25 Years of Cryptographic Hardware Design

Çetin Kaya Koç

City University of Istanbul &
University of California Santa Barbara

koc@cs.ucsb.edu
http://cryptocode.net
koc@cryptocode.net

25 AÑOS DE LA COMPUTACIÓN EN EL CINVESTAV

25 Years of Cryptographic Hardware Design

- 1975-1977: Invention of Public-Key Cryptography
- Diffie-Hellman & RSA Algorithms
- Publication Dates: Nov 1976 & Feb 1978
- First hardware implementation:

  R. L. Rivest. A Description of a Single-Chip Implementation of the RSA Cipher. *Lambda*, vol. 1, pages 14-18, 1980.
- In 1984, I was a graduate student at UCSB's ECE Department
- My interest started with Rivest's hardware paper

Essential Milestones

- This talk gives a brief summary of advanced algorithms for creating better hardware realizations of public-key cryptographic algorithms: Diffie-Hellman, RSA, elliptic curve cryptography

- Essential milestones:
  - Naive algorithms, 1978-1985
  - Montgomery algorithm, 1985
  - Advanced Karatsuba algorithms, 1994
  - Advanced Montgomery algorithms, 1996
  - Montgomery algorithm in $GF(2^k)$, 1998
  - Unified arithmetic, 2002
  - Spectral arithmetic, 2006

RSA Computation

- The RSA algorithm uses modular exponentiation for encryption

$$C := M^e \pmod{n}$$

and decryption

$$M : C^d \pmod{n}$$

- The computation of $M^e \bmod n$ is performed using exponentiation heuristics

- Modular exponentiation requires implementation of three basic modular arithmetic operations: addition, subtraction, and multiplication

Diffie-Hellman Computation

- Similarly, the Diffie-Hellman key exchange algorithm executes the steps

$$\begin{aligned} R_A &:= g^a \pmod{p} \\ R_B &:= g^b \pmod{p} \\ R'_B &:= R_A^b = g^{ab} \pmod{p} \\ R'_A &:= R_B^a = g^{ba} \pmod{p} \end{aligned}$$

  between two parties, Alice & Bob

- These computations are also modular exponentiations, requiring modular addition, subtraction, and multiplication operations

NIST Digital Signature Algorithm

- The signature computation on M and k is the pair (r, s)

$$
\begin{aligned}
r &:= (g^k \bmod p) \bmod q \\
s &:= (M + xr)k^{-1} \bmod q
\end{aligned}
$$

- The signature verification

$$
\begin{aligned}
w &:= s^{-1} \bmod q \\
u_1 &:= Mw \bmod q \\
u_2 &:= rw \bmod q \\
v &:= (g^{u_1} y^{u_2} \bmod p) \bmod q \\
\text{Check if } r &= v
\end{aligned}
$$

Ellliptic Curve Cryptography

- Elliptic curves defined over $GF(p)$ or $GF(2^k)$ are used in cryptography
- The arithmetic of $GF(p)$ is the usual mod p arithmetic
- The arithmetic of $GF(2^k)$ is similar to that of $GF(p)$, however, there are some differences
- Elliptic curves over $GF(2^k)$ are more popular due to the space and time-efficient algorithms for doing arithmetic in $GF(2^k)$
- Elliptic curve cryptosystems based on discrete logarithms seem to provide similar amount of security to that of RSA, but with relatively shorter key sizes

Computations of Cryptographic Functions

- It is interesting to note that all public-key cryptographic algorithms are based on number-theoretic and algebraic finite structures, such as groups, rings, and fields

- In fact, most of them need modular arithmetic, i.e., the arithmetic of integers in finite rings or fields

- The challenge is however that the sizes of operands are large, starting from about 160 bits up to 16,000 bits

- Therefore, the algorithmic development of cryptographic hardware design is essentially based on (exact) computer arithmetic with very large integers

- Since exponentiations & multiplications are most time/energy/space consuming computations, we will only study those in our talk

Computing Exponentiations

- Given the integer e, the computation of M^e or eP is an exponentiation operation

- The objective is to use as few multiplications (or elliptic curve additions) as possible for a given integer e

