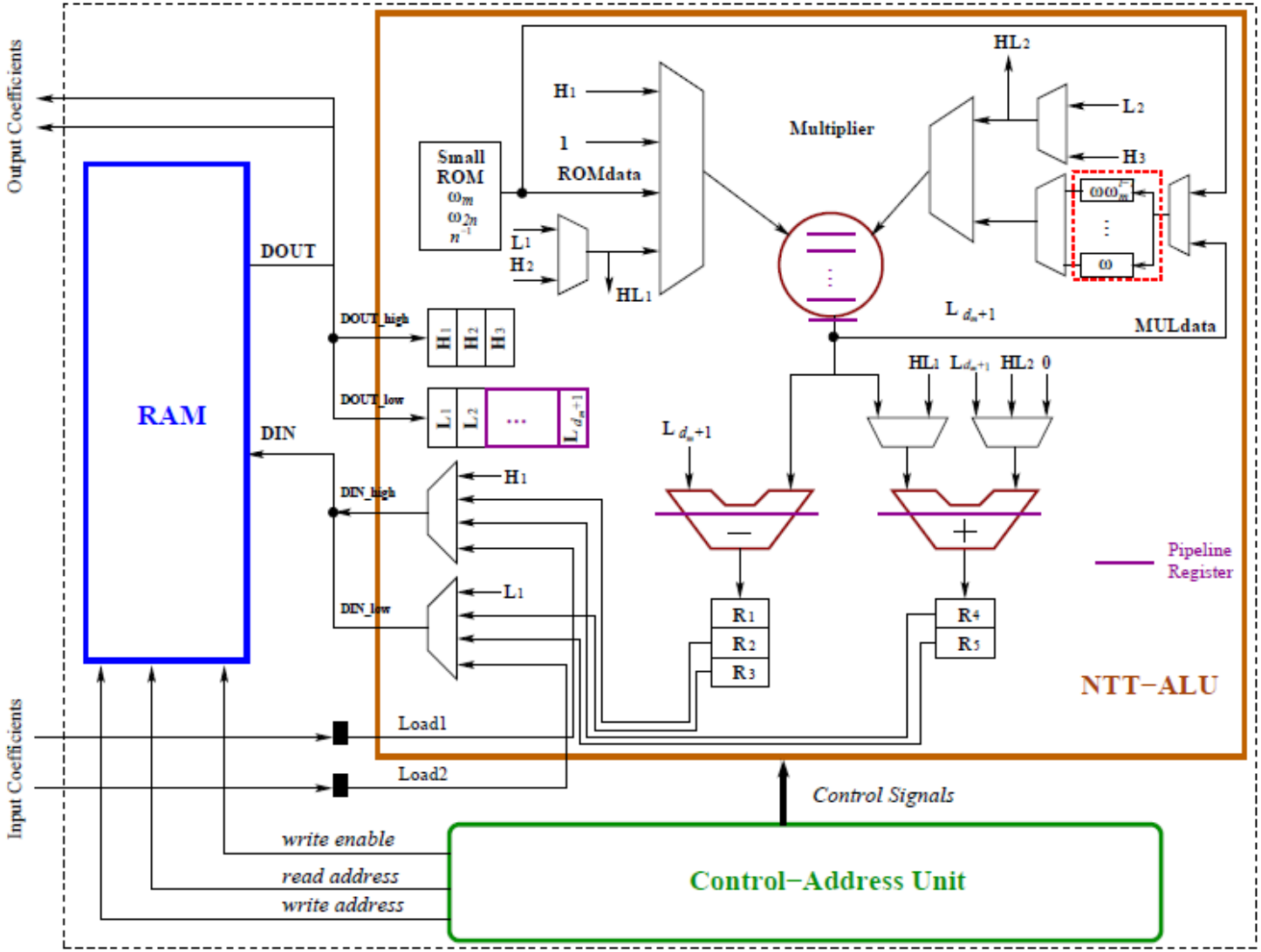
Before O(n) + O(n) + O(n log n) + O(n)



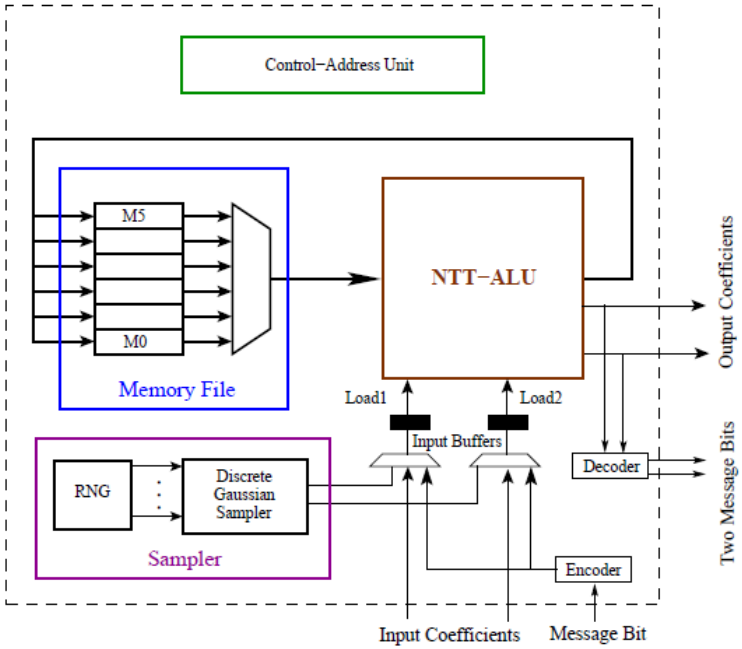
Optimization 2
Memory access reduction

Now ~~O(n)~~ + ~~O(n)~~ + ½O(n log n) + O(n)

Architecture of NTT-based polynomial multiplier



Lattice-based public-key instruction-set encryption processor



Instruction Set

1. LOAD
2. ENCODE-LOAD
3. GAUSSIAN-LOAD
4. FFT
5. INV-FFT
6. ADD
7. CMULT
8. REARRANGE
9. READ

Throughput:
50,000 encryptions/sec

100,000 decryptions/sec

Area: 1349 LUT, 860 FF,
1 DSPMULT, 2 BRAM18

(polynomial degree 256)

Publication: CHES 2014

Instruction-set ring-LWE cryptoprocessor

Throughput:
50,000 encryption/sec
100,000 decryption/sec

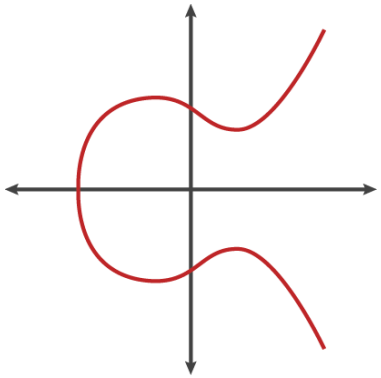
Area: 1.3K LUT, 860 FF,
1 DSPMULT, 2 BRAM18

CHES 2014

Throughput:
40,000 encryption/sec <
80,000 decryption/sec <

Area: 18349 LUT, 5644 FF

CHES 2012



ECC

Ring-LWE encryption: followup works

Software implementations

- On 32-bit ARM

R. de Clercq, S. Sinha Roy, F. Vercauteren, I.Verbauwhede, "Efficient software implementation of ring-LWE encryption", DATE 2015

Encryptions 121,166 cycles
Decryptions 43,324 cycles

Orders of magnitude faster than ECC

- On 8-bit AVR

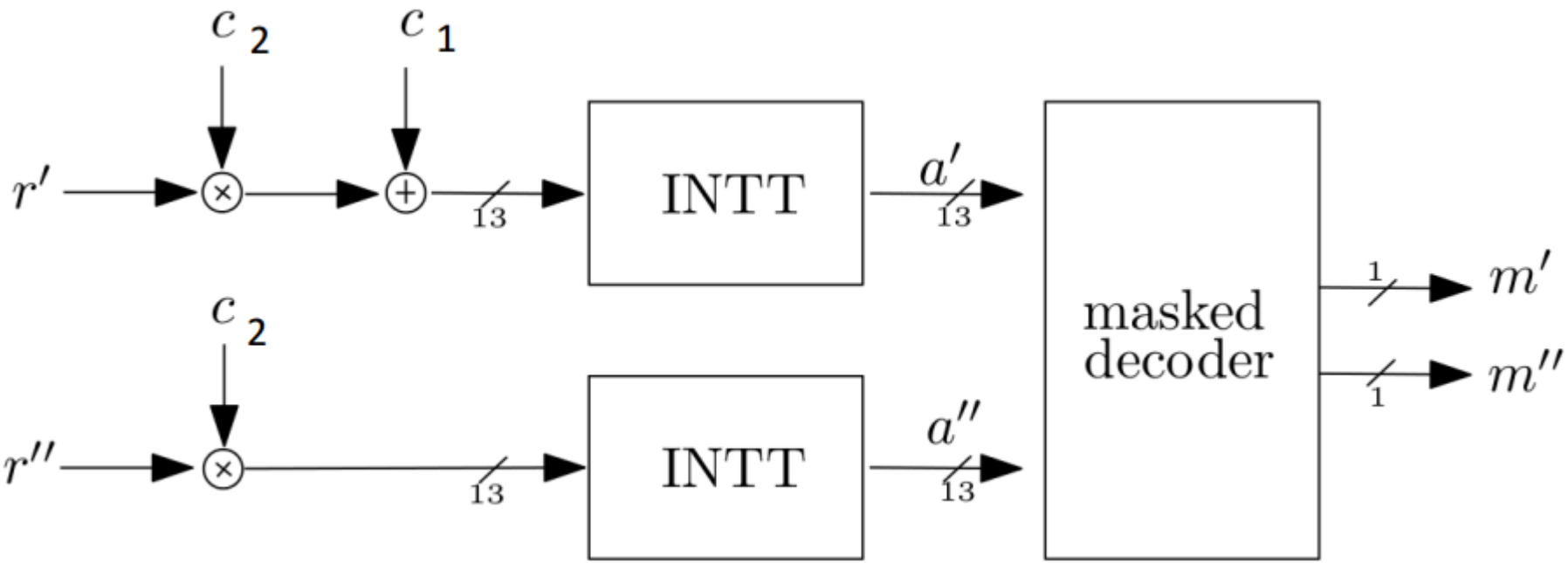
Z. Liu, H. Seo, S. Sinha Roy, J. Großschädl, H. Kim, I. Verbauwhede, "Efficient Ring-LWE Encryption on 8-Bit AVR Processors", CHES 2015

Encryptions 671,628 cycles
Decryptions 275,646 cycles

Ring-LWE encryption: followup works

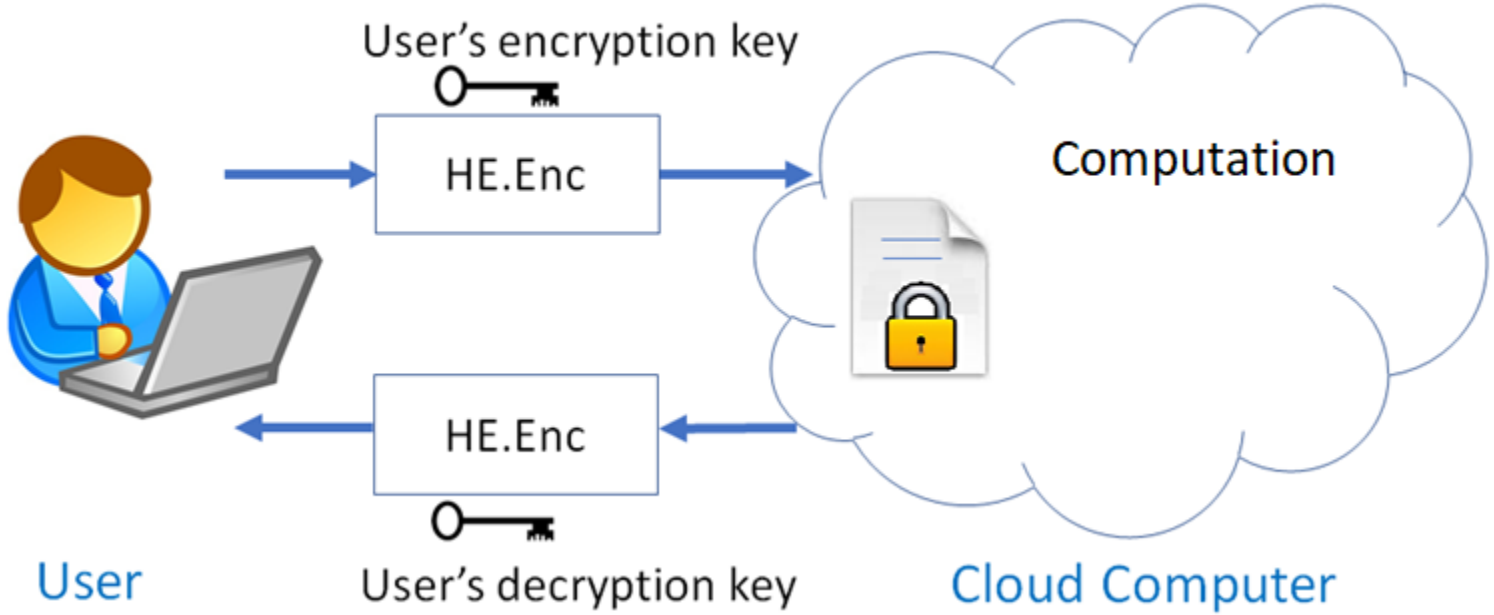
Side channel security: masking scheme

O. Reparaz, S. Sinha Roy, F. Vercauteren, I. Verbauwhede, "A masked ring-LWE implementation", in CHES 2015



Hardware accelerators for Homomorphic Computation

Homomorphic computation



Interesting applications :

- Machine learning on encrypted data
- Prediction from consumption data in smart electricity meters
- Health-care applications
- Encrypted web-search engine

Lattice-based Homomorphic encryption

1. Encrypt

public keys $(\boldsymbol{p_0}, \boldsymbol{p_1})$

2. Decrypt

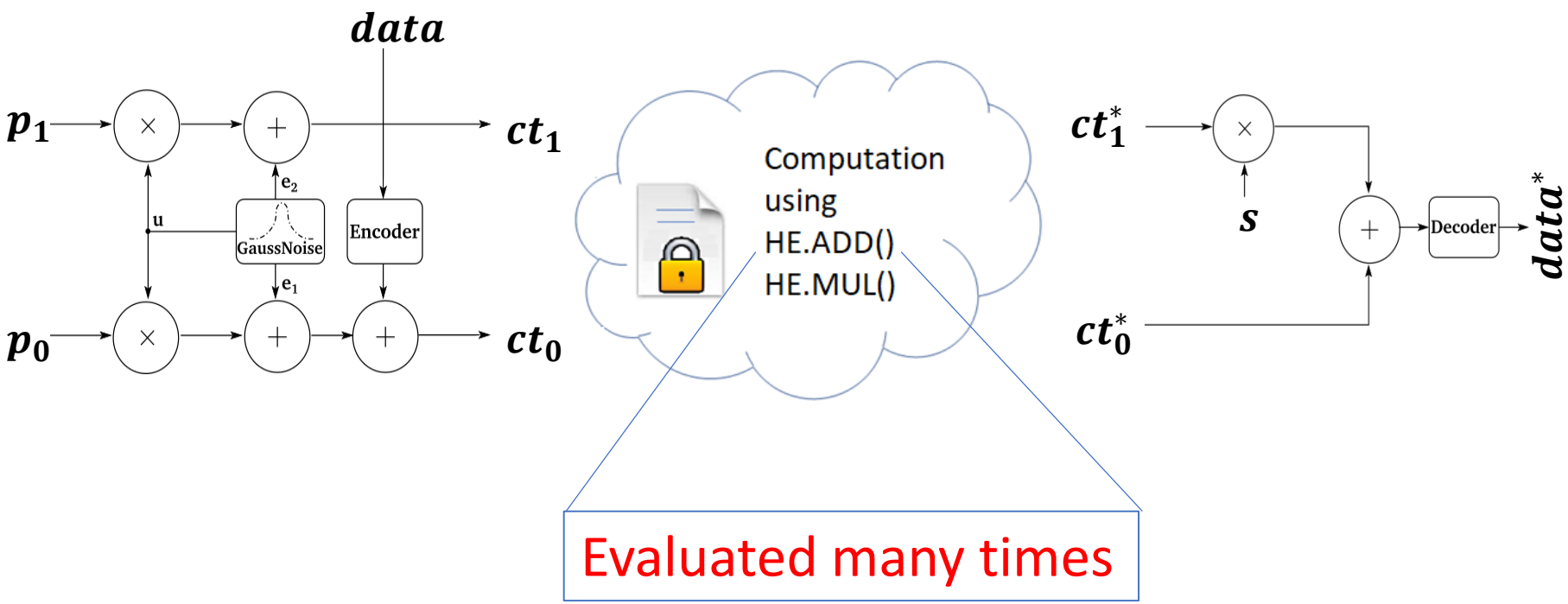
private key $\boldsymbol{s}$

Lattice-based Homomorphic encryption

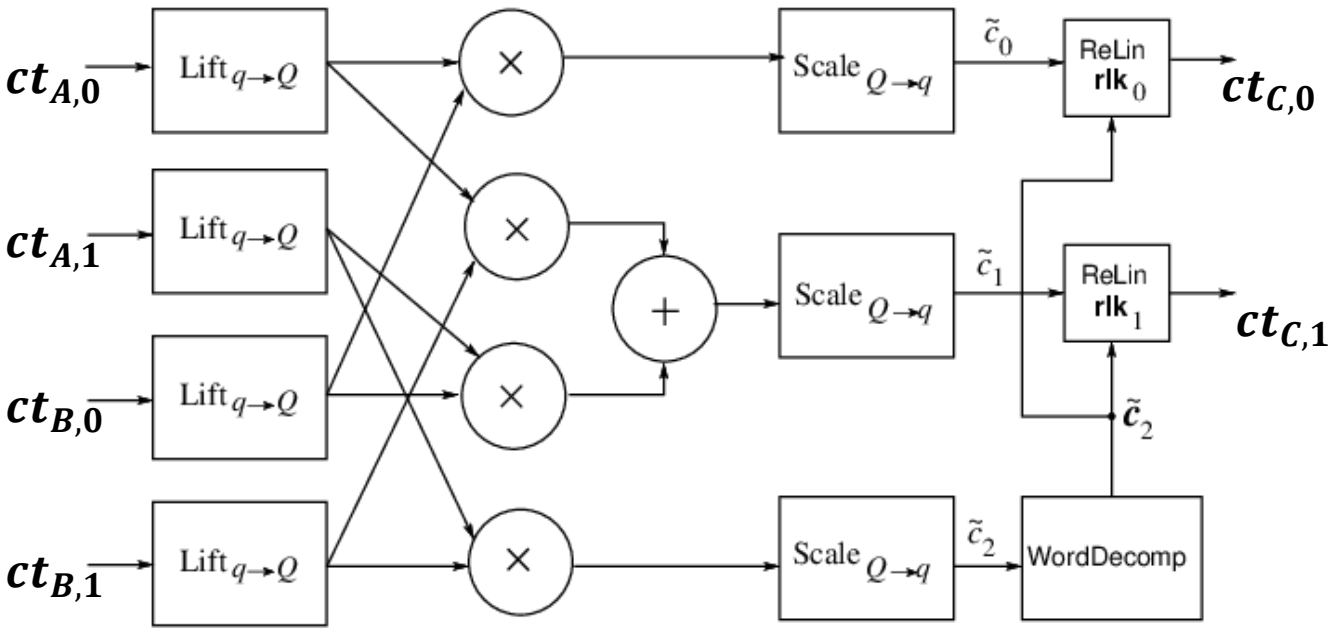
1. Encrypt locally
public keys $(\boldsymbol{p_0}, \boldsymbol{p_1})$

2. Process on cloud

3. Decrypt locally
private key $\boldsymbol{s}$



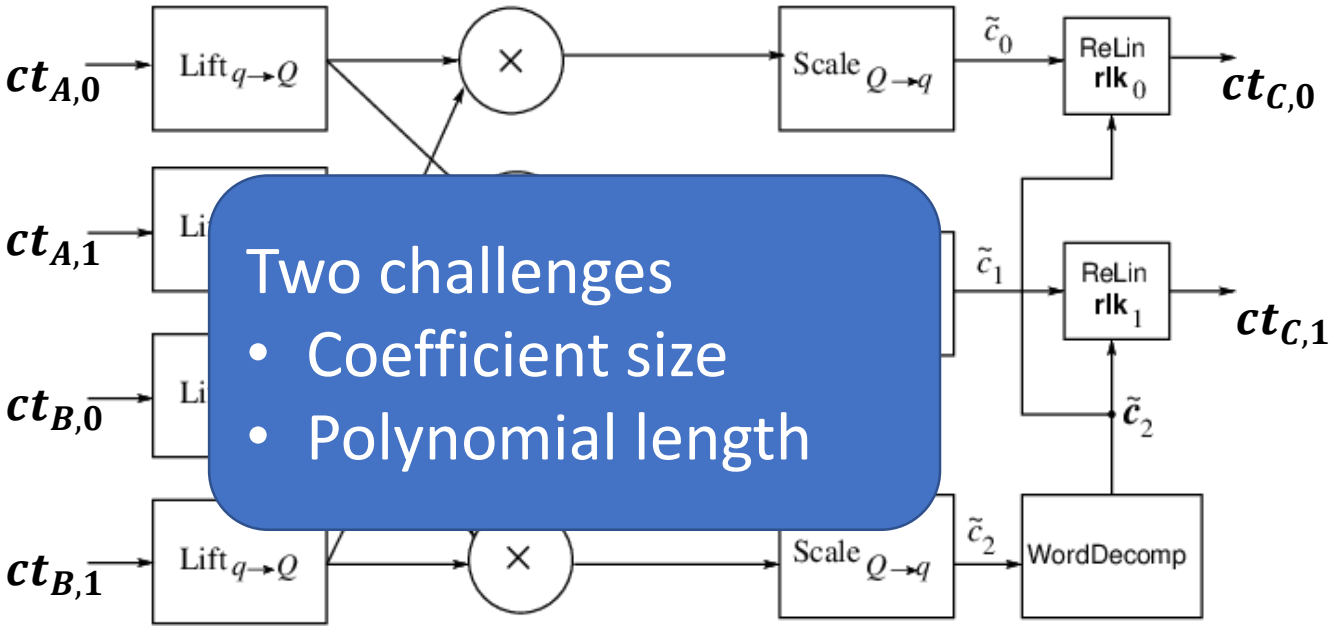
Homomorphic Multiplication



Uses multiple computation blocks:

- Lift
- Polynomial multiplication
- Scale

Homomorphic Multiplication. How complex?



Two challenges
- Coefficient size
- Polynomial length

Low complexity applications:
- Polynomials have 4,000 coeffs.
- Coeffs are ~180 bit wide

M**edium** complexity applications:
- Polynomials have 32,000 coeffs.
- Coeffs are ~1200 bit wide

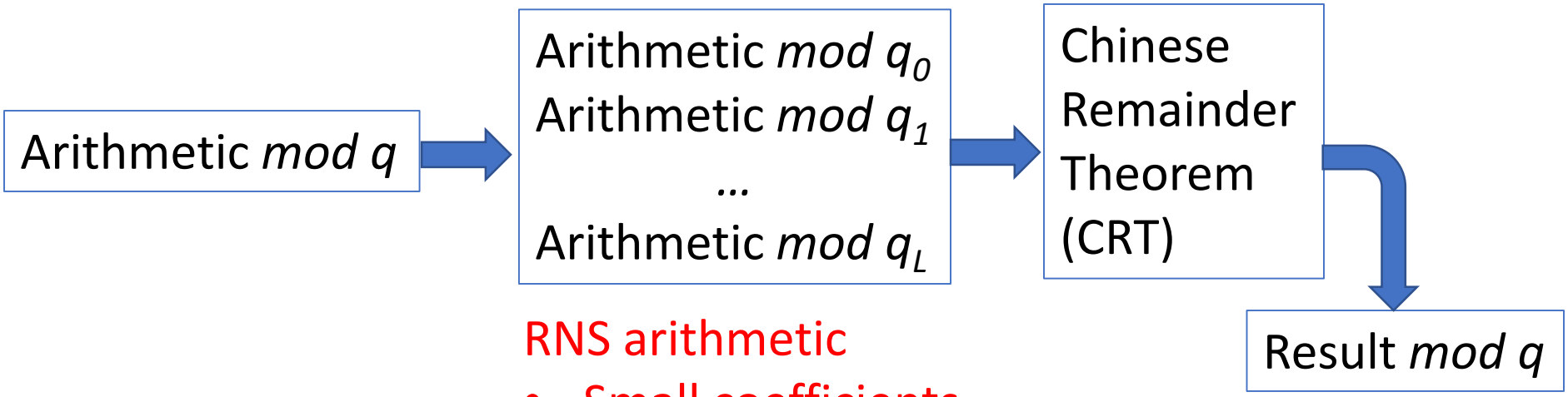
Lattice-based key exchange schemes
- Polynomials with 256 or 512 coeffs
- Coeff size ~10 bits

Hardware accelerators for Homomorphic Computation

→Arithmetic of large coefficients

Application of Residue Number System

- We need to compute arithmetic modulo q
- Let $q = \prod q_i$ where q_i are coprime
- Then we can work with Residue Number System (RNS)

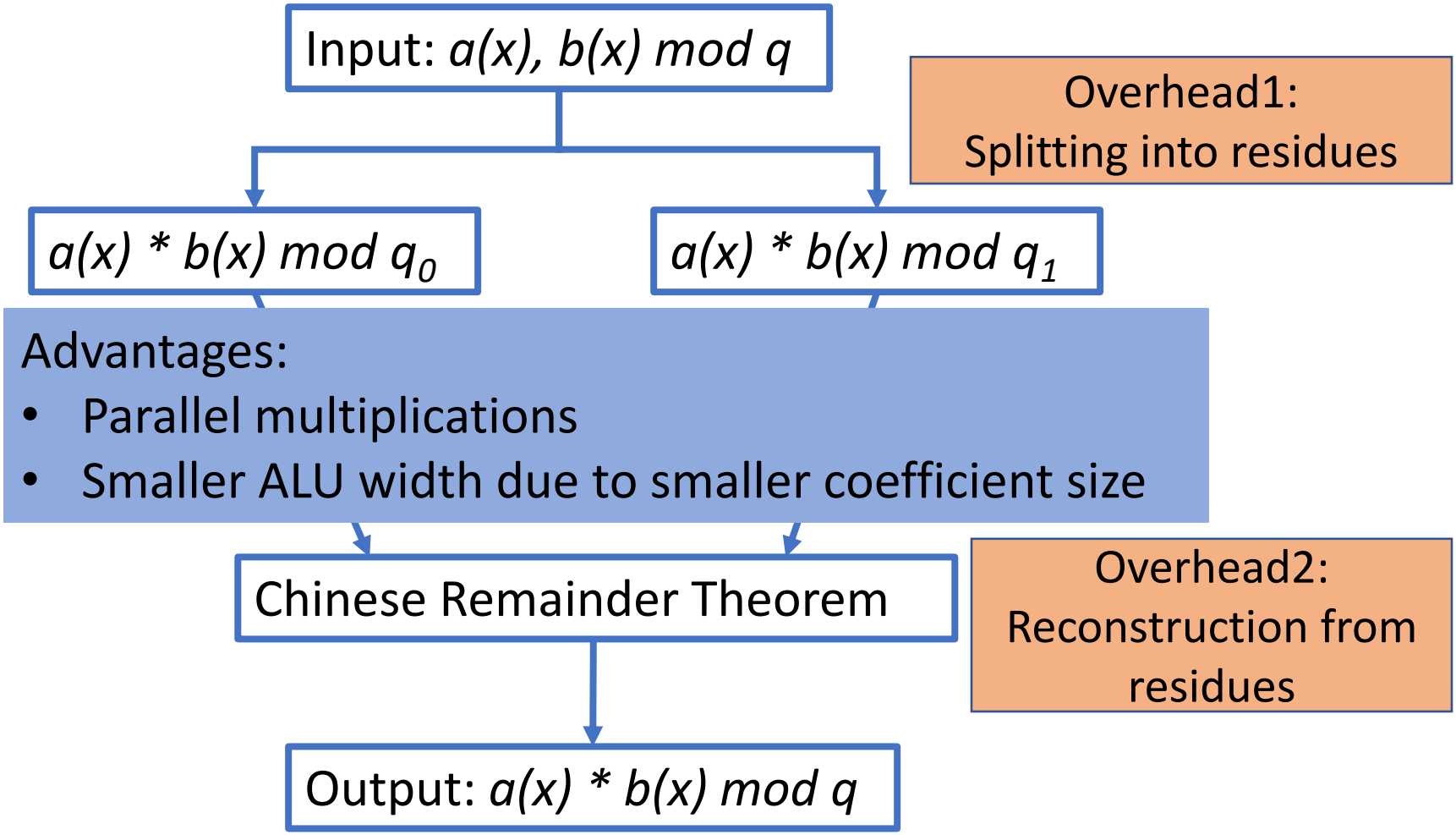


RNS arithmetic

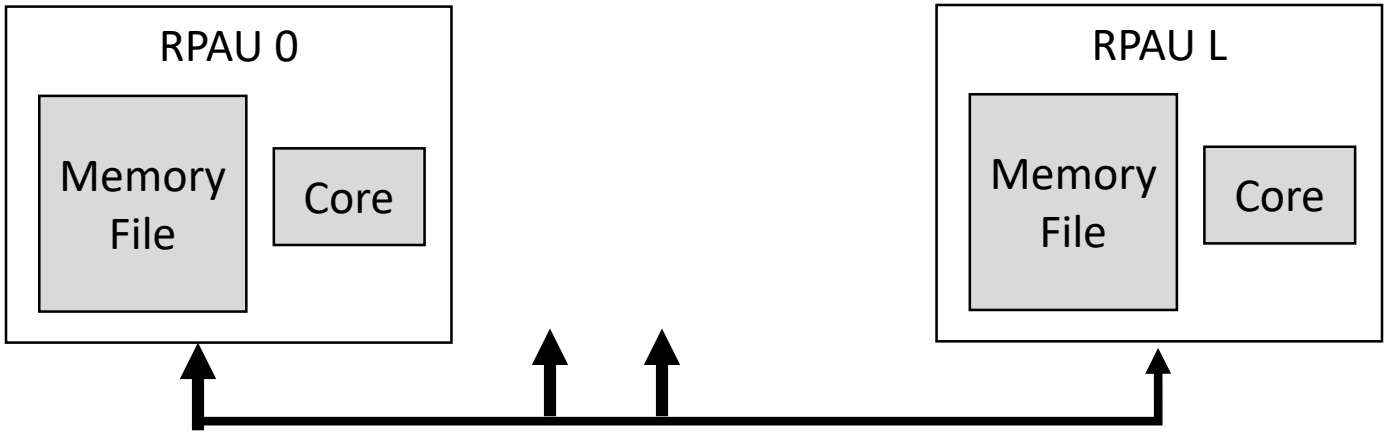
- Small coefficients
- Parallel computation

Example: polynomial multiplication

Let $q = q_0 \cdot q_1$ where q_0 and q_1 are of equal bit-length



On Hardware Parallel processing using multiple Residue Polynomial Arithmetic Unit (RPAU)