- This problem is related to *addition chains*

- An addition chain yields an algorithm for computing M^e or eP given the integer e

$$M^1 \rightarrow M^2 \rightarrow M^3 \rightarrow M^5 \rightarrow M^{10} \rightarrow M^{11} \rightarrow M^{22} \rightarrow M^{44} \rightarrow M^{55}$$

$$P \rightarrow 2P \rightarrow 3P \rightarrow 5P \rightarrow 10P \rightarrow 11P \rightarrow 22P \rightarrow 44P \rightarrow 55P$$

Computing Exponentiations

- Finding the shortest addition chain is an NP-complete problem

- Lower bound: $\log_2 e + \log_2 H(e) - 2.13$ (*Schönhage*)

- Upper bound: $\lfloor \log_2 e \rfloor + H(e) - 1$, where $H(e)$ is the Hamming weight of e (the binary method, the SX method, *Knuth*)

- It turns out the oldest known algorithm for computing exponentiation is not too far in efficiency to the best algorithm

- Heuristics, m-ary, adaptive m-ary, sliding windows, power tree methods offer only slight improvements

Computing Modular Multiplication - Naive Algorithms

- Given $a, b < n$, compute $P = a \cdot b \bmod n$
- Multiply and reduce:

  Multiply: $P' = a \cdot b$ ($2k$-bit number)

  Reduce: $P = P' \bmod n$ (k-bit number)
- Reductions are essentially integer divisions
- However, multiply and reduce steps can be interleaved, but offering only slight improvements

Interleaved Multiply & Reduce - Naive Algorithms

$$
\begin{aligned}
P' &= a \cdot b = a \cdot \sum_{i=0}^{k-1} b_i 2^i = \sum_{i=0}^{k-1} (a \cdot b_i) 2^i \\
&= 2(\cdots 2(2(0 + a \cdot b_{k-1}) + a \cdot b_{k-2}) + \cdots) + a \cdot b_0
\end{aligned}
$$

1. $P := 0$
2. **for** $i = k - 1$ **downto** 0
2a. $P := 2P + a \cdot b_i$
2b. $P := P \bmod n$
3. **return** P

- Unfortunately, Step 2b is highly time consuming (a full division for every bit of the operands)

Montgomery Multiplication - 1985

- Attempts to create good hardware to compute the RSA functions (sign, verify, encrypt, decrypt) in acceptable time have essentially failed because of the excessive requirements of the naive algorithms

- This includes Rivest's hardware proposal and all other implementations until the Montgomery multiplication algorithm came about

- Peter Montgomery discovered a method to replace Step 2b with a step similar to Step 2a: an addition instead of a division

- It is brilliant and efficient

- Montgomery's algorithm changed cryptographic design in a way very much like the FFT algorithm changed the digital signal processing

Montgomery Multiplication

- Montgomery's method maps the integers $\{0, 1, 2, \ldots, n-1\}$ to the same set with the map $\bar{x} = x \cdot r \pmod{n}$ using the integer $r = 2^k$

- It then works in this set (numbers with the "bar" sign) and performs the multiplication

$$\text{MonPro}(\bar{a}, \bar{b}) = \bar{a} \cdot \bar{b} \cdot 2^{-k} \pmod{n}$$

- The above operation turns out to be significantly simpler than the standard modular multiplication $a \cdot b \pmod{n}$ because the division by n in Step 2b (reduction) is avoided

- Transformation to and back from the "bar" domain is also quite easily done, i.e., $\bar{x} = \text{MonPro}(x, r^2)$ and $x = \text{MonPro}(\bar{x}, 1)$

Montgomery Multiplication

- In order to compute $u = \text{MonPro}(a, b) = a \cdot b \cdot 2^{-k} \pmod{n}$, we use the steps below

1. $u := 0$
2. **for** $i = 0$ **to** $k - 1$
2a. $u := u + a_i \cdot b$
2b. **if** u_0 is 1 **then** $u := u + n$
3. $u := u/2$

- Now, Step 2b is only an addition!

- And, it is is done about half of the time!

- We remain in the Montgomery ("bar") domain of integers until the final step of the exponentiation, and then use the conversion routine to go back to the "no bar" domain

Karatsuba-Ofman Multiplication

- Algorithms Textbooks offer a few asymptotically faster multiplication algorithms: Karatsuba-Ofman, Toom-Cook, Winograd, and DFT-based algorithms

- These algorithms are all good: they help you to multiply faster

- But, they are no help in modular multiplication, i.e., they do not multiply-and-reduce (Montgomery's method is special)

- They also have large overhead, and start being faster only after a few thousand bits

- However, there has been significant algorithmic developments to bring down their break-even point to a few hundred bits

Advanced Montgomery Multiplication

- On the other hand, Montgomery algorithms also improved
- They can be made fit into specific archiectures, by changing the way they scan the bits of the multiplicand, the multiplier, and the product
- Separated Operand Scanning (SOS): First computes $t = a \cdot b$ and then interleaves the computations of $m = t \cdot n' \bmod r$ and $u = (t + m \cdot n)/r$. Squaring can be optimized.

  SOS requires $2s + 2$ words of space
- Finely Integrated Product Scanning (FIPS): Interleaves computation of $a \cdot b$ and $m \cdot n$ by scanning the words of m

  It uses the same space to keep m and u, reducing the temporary space to $s + 3$ words

Advanced Montgomery Multiplication

- Finely Integrated Operand Scanning (FIOS): The computation of $a \cdot b$ and $m \cdot n$ is performed in a single loop

  FIOS also requires $s + 3$ words of space

- Coarsely Integrated Hybrid Scanning (CIHS): The computation of $a \cdot b$ is split into 2 loops, and the second loop is interleaved with the computation of $m \cdot n$

  CIHS also requires $s + 3$ words of space

- Coarsely Integrated Operand Scanning(CIOS): Improves the SOS method by integrating the multiplication and reduction steps. It alternates between iterations of the outer loops for multiplication and reduction

  CIOS also requires $s + 3$ words of space

Advanced Montgomery Multiplication

- All methods require $2s^2 + s$ multiplications
- Add, Read/Write and Space requirements are below

	Add	Read/Write	Space
SOS	$4s^2+4s+2$	$8s^2+13s+5$	$2s+2$
FIPS	$6s^2+2s+2$	$14s^2+16s+3$	$s+3$
FIOS	$5s^2+3s+2$	$10s^2+9s+3$	$s+3$
CIHS	$4s^2+4s+2$	$9.5s^2+11.5s+3$	$s+3$
CIOS	$4s^2+4s+2$	$8s^2+12s+3$	$s+3$

- Depending on the availability of functional units (multipliers, adders, registers), one method can outperform another and thus should be selected

Montgomery Multiplication in $GF(2^k)$

- It turns out that the Montgomery multiplication can also be performed in the finite field $GF(2^k)$ if the polynomial basis representations of the field elements are employed

- It imitates the the Montgomery multiplication in $GF(p)$ by taking the modulus the irreducible polynomial $p(x)$ generating the field of 2^k elements

- It is not as fast as the normal basis, but it has some advantages

Montgomery Multiplication in $GF(2^k)$

- In order to compute

$$u(x) = \text{MonPro}(a(x), b(x)) = a(x) \cdot b(x) \cdot x^{-k} \bmod p(x) \ ,$$

we use the steps below

1. $u(x) := 0$
2. **for** $i = 0$ **to** $k-1$

2a. $u(x) := u(x) + a_i \cdot b(x) \bmod 2$

2b. **if** u_0 is 1 **then** $u(x) := u(x) + p(x) \bmod 2$

3. $u := u/2$

- Now Steps 2a and 2b use mod 2 additions (XOR gates)

Unified Arithmetic

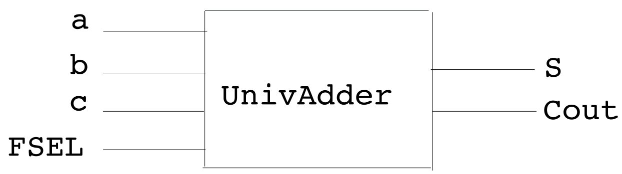
- One advantage of the Montgomery multiplication in $GF(2^k)$ is that a single arithmetic unit can be used to handle both kinds of fields: $GF(p)$ and $GF(2^k)$: This is called unified arithmetic (or, dual-field arithmetic)