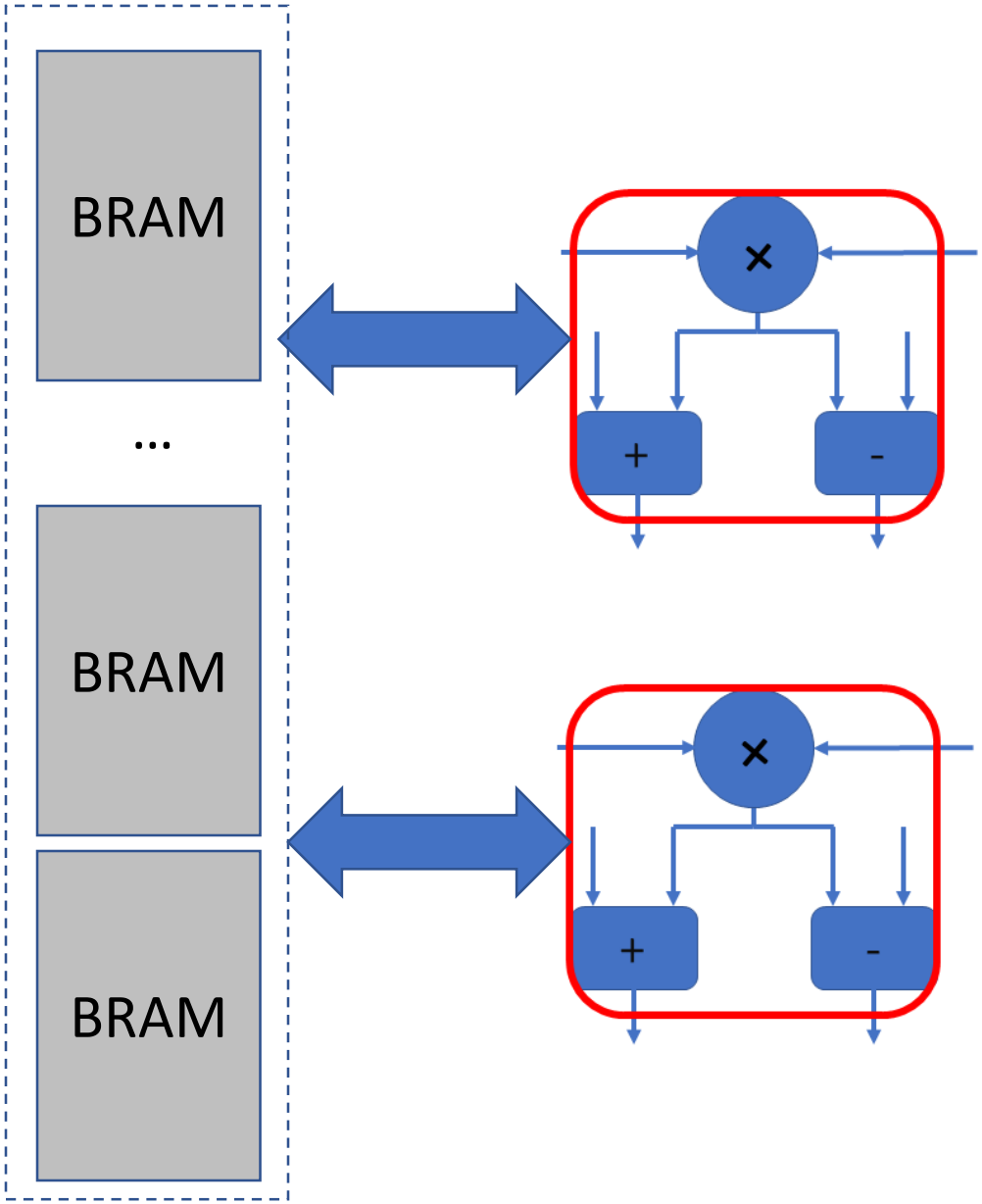


Number or RPAUs is a design parameter

Hardware accelerators for Homomorphic Computation

→Arithmetic of large coefficients
→Arithmetic of large polynomials

Polynomial multiplication: multiple butterfly cores



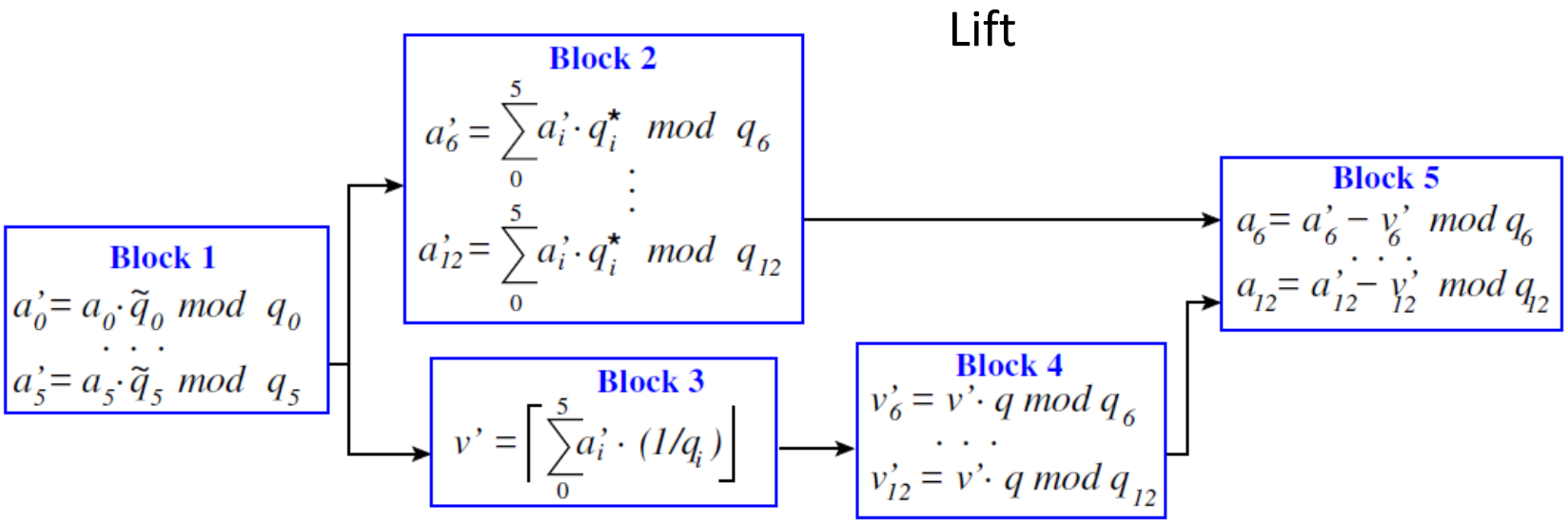
Single core NTT too slow!

Design Challenges:

- Long routing
- Memory access conflicts

Memory access parallelism

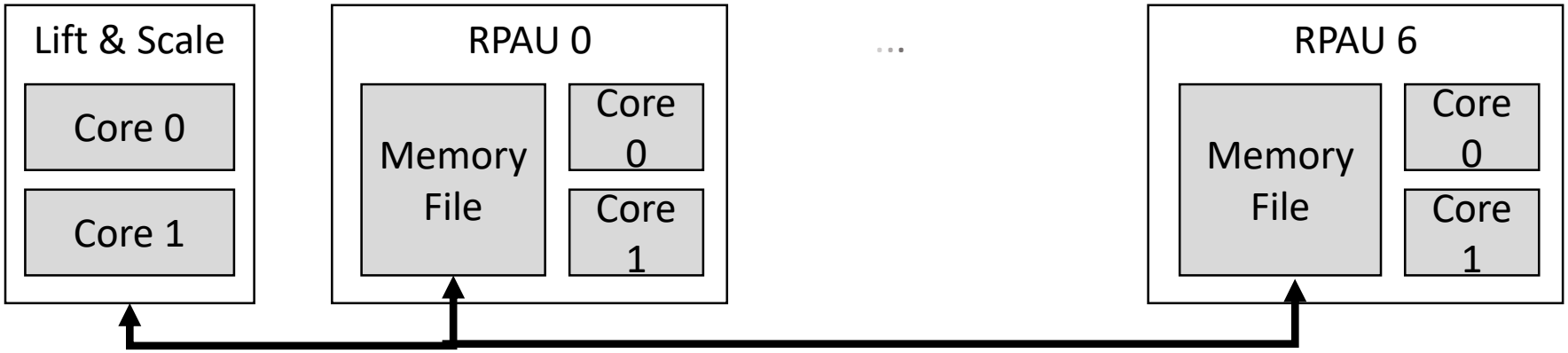
Block Level Pipelining



- Separate building blocks for block-level pipeline
- Realize a resource shared architecture
  - Reduces the area requirement
  - Increase the computation time

Execution Units

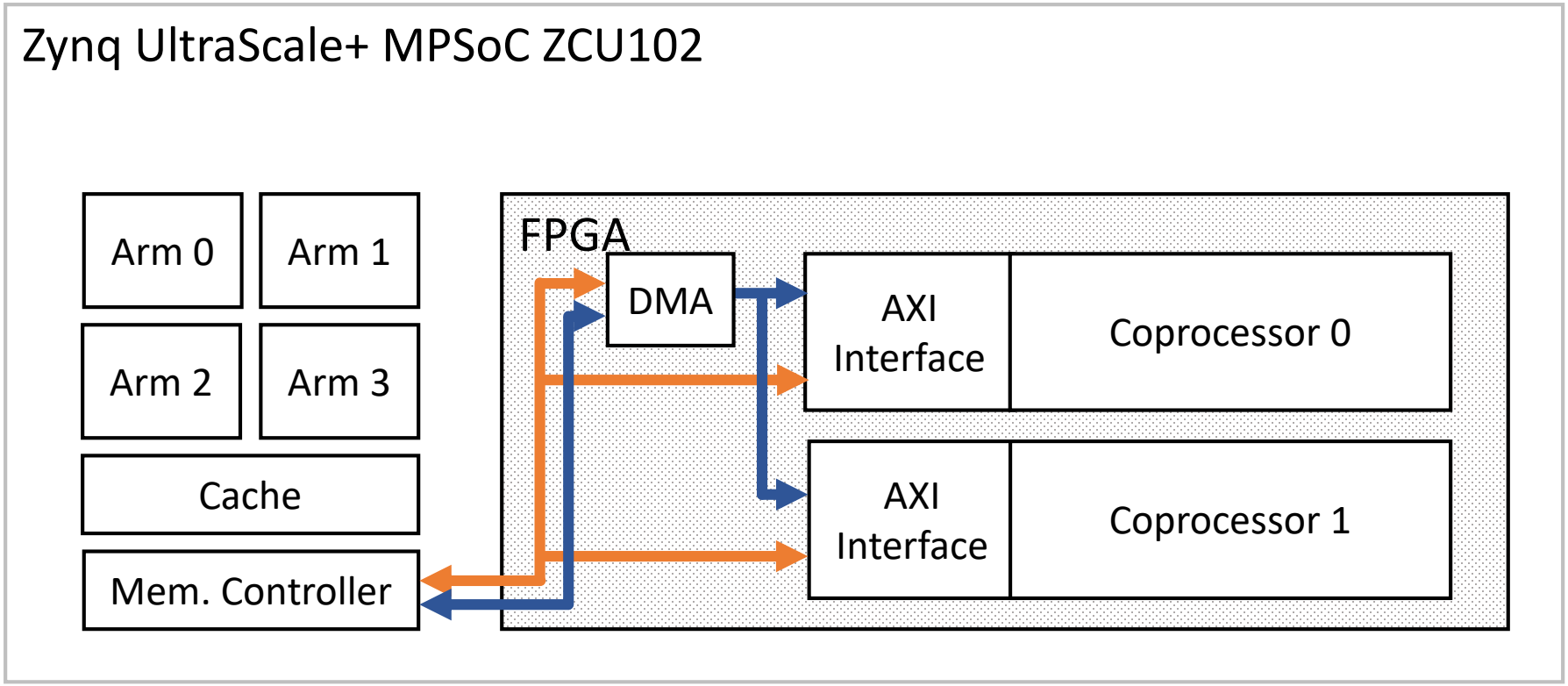
- Two parallel cores for Lift and Scale
- Seven Residue Polynomial Arithmetic Unit (RPAU)



Parameter:

- Ciphertext polynomial degree 4096
- Ciphertext coefficient size 180

Arm + FPGA Implementation



Source code public on Github

Performance of High-Level Operations

Operation	Speed (cycles)	Speed (msec)
Add in HW	31,339	0.026
Multiply in HW	**5,349,567**	**4.458**
Send two ciphertext to HW	434,013	0.362
Receive result ciphertext from HW	216,697	0.180

Measurements are in cycles of CPU clocked at 1200 MHz
Coprocessor is clocked at 200 MHz

Publication: HPCA 2019
400 homomorphic multiplications per sec (2 cores)
Faster than Tesla K80 GPU

Resource Utilization

	LUTs	REGs	BRAMs	DSPs
	# of used instances % utilization			
Two Coprocessors & Interface	133,692 49	60,312 11	815 89	416 16
A Single Coprocessor & Interface	63,522 23	25,622 5	388 43	208 8

Conclusions so far

- Ring-LWE is efficient in hardware and software
- But, there are security concerns due to special structure

$$\begin{pmatrix} a_0 & -a_3 & -a_2 & -a_1 \\ a_1 & a_0 & -a_3 & -a_2 \\ a_2 & a_1 & a_0 & -a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} * \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{pmatrix} \approx \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \pmod q$$

Special structure in matrix

Interpolating LWE and ring-LWE: Module LWE

$$\begin{pmatrix} \begin{matrix} a_0 & -a_3 & -a_2 & -a_1 \\ a_1 & a_0 & -a_3 & -a_2 \\ a_2 & a_1 & a_0 & -a_3 \\ a_3 & a_2 & a_1 & a_0 \end{matrix} & \begin{matrix} a_8 & -a_{11} & -a_{10} & -a_9 \\ a_9 & a_8 & -a_{11} & -a_{10} \\ a_{10} & a_9 & a_8 & -a_{11} \\ a_{11} & a_{10} & a_7 & a_8 \end{matrix} \\ \begin{matrix} a_4 & -a_7 & -a_6 & -a_5 \\ a_5 & a_4 & -a_7 & -a_6 \\ a_6 & a_5 & a_4 & -a_7 \\ a_7 & a_6 & a_5 & a_4 \end{matrix} & \begin{matrix} a_{12} & -a_{15} & -a_{14} & -a_{13} \\ a_{13} & a_{12} & -a_{15} & -a_{14} \\ a_{14} & a_{13} & a_{12} & -a_{15} \\ a_{15} & a_{14} & a_{13} & a_{12} \end{matrix} \end{pmatrix} * \begin{pmatrix} \begin{matrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{matrix} \\ \begin{matrix} s_4 \\ s_5 \\ s_6 \\ s_7 \end{matrix} \end{pmatrix} + \begin{pmatrix} \begin{matrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{matrix} \\ \begin{matrix} e_4 \\ e_5 \\ e_6 \\ e_7 \end{matrix} \end{pmatrix} \approx \begin{pmatrix} \begin{matrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{matrix} \\ \begin{matrix} b_4 \\ b_5 \\ b_6 \\ b_7 \end{matrix} \end{pmatrix}$$

$$\begin{pmatrix} a_{0,0}(x) & a_{0,1}(x) \\ a_{1,0}(x) & a_{1,1}(x) \end{pmatrix} * \begin{pmatrix} s_0(x) \\ s_1(x) \end{pmatrix} + \begin{pmatrix} e_0(x) \\ e_1(x) \end{pmatrix} \approx \begin{pmatrix} b_0(x) \\ b_1(x) \end{pmatrix} \pmod{q} \pmod{x^4 + 1}$$

Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM

a lattice-based candidate for NIST standardization moved to second round!

Jointly designed by EE and Math team!

SABER: flexibility and efficiency

- Saber uses module-LWR problem
- Polynomials are always of 256 coefficients **[Efficient pol. arithmetic]**
- **Flexibility:** matrix dimensions is parameterizable
  - 2-by-2 for 115-bit post-quantum security

Light SABER

  - 3-by-3 for 180-bit post-quantum security

SABER

  - 4-by-4 for 245-bit post-quantum security

Fire SABER

SABER: Parameter set

$$\left\lfloor \frac{p}{q} \begin{pmatrix} a_{0,0}(x) & \dots & a_{0,k-1}(x) \\ & \dots & \\ a_{k-1,0}(x) & \dots & a_{k-1,k-1}(x) \end{pmatrix} * \begin{pmatrix} s_0(x) \\ \dots \\ s_{k-1}(x) \end{pmatrix} \right\rceil \approx \begin{pmatrix} b_0(x) \\ \\ b_{k-1}(x) \end{pmatrix} \pmod{x^{256} + 1}$$

- Polynomials of fixed size 256 coefficients ✓
- Flexible dimension k = 2, 3 or 4 ✓
- How to choose p and q?

Learning with rounding (LWR)

A problem with rounding:

$$\left\lfloor \frac{p}{q} \boxed{\text{Uniform in [0, q-1]}} \right\rceil \quad \text{where } p < q$$

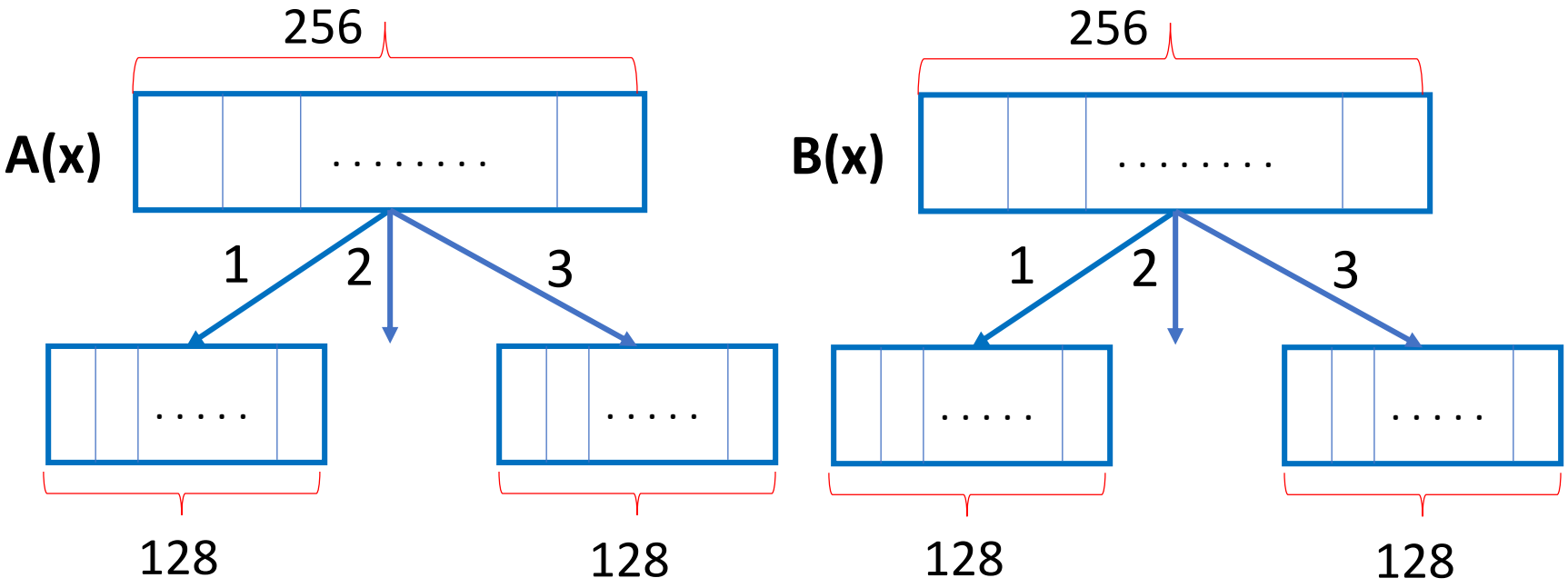
Prime q introduces rounding bias

- Cannot use prime q ☹
- Hence, no NTT-based fast polynomial multiplication
+ No modular reduction + Easy rounding