- Advantages of the unified arithmetic are low manufacturing cost, compatibility, parallelism, and *scalability*

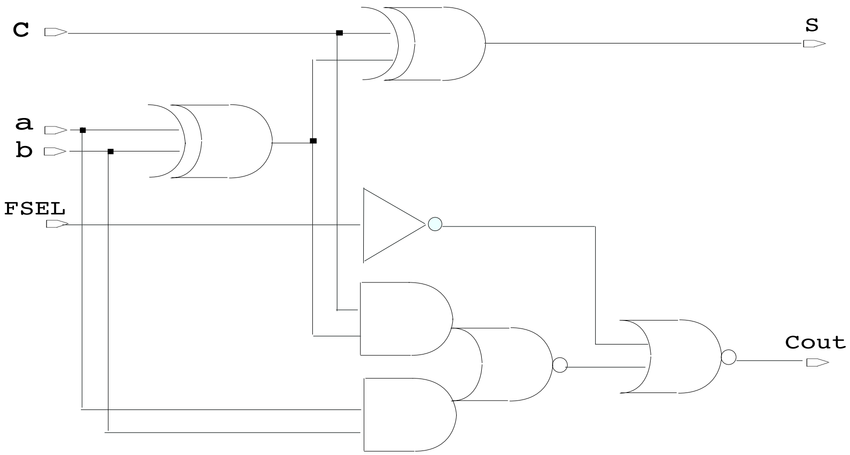
- Furthermore, unified arithmetic is impartial: it does not favor one prime against another or one irreducible polynomial against another

- The building block of the unified architecture is the unified full adder: a 1-bit adder that handles both $GF(p)$ and $GF(2^k)$