→ We need to use generic polynomial multiplication algorithm

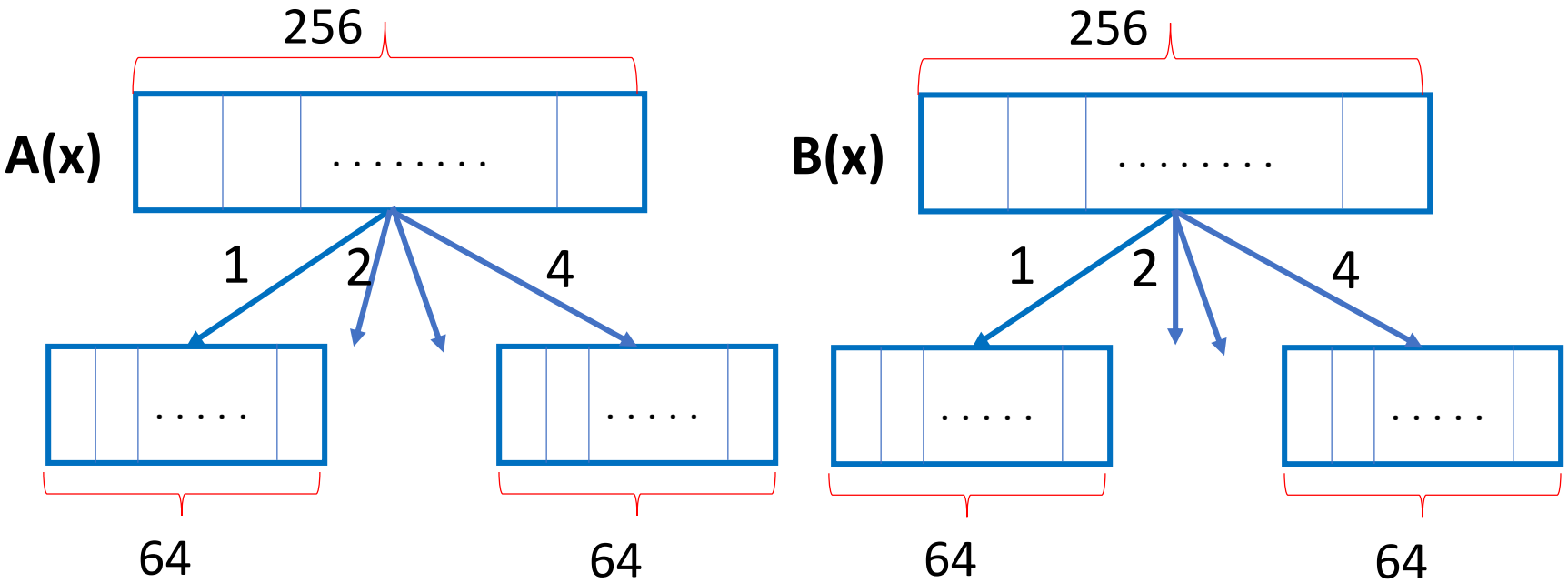
Next best polynomial multiplication algorithms

- Karatsuba multiplication $O(n^{\log_2 3})$



Next best polynomial multiplication algorithms

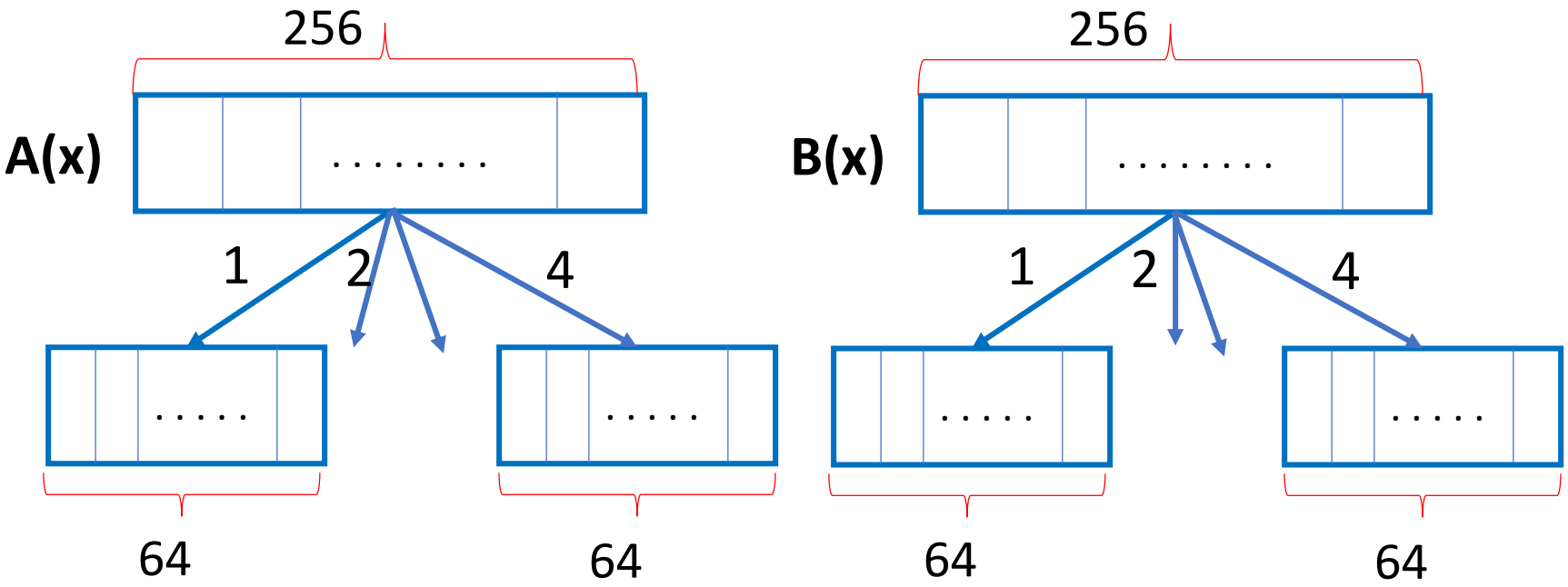
- Toom-Cook multiplication



Toom-Cook 4 Way needs 7 multiplications

Karatsuba would need 9 multiplications

Toom-Cook 4 Way: step-by-step: splitting



Splitting operand into 4 polynomials

Take $y = x^{64}$

$$\mathbf{A}(y) = A_3\, y^3 + A_2\, y^2 + A_1\, y + A_0$$

$$\mathbf{B}(y) = B_3\, y^3 + B_2\, y^2 + B_1\, y + B_0$$

Toom-Cook 4 Way: step-by-step: evaluation

$$
\begin{aligned}
w_1 &= A(\infty) * B(\infty) = A_3 * B_3 \\
w_2 &= A(2) * B(2) = (A_0 + 2 \cdot A_1 + 4 \cdot A_2 + 8 \cdot A_3) * (B_0 + 2 \cdot B_1 + 4 \cdot B_2 + 8 \cdot B_3) \\
w_3 &= A(1) * B(1) = (A_0 + A_1 + A_2 + A_3) * (B_0 + B_1 + B_2 + B_3) \\
w_4 &= A(-1) * B(-1) = (A_0 - A_1 + A_2 - A_3) * (B_0 - B_1 + B_2 - B_3) \\
w_5 &= A(\tfrac{1}{2}) * B(\tfrac{1}{2}) = (8 \cdot A_0 + 4 \cdot A_1 + 2 \cdot A_2 + A_3) * (8 \cdot B_0 + 4 \cdot B_1 + 2 \cdot B_2 + B_3) \\
w_6 &= A(\tfrac{-1}{2}) * B(\tfrac{-1}{2}) = (8 \cdot A_0 - 4 \cdot A_1 + 2 \cdot A_2 - A_3) * (8 \cdot B_0 - 4 \cdot B_1 + 2 \cdot B_2 - B_3) \\
w_7 &= A(0) * B(0) = A_0 * B_0
\end{aligned}
$$

Linear operations

+

Seven multiplications are computed

Toom-Cook 4 Way: step-by-step: interpolation

```
// Interpolation
```

$$w_2 = w_2 + w_5$$
$$w_6 = w_6 - w_5$$
$$w_4 = (w_4 - w_3)/2$$
$$w_5 = w_5 - w_1 - 64 \cdot w_7$$
$$w_3 = w_3 + w_4$$
$$w_5 = 2 \cdot w_5 + w_6$$
$$w_2 = w_2 - 65 \cdot w_3$$
$$w_3 = w_3 - w_7 - w_1$$
$$w_2 = w_2 + 45 \cdot w_3$$
$$w_5 = (w_5 - 8 \cdot w_3)/\boxed{24}$$
$$w_6 = w_6 + w_2$$
$$w_2 = (w_2 + 16 \cdot w_4)/18$$
$$w_3 = w_3 - w_5$$
$$w_4 = -(w_4 + w_2)$$
$$w_6 = (30 \cdot w_2 - w_6)/60$$
$$w_2 = w_2 - w_6$$
$$\textbf{return } w_1 \cdot y^6 + w_2 \cdot y^5 + w_3 \cdot y^4 + w_4 \cdot y^3 + w_5 \cdot y^2 + w_6 \cdot y + w_7;$$

Linear operations

This number has a role to play

Linear operations

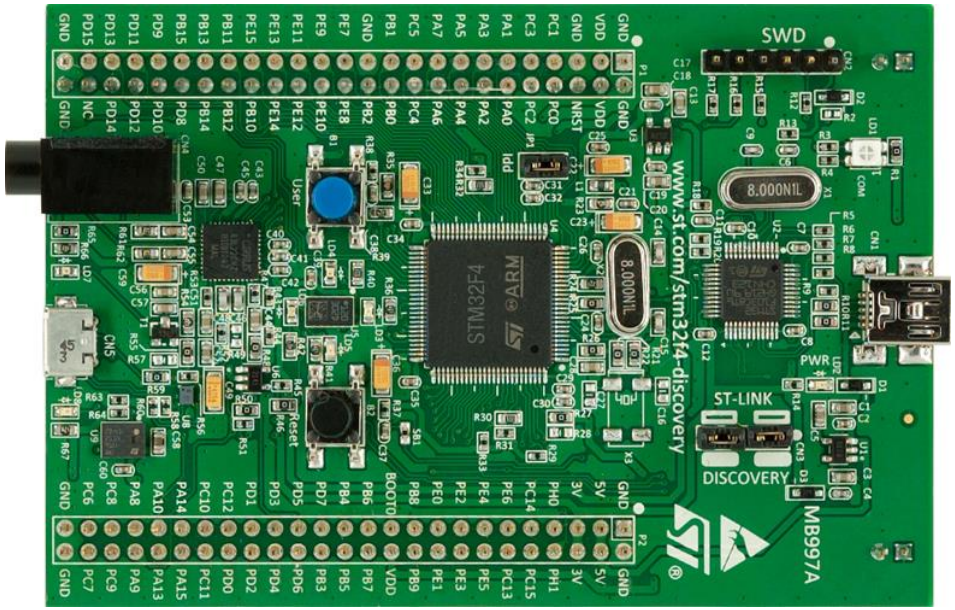
Advanced Vector Extensions (AVX)

Intrinsics Guide

```
__m256i _mm256_abs_epi16 (__m256i a)
__m256i _mm256_add_epi16 (__m256i a, __m256i b)
__m256i _mm256_adds_epi16 (__m256i a, __m256i b)
__m256i _mm256_blend_epi16 (__m256i a, __m256i b, const int imm8)
__m128i _mm_broadcastw_epi16 (__m128i a)
__m256i _mm256_broadcastw_epi16 (__m128i a)
__m256i _mm256_cmpeq_epi16 (__m256i a, __m256i b)
__m256i _mm256_cmpgt_epi16 (__m256i a, __m256i b)
__m256i _mm256_cvtepi16_epi32 (__m128i a)
__m256i _mm256_cvtepi16_epi64 (__m128i a)
__m256i _mm256_cvtepi8_epi16 (__m128i a)
__m256i _mm256_cvtepu8_epi16 (__m128i a)
int _mm256_extract_epi16 (__m256i a, const int index)
__m256i _mm256_hadd_epi16 (__m256i a, __m256i b)
__m256i _mm256_hadds_epi16 (__m256i a, __m256i b)
__m256i _mm256_hsub_epi16 (__m256i a, __m256i b)
__m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)
```

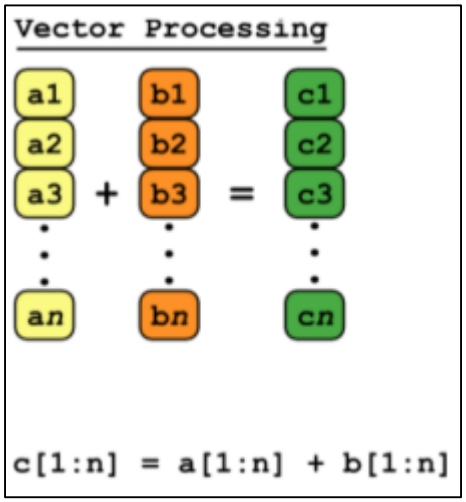
Vectorized instructions for 16-bit operands

DSP instructions



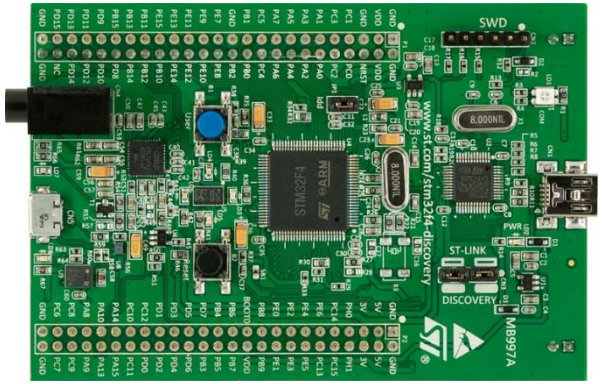
ARM Cortex-M4

- Popular 32-bit microcontroller
- Has DSP instructions for half-word operations



AVX

+



Microcontroller with DSP

Keep coefficients smaller/equal to 16 bits to use

- _epi16( ) AVX intrinsics in high-end platforms
- DSP instructions in low-end microcontrollers

Options for q: 2^{16}, 2^{15}, 2^{14}, 2^{13} … etc

Toom-Cook 4 Way: step-by-step: interpolation

```
// Interpolation
```

$w_2 = w_2 + w_5$

$w_6 = w_6 - w_5$

$w_4 = (w_4 - w_3)/2$

$w_5 = w_5 - w_1 - 64 \cdot w_7$

$w_3 = w_3 + w_4$

$w_5 = 2 \cdot w_5 + w_6$

$w_2 = w_2 - 65 \cdot w_3$

$w_3 = w_3 - w_7 - w_1$

$w_2 = w_2 + 45 \cdot w_3$

$w_5 = (w_5 - 8 \cdot w_3)/\boxed{24}$ → This number has a role to play

$w_6 = w_6 + w_2$

$w_2 = (w_2 + 16 \cdot w_4)/18$

$w_3 = w_3 - w_5$

$w_4 = -(w_4 + w_2)$

$w_6 = (30 \cdot w_2 - w_6)/60$

$w_2 = w_2 - w_6$

return $w_1 \cdot y^6 + w_2 \cdot y^5 + w_3 \cdot y^4 + w_4 \cdot y^3 + w_5 \cdot y^2 + w_6 \cdot y + w_7;$

Division by 24 in Toom-Cook Interpolation

$$w_5 = (w_5 - 8 \cdot w_3)/24$$

- $24 = 8 \cdot 3$
- We are working in R_q where $q = 2^i$
- 3 has inverse in mod q
  E.g. $3^{-1} \bmod 2^{15} \rightarrow 10923$
- So, division by 3 is same as multiplying by $3^{-1} \bmod q$

Division by 24 in Toom-Cook Interpolation

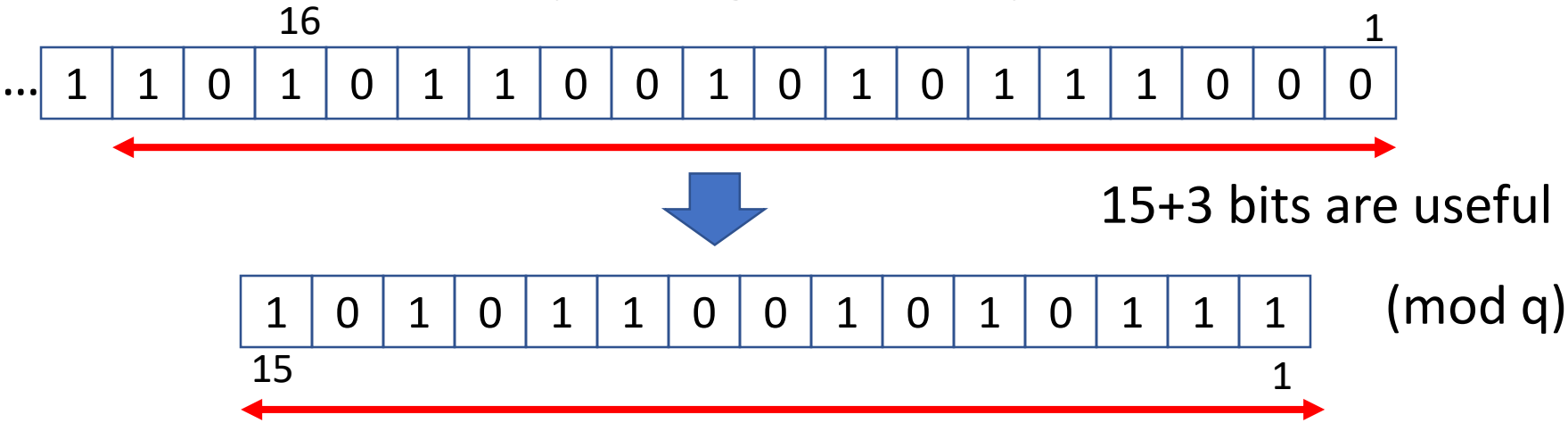
$$w_5 = (w_5 - 8 \cdot w_3)/24$$

- $24 = \mathbf{8} \cdot 3$
- We are working in R_q where $q = 2^i$
- But, 8 does not have inverse in mod q

Only option: do actual division

Working with $q = 2^{15}$

Example: integer division by $8=2^3$



In 16-bit Computer:

- Difficult to implement: requires careful arithmetic of two words
- Slower arithmetic

Working with $q = 2^{13}$

Example: integer division by $8=2^3$

…	1	1	0	1	0	1	1	0	0	1	0	1	0	1	1	1	0	0	0
				16															1

13+3 bits are useful

1	0	1	1	0	0	1	0	1	0	1	1	1
13												1

(mod q)

Fits in 16-bit words ☺

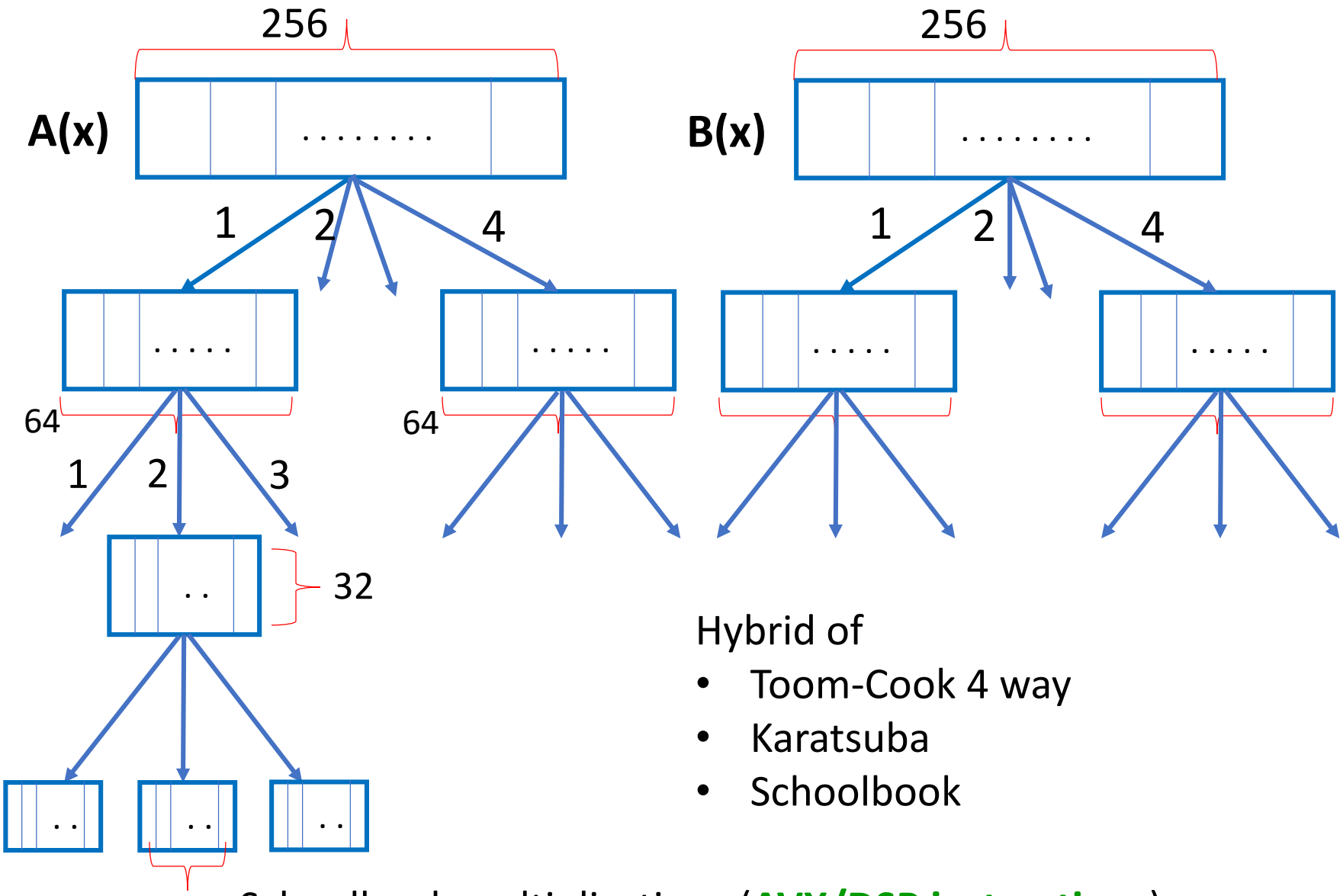
In 16-bit Computer:

- **Easy** to implement
- **Less** complicated arithmetic

Saber Parameters

- Polynomial length n = 256
- q = 2^13
- p = 2^10

Polynomial multiplication in Saber



Hybrid of

- Toom-Cook 4 way
- Karatsuba
- Schoolbook

Schoolbook multiplications (**AVX/DSP instructions**)

School-book multiplication using AVX

Consider one polynomial multiplication

$$\ldots\ b_3\,x^3 + b_2\,x^2 + b_1\,x + b_0$$

$$\ldots\ a_3\,x^3 + a_2\,x^2 + a_1\,x + a_0$$

$$\begin{array}{cccccc}
 & & & a_0b_2 & a_0b_1 & a_0b_0 \\
\cdot & \cdot & \cdot & a_1b_1 & a_1b_0 & \\
 & & & a_2b_0 & &
\end{array}$$

School-book multiplication using AVX

Consider 16 polynomial multiplications

$a^{15}(x)*b^{15}(x)$

$$\ldots\ b_3^{15}\,x^3 + b_2^{15}\,x^2 + b_1^{15}\,x + b_0^{15}$$

$$\ldots\ a_3^{15}\,x^3 + a_2^{15}\,x^2 + a_1^{15}\,x + a_0^{15}$$

$$\begin{array}{cccccc} & & a_0^{15}b_2^{15} & a_0^{15}b_1^{15} & a_0^{15}b_0^{15} \\ \cdot & \cdot & \cdot\ a_1^{15}b_1^{15} & a_1^{15}b_0^{15} & \\ & & a_2^{15}b_0^{15} & & \end{array}$$

$a^{0}(x)*b^{0}(x)$

$$\ldots\ b_3^{0}\,x^3 + b_2^{0}\,x^2 + b_1^{0}\,x + b_0^{0}$$

$$\ldots\ a_3^{0}\,x^3 + a_2^{0}\,x^2 + a_1^{0}\,x + a_0^{0}$$

$$\begin{array}{cccccc} & & a_0^{0}b_2^{0} & a_0^{0}b_1^{0} & a_0^{0}b_0^{0} \\ \cdot & \cdot & \cdot\ a_1^{0}b_1^{0} & a_1^{0}b_0^{0} & \\ & & a_2^{0}b_0^{0} & & \end{array}$$

and 16x vectorized processor

School-book multiplication using AVX

Consider 16 polynomial multiplications

$$\ldots\ b_3^{15}\,x^3 + b_2^{15}\,x^2 + b_1^{15}\,x + b_0^{15}$$

$$\ldots\ a_3^{15}\,x^3 + a_2^{15}\,x^2 + a_1^{15}\,x + a_0^{15}$$

$$\begin{array}{cccc}
 & a_0^{15}b_2^{15} & a_0^{15}b_1^{15} & a_0^{15}b_0^{15} \\
\cdot\ \cdot\ \cdot & a_1^{15}b_1^{15} & a_1^{15}b_0^{15} & \\
 & a_2^{15}b_0^{15} & &
\end{array}$$

$$\ldots\ b_3^{0}\,x^3 + b_2^{0}\,x^2 + b_1^{0}\,x + b_0^{0}$$

$$\ldots\ a_3^{0}\,x^3 + a_2^{0}\,x^2 + a_1^{0}\,x + a_0^{0}$$

$$\begin{array}{cccc}
 & a_0^{0}b_2^{0} & a_0^{0}b_1^{0} & a_0^{0}b_0^{0} \\
\cdot\ \cdot\ \cdot & a_1^{0}b_1^{0} & a_1^{0}b_0^{0} & \\
 & a_2^{0}b_0^{0} & &
\end{array}$$

and 16x vectorized processor

A0 =

a_0^{15}	$\cdots$	a_0^{2}	a_0^{1}	a_0^{0}

B0 =

b_0^{15}	$\cdots$	b_0^{2}	b_0^{1}	b_0^{0}

School-book multiplication using AVX

Consider 16 polynomial multiplications

$$\ldots\ b_3^{15}x^3 + b_2^{15}x^2 + b_1^{15}x + b_0^{15}$$

$$\ldots\ a_3^{15}x^3 + a_2^{15}x^2 + a_1^{15}x + a_0^{15}$$

$$\begin{array}{cccc} & a_0^{15}b_2^{15} & a_0^{15}b_1^{15} & \boxed{a_0^{15}b_0^{15}} \\ \cdot\ \cdot\ \cdot & a_1^{15}b_1^{15} & a_1^{15}b_0^{15} & \\ & a_2^{15}b_0^{15} & & \end{array}$$

$$\ldots\ b_3^{0}x^3 + b_2^{0}x^2 + b_1^{0}x + b_0^{0}$$

$$\ldots\ a_3^{0}x^3 + a_2^{0}x^2 + a_1^{0}x + a_0^{0}$$

$$\begin{array}{cccc} & a_0^{0}b_2^{0} & a_0^{0}b_1^{0} & \boxed{a_0^{0}b_0^{0}} \\ \cdot\ \cdot\ \cdot & a_1^{0}b_1^{0} & a_1^{0}b_0^{0} & \\ & a_2^{0}b_0^{0} & & \end{array}$$

and 16x vectorized processor

AVX_MUL(A0, B0)

A0 = $\boxed{a_0^{15}} \cdots \boxed{a_0^{2}}\ \boxed{a_0^{1}}\ \boxed{a_0^{0}}$

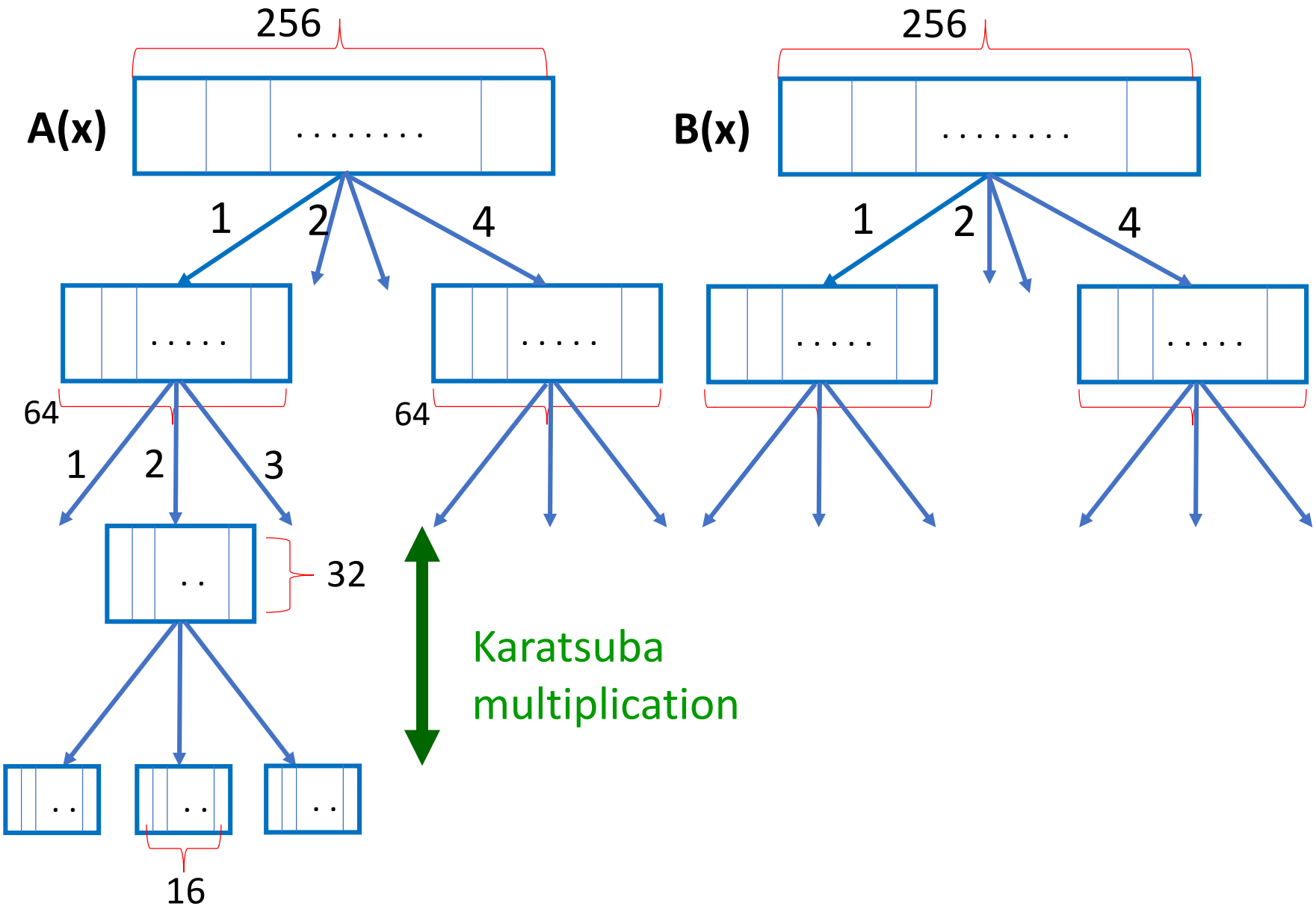
A0B0= $\boxed{a_0^{15}b_0^{15}}\ \ldots\ \boxed{a_0^{1}b_0^{1}}\ \boxed{a_0^{0}b_0^{0}}$

B0 = $\boxed{b_0^{15}} \cdots \boxed{b_0^{2}}\ \boxed{b_0^{1}}\ \boxed{b_0^{0}}$

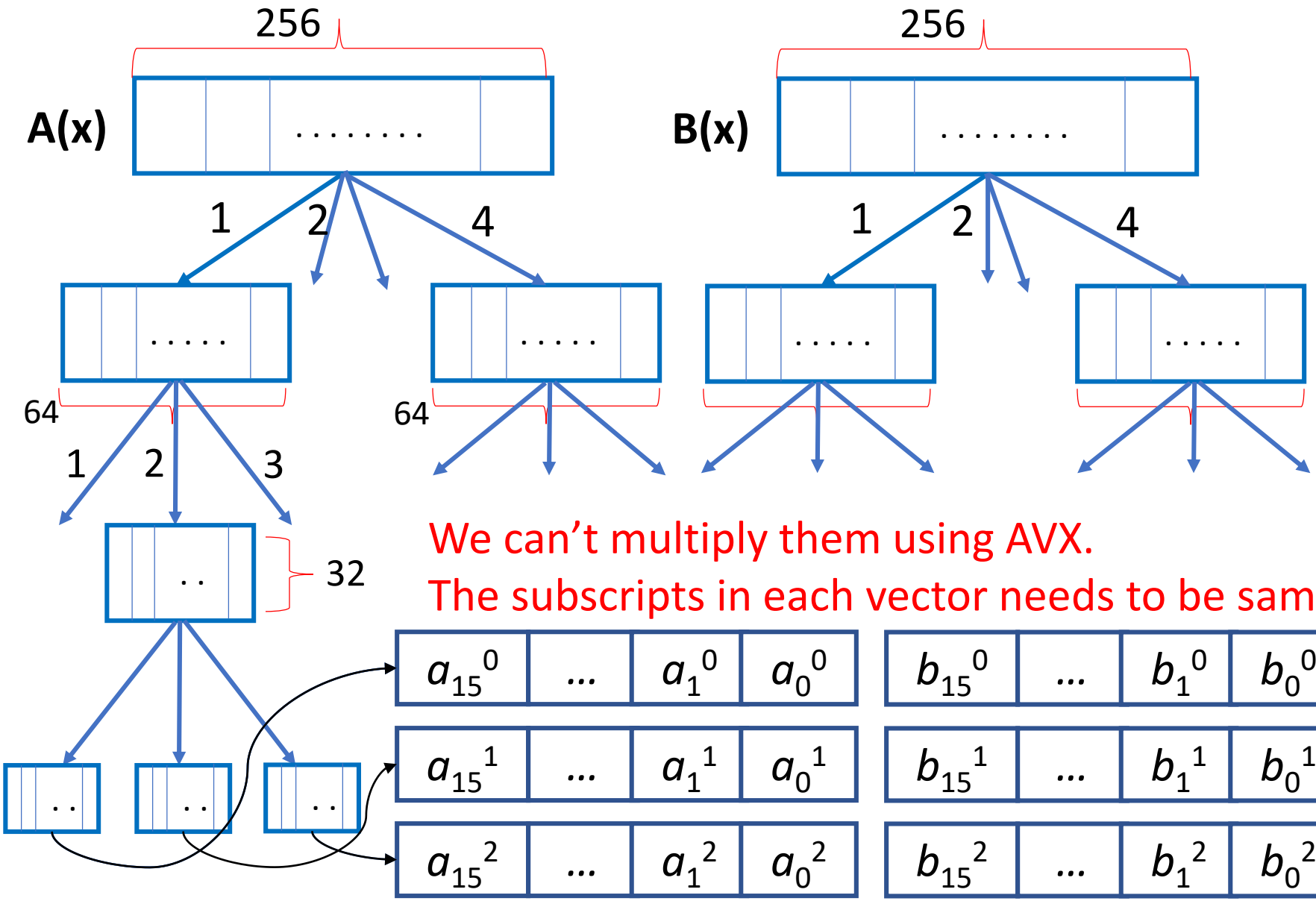
All in parallel

(assumes all coeffs. are available)

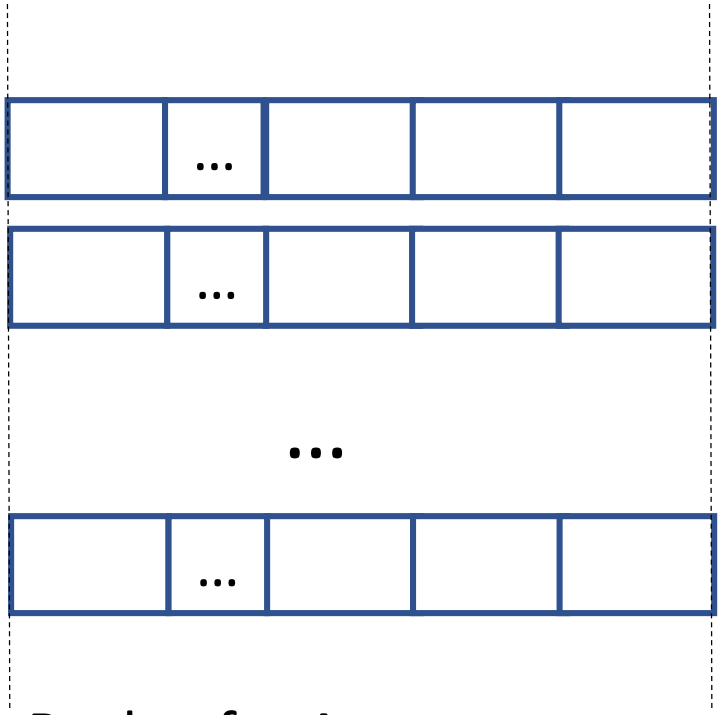
Polynomial multiplication



Polynomial multiplication



Buckets for 16 coeff. polynomials

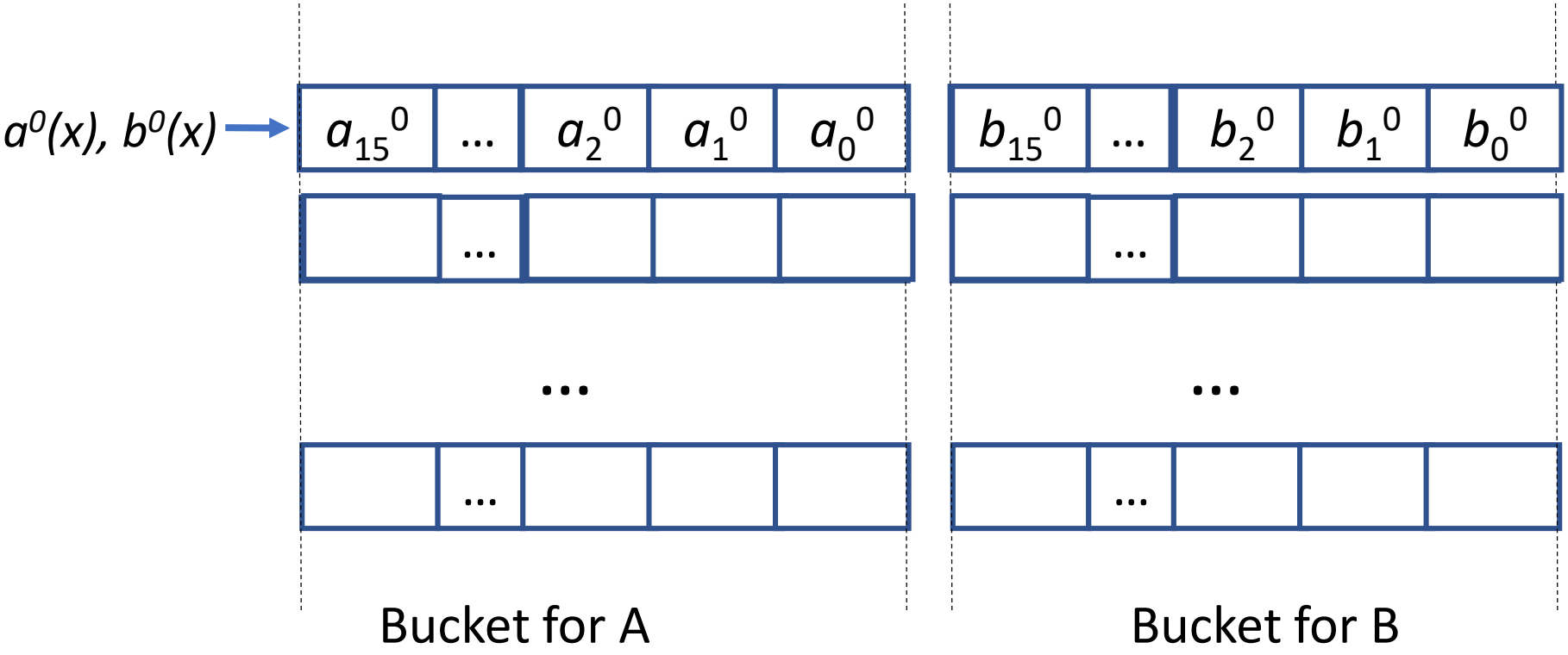


Bucket for A.

- Row contains one 16 coeff. Pol
- There are 16 rows

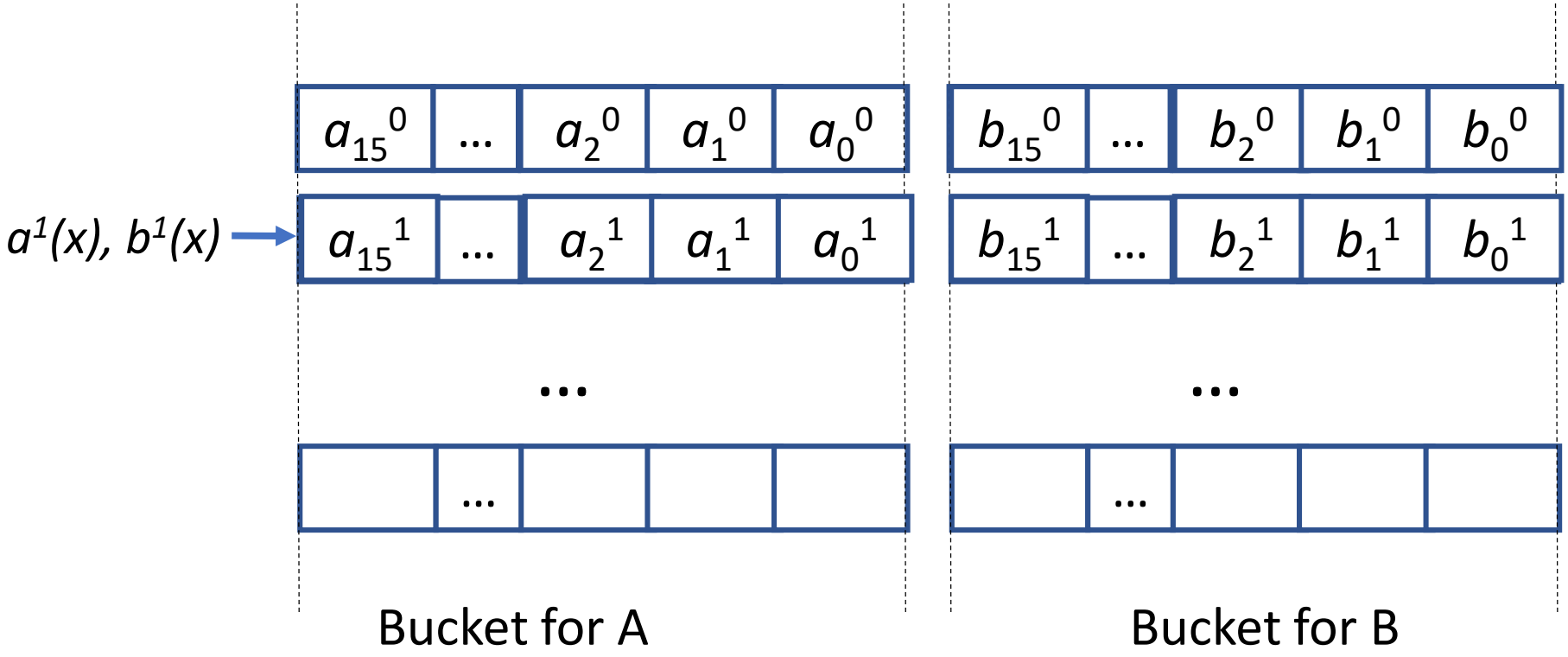
Total 256 coeff. In the bucket

Buckets for 16 coeff. polynomials



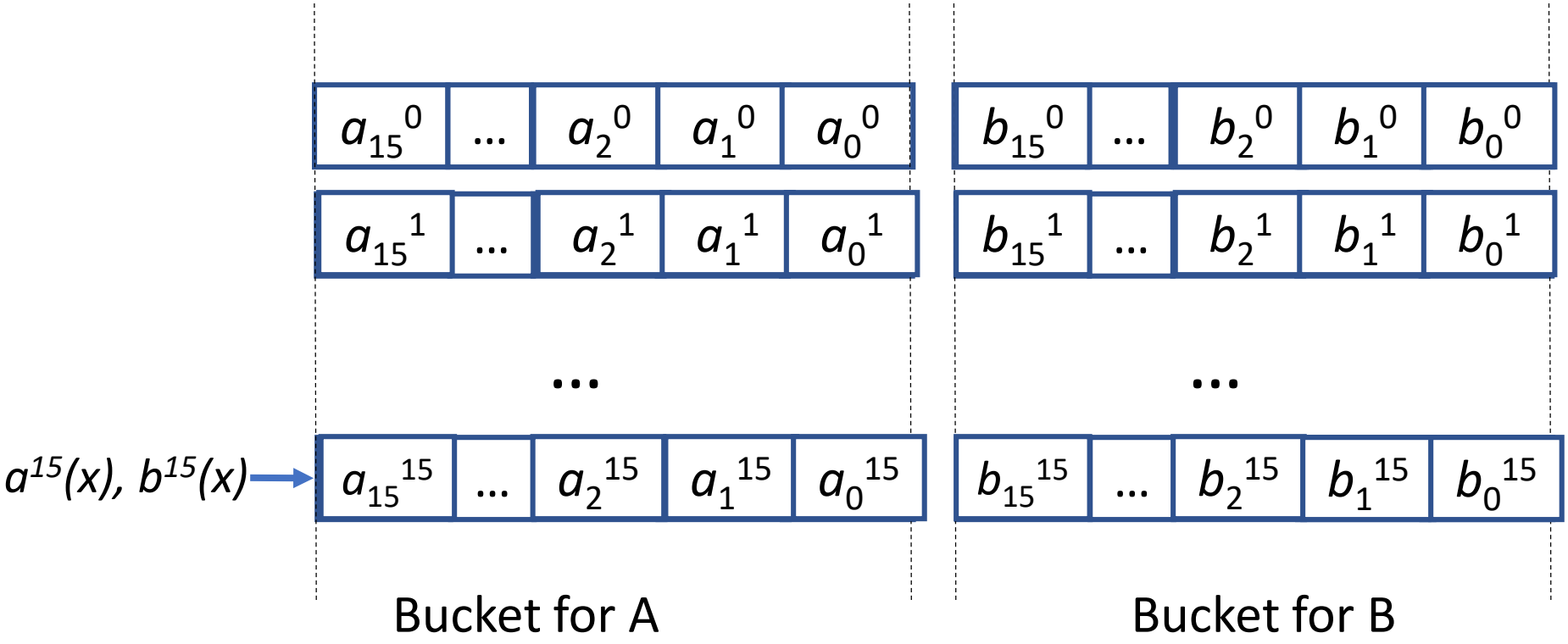
- Multiplications are not performed immediately
- The buckets are filled first

Buckets for 16 coeff. polynomials



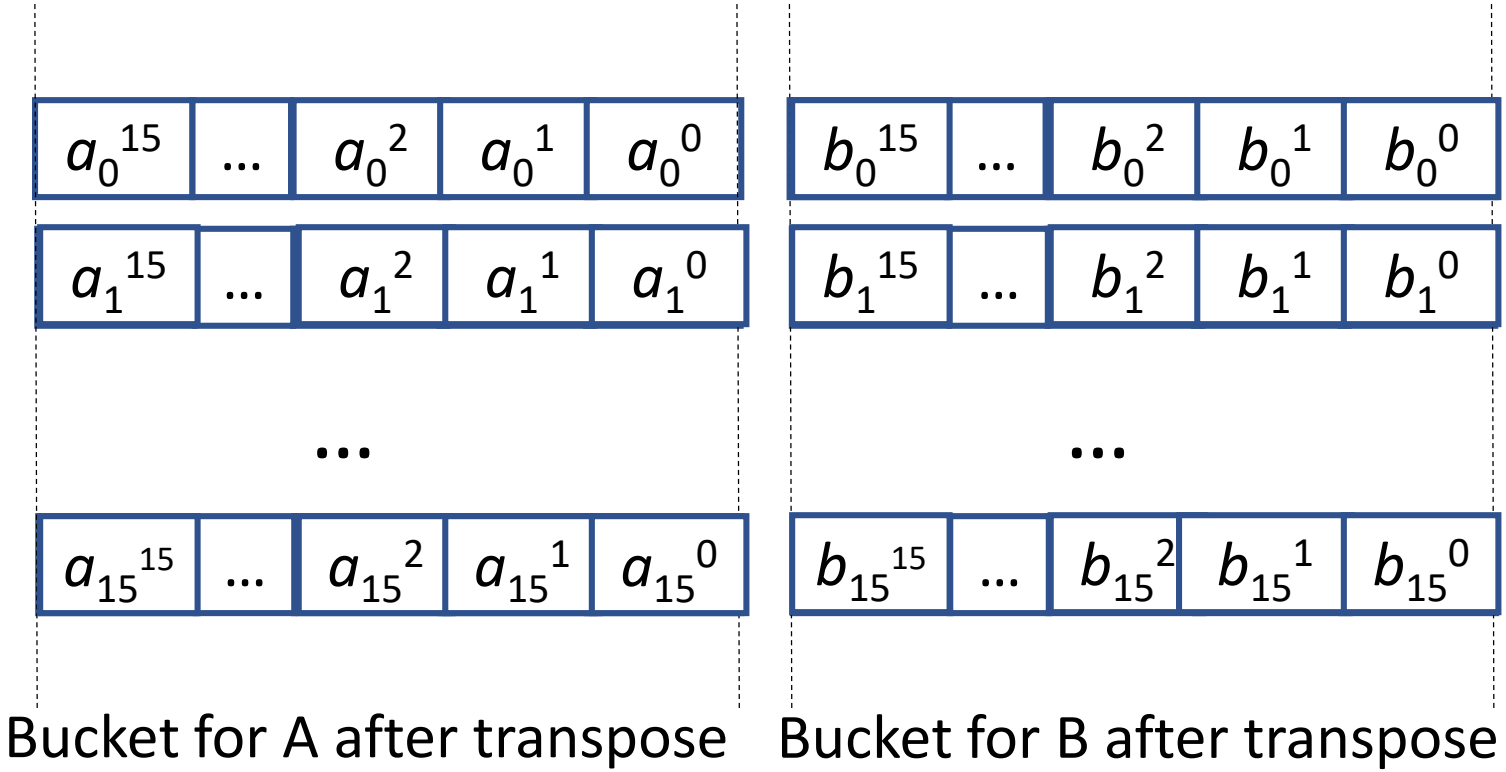
- Multiplications are not performed immediately
- The buckets are filled first

Buckets for 16 coeff. polynomials



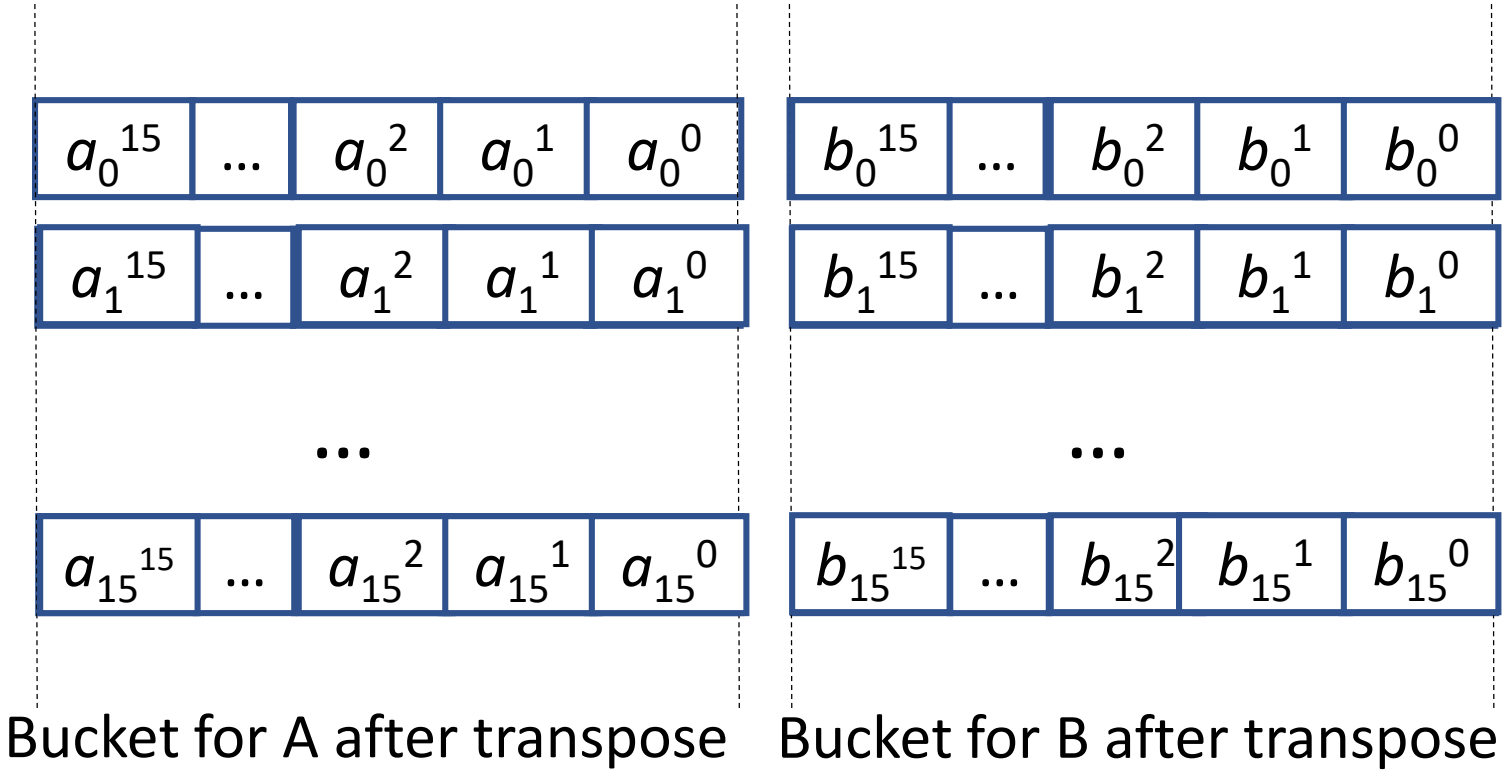
- Multiplications are not performed immediately
- The buckets are filled first

Buckets for 16 coeff. polynomials



- Transpose each bucket
- Transpose using AVX is an interesting optimization problem

Buckets for 16 coeff. polynomials



- Now multiply vectors using AVX instruction
- Obtain C which is a vector length 32, containing 32*16 coeffs.
- Finally, transpose C to get 16 result polynomials

School-book multiplication using DSP instructions

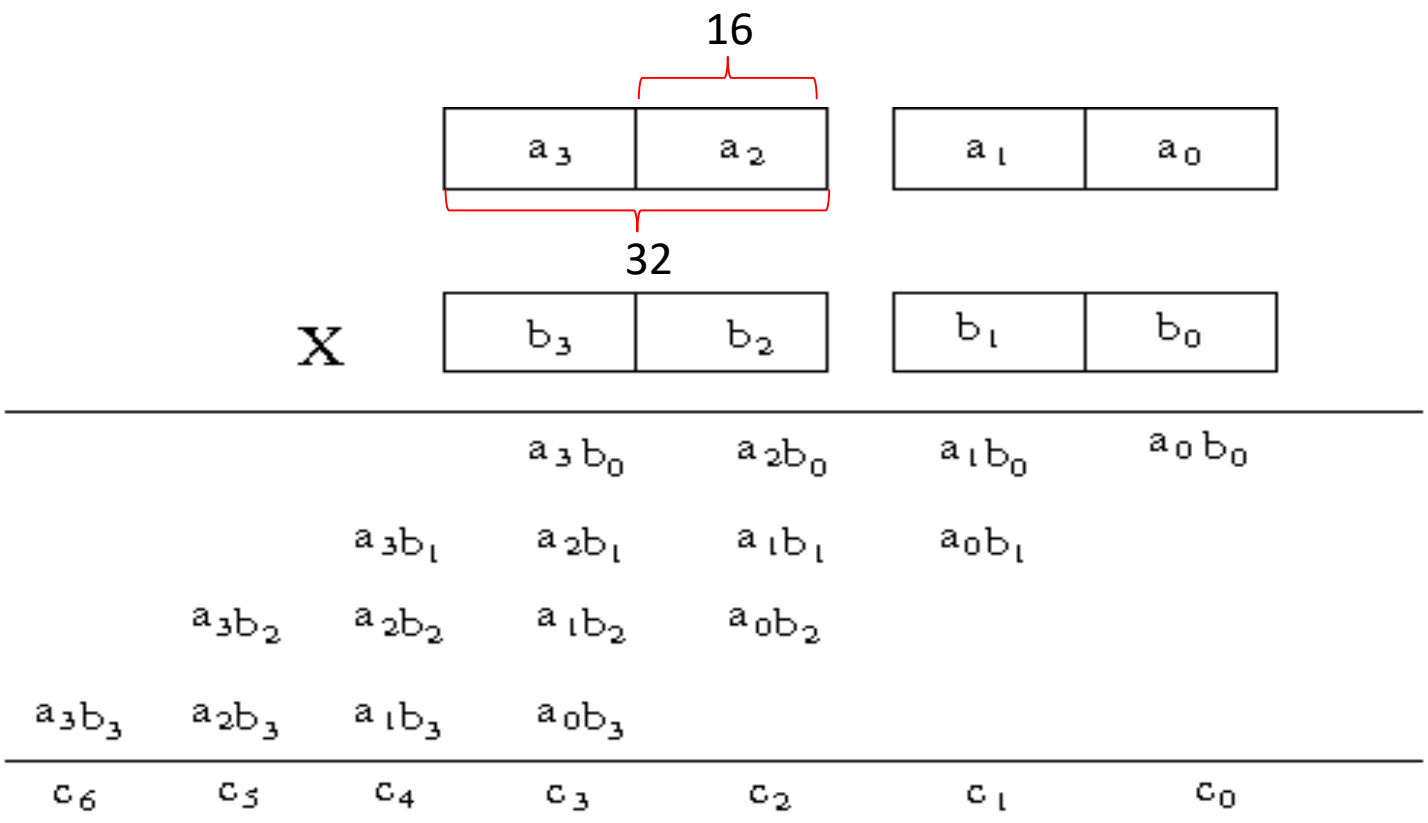
Cortex-M4: STM32F4-discovery by STMicroelectronics

- DSP instructions
- 14 registers fully available
- 192 KB of RAM

School-book multiplication using SMLA instruction

Multiplications between half words

$$\text{SMLA(B/T)(B/T)}(r^a, r^b, r^c, r^d) := r^a \leftarrow r^b_{0/1} * r^c_{0/1} + r^d$$

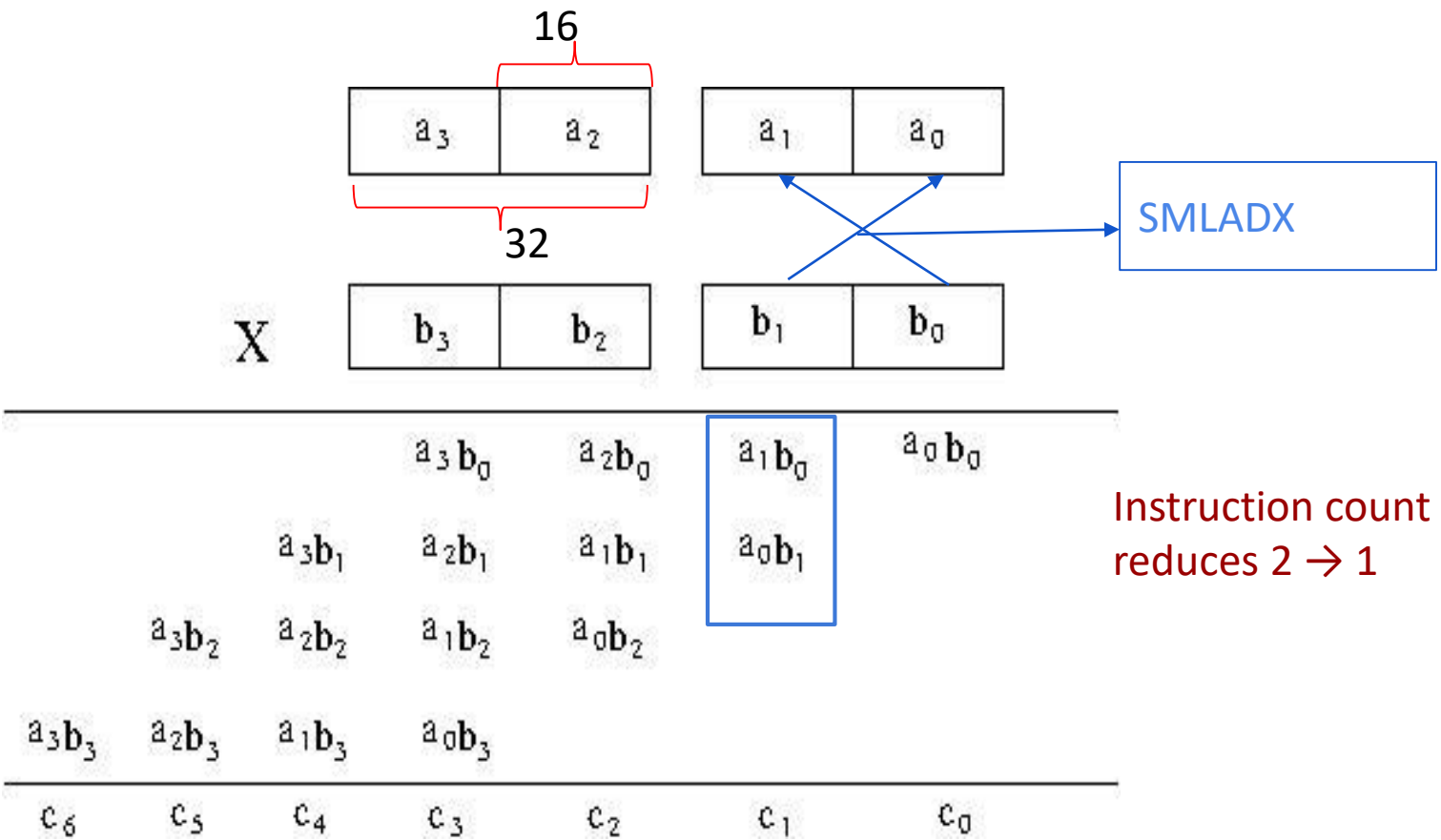


16 instructions !

School-book multiplication using SMLA instruction

DSP instruction for cross multiplication of half words

$$SMLADX(r^a, r^b, r^c, r^d) := r^a \leftarrow r^b_0 * r^c_1 + r^b_1 * r^c_0 + r^d$$

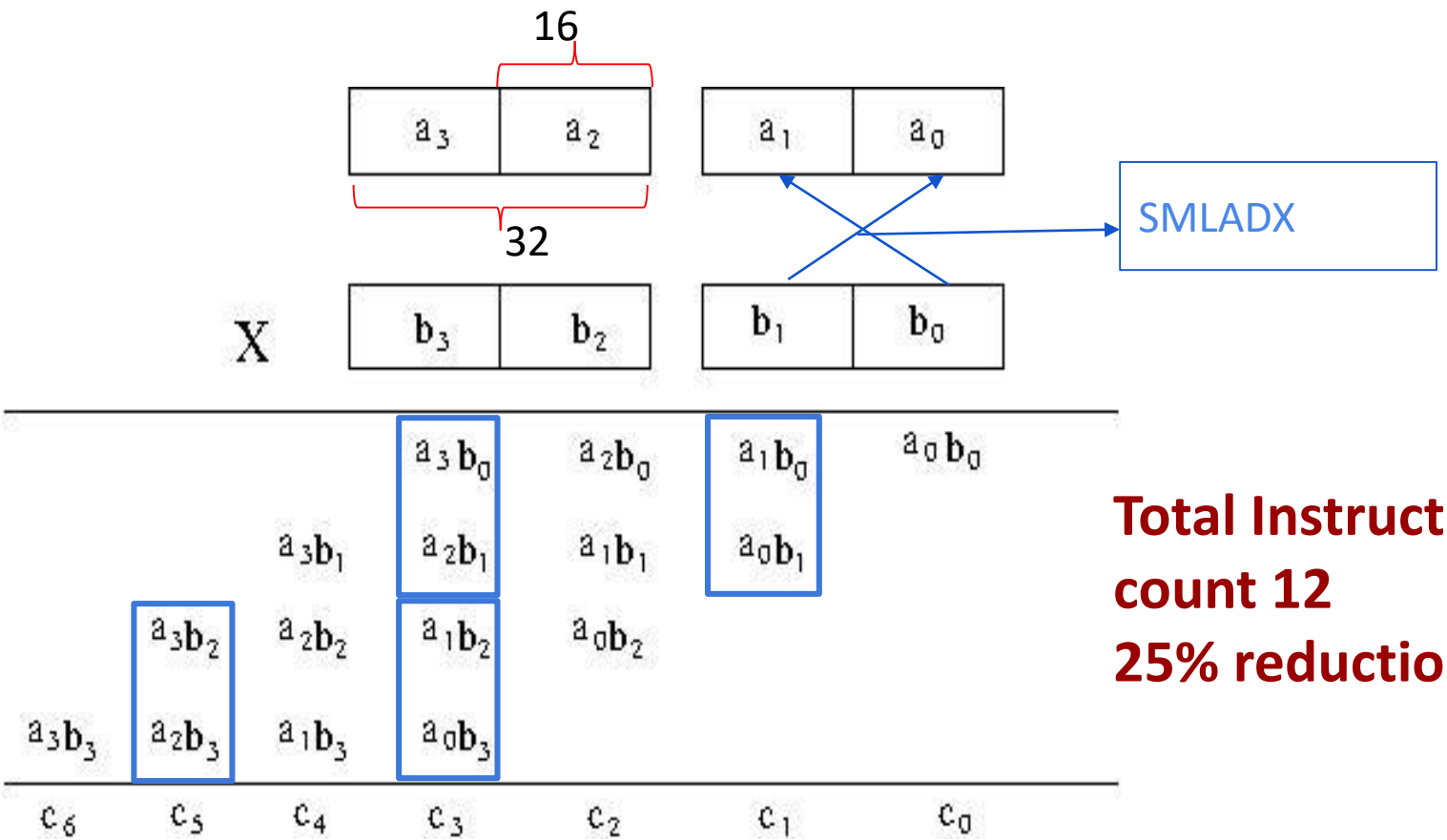


Instruction count reduces 2 → 1

School-book multiplication using SMLA instruction

DSP instruction for cross multiplication of half words

$$SMLADX(r^a, r^b, r^c, r^d) := r^a \leftarrow r^b_0 * r^c_1 + r^b_1 * r^c_0 + r^d$$



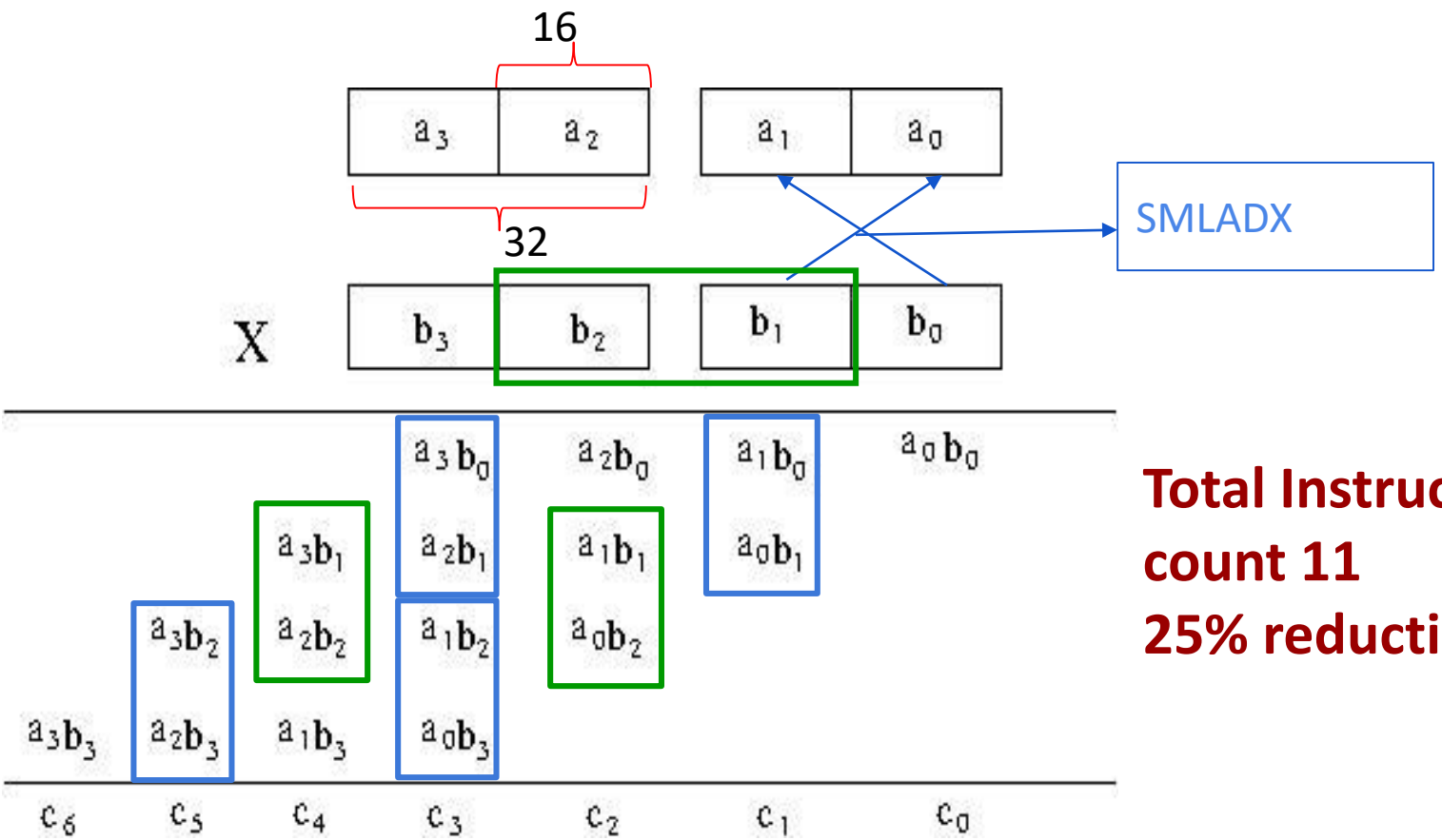
Total Instruction count 12

25% reduction

School-book multiplication using SMLA instruction

Pack non-adjacent coefficients in spare register using PKHBT

Apply SMLADX again



Total Instruction count 11

25% reduction

For 16x16 Schoolbook multiplication → 37.5% reduction overall

SABER: High-level optimization

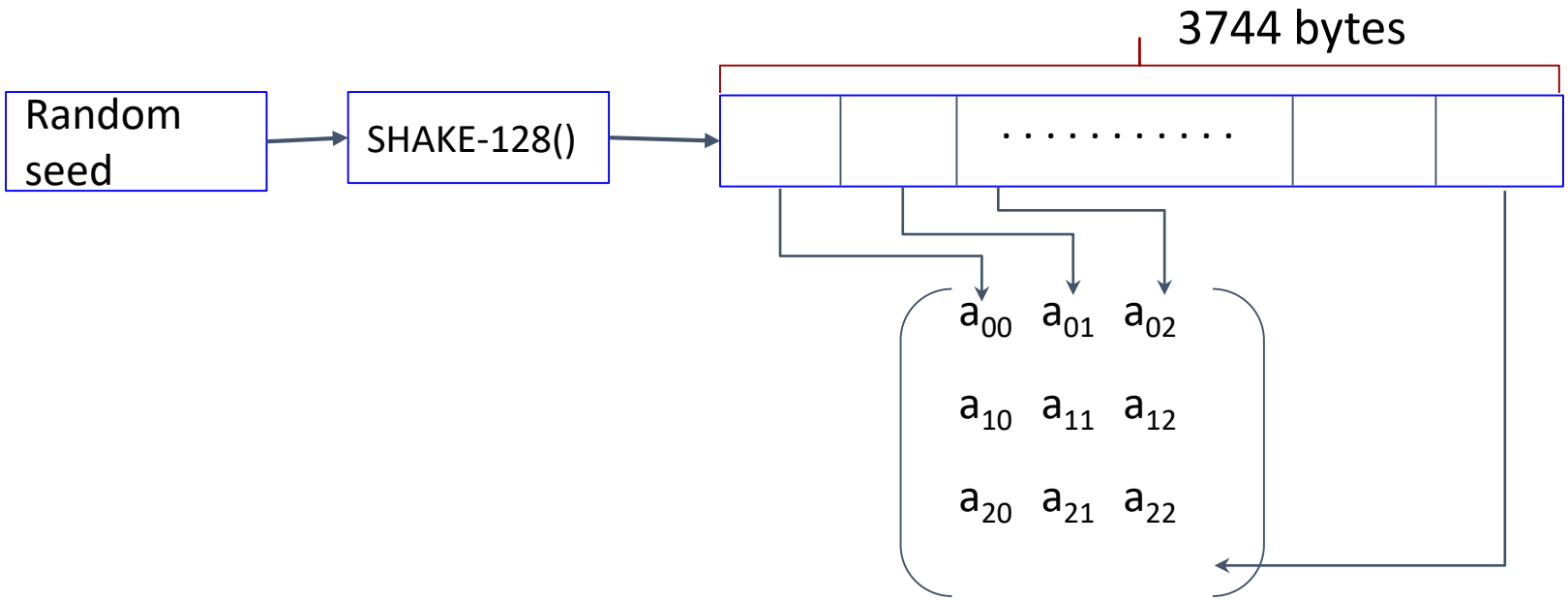
$$\left\lfloor \frac{p}{q} \begin{pmatrix} a_{0,0}(x) & \dots & a_{0,k-1}(x) \\ & \dots & \\ a_{k-1,0}(x) & \dots & a_{k-1,k-1}(x) \end{pmatrix} * \begin{pmatrix} s_0(x) \\ \dots \\ s_{k-1}(x) \end{pmatrix} \right\rceil \approx \begin{pmatrix} b_0(x) \\ \\ b_{k-1}(x) \end{pmatrix} \pmod{x^{256} + 1}$$

Matrices and vectors are generated by expanding random seed using XOF

Matrix and vector generation

- The public matrix A, secrete vectors s and s’ require large number of random bytes

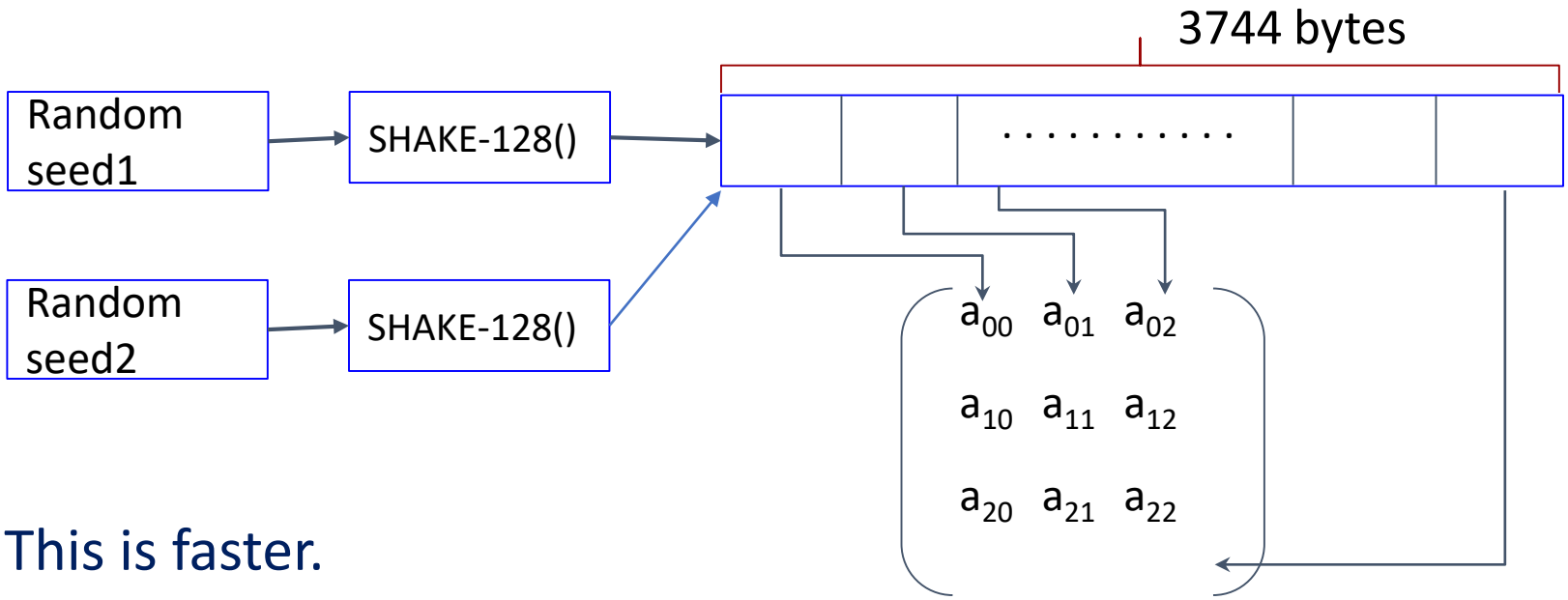
Option 1: Use single SHAKE and generate RAND sequentially



Matrix and vector generation

- The public matrix A, secrete vectors s and s' require large number of random bytes

Option 2: Use multiple SHAKE and generate RAND parally



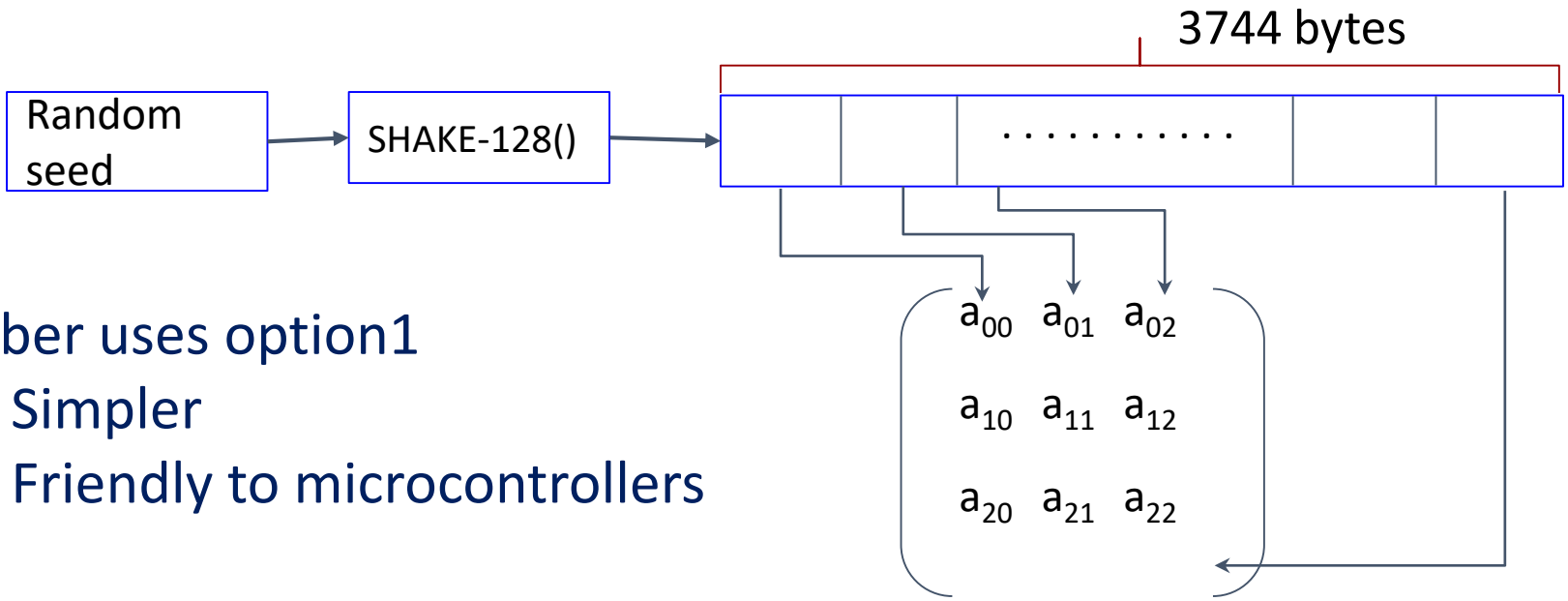
This is faster.

Kyber uses 4x parallel processing

Matrix and vector generation

- The public matrix A, secrete vectors s and s' require large number of random bytes

Option 1: Use single SHAKE and generate RAND sequentially



Saber uses option1

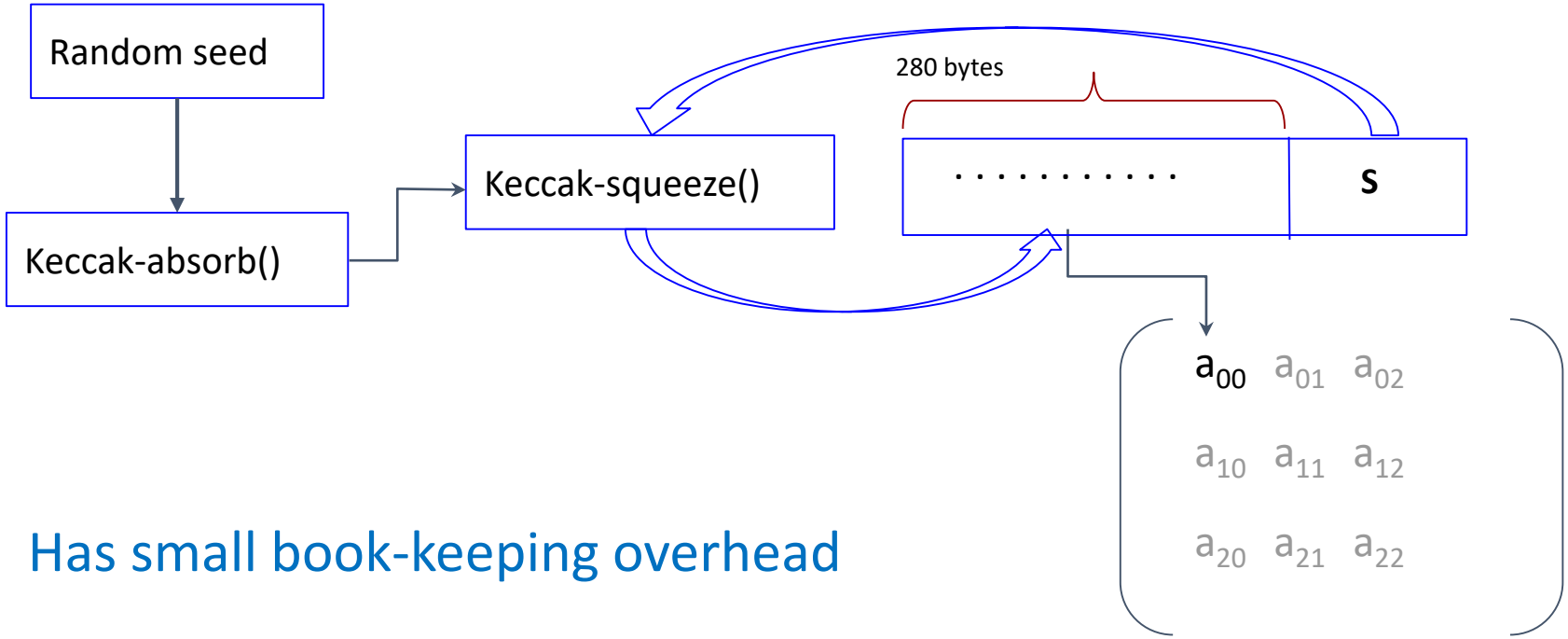
- Simpler
- Friendly to microcontrollers

- Hardware: single Keccak core consumes roughly 50% of area in LPR encryption scheme

Matrix and vector generation: memory efficient

- ‘Just in time’ approach

Cortex M0 with 8KB RAM



Has small book-keeping overhead

Matrix and vector Multiplication

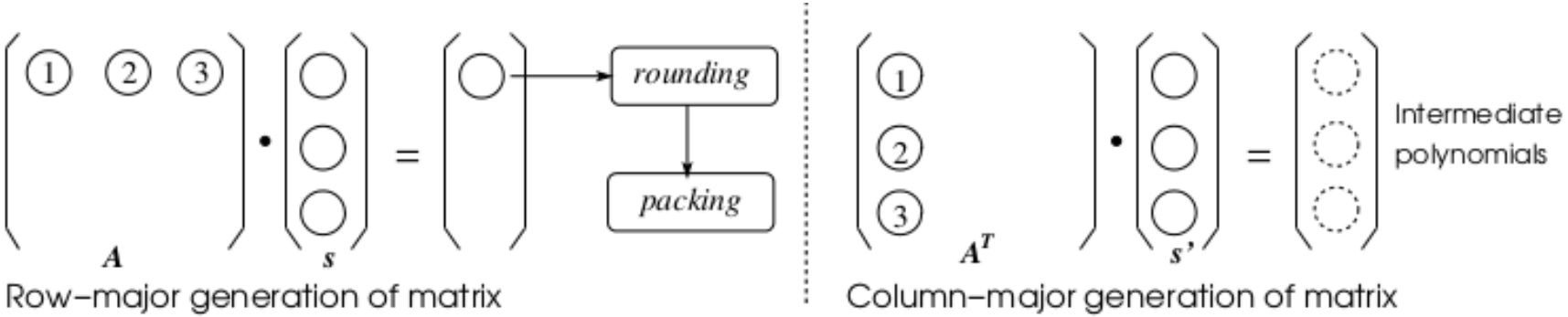
$\mathbf{A} \leftarrow \mathrm{XOF}(\mathrm{seed}_A)$

Keygen:
$\mathbf{s} \leftarrow \mathrm{XOF}(\mathrm{seed}_s)$
Computes $\mathbf{A}*\mathbf{s}$

Order of A is different in KeyGen and Enc

Encryption:
$\mathbf{s'} \leftarrow \mathrm{XOF}(\mathrm{seed}_{s'})$
Computes $\mathbf{A}^T*\mathbf{s'}$

Matrix and vector Multiplication



A is generated 'one polynomial at a time'

- Row-major has smaller memory requirement

Enc is costlier than KeyGen.
→ Use row-major for Enc and col-major for KeyGen
→ Round2 spex of Saber does this

Saber: results

- Secret key size 1344 bytes
- Public key size 992 bytes
- Ciphertext size 1088 bytes

Performance on Intel Haswell

Implementation	Key Generation	Encapsulation	Decapsulation
AVX2	104 K	122 K	120 K

Performance on ARM Cortex M (CHES 2018)

Platform	Key Generation	Encapsulation	Decapsulation	Stack
Cortex M4 (DSP)	1.1 M	1.5 M	1.6 M	8 KB
Cortex M0	4.7 M	6.3 M	7.5 M	6 KB

Work in progress: hardware implementation of Saber

Thank you