Unified Full Adder



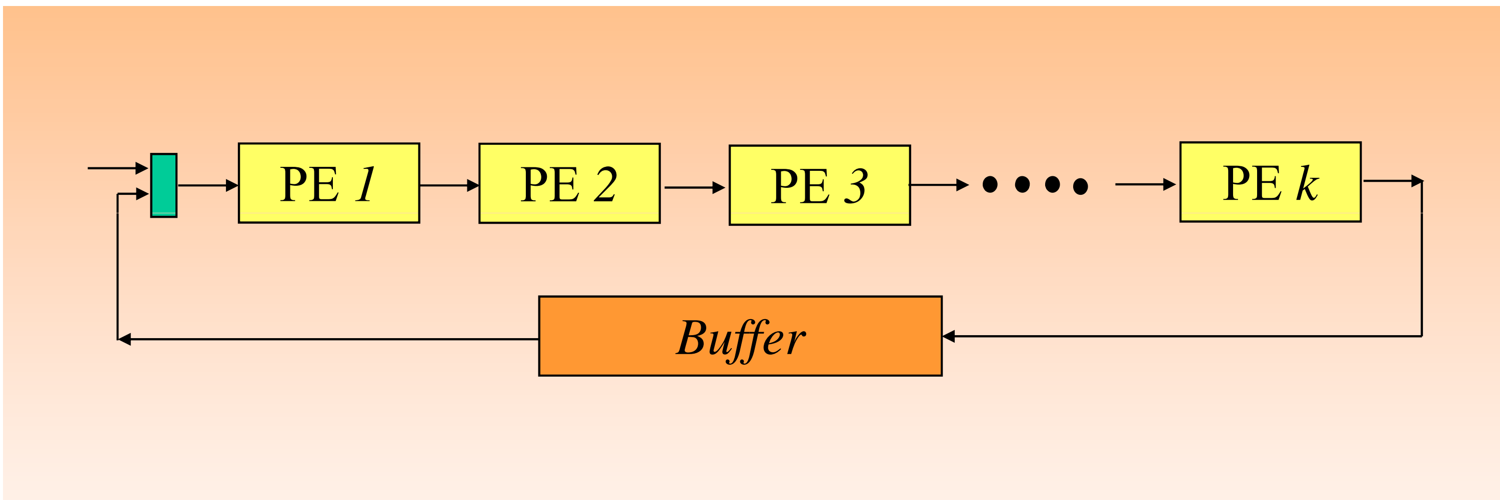
(a) Universal Adder



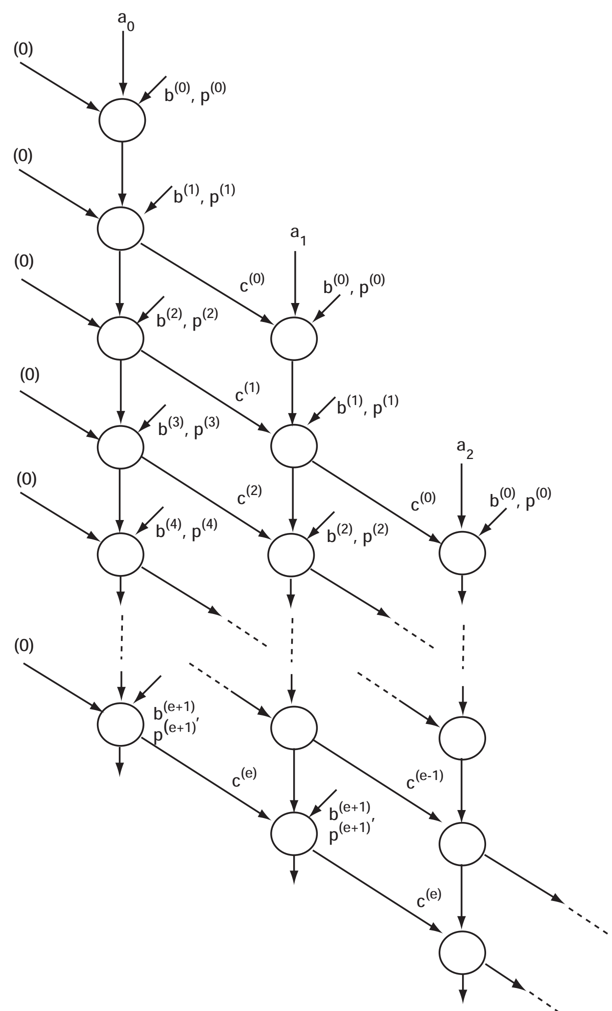
(b) Synthesized circuit by Mentor

Scalability

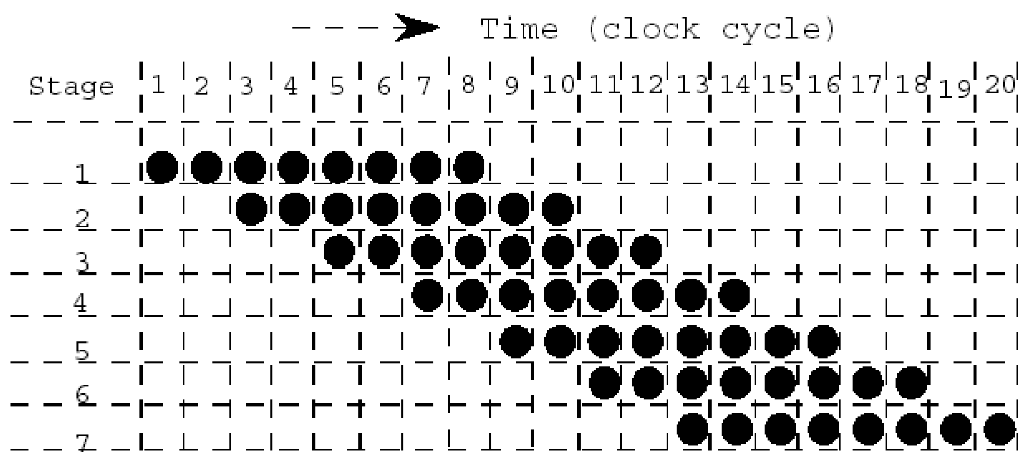
- *Scalability* is an important concept: it allows to make small changes in the hardware to handle larger operands without a complete redesign (such as switching from 1024-bit RSA keys to 1536-bit RSA keys)



Dependency Graph of Montgomery Multiplication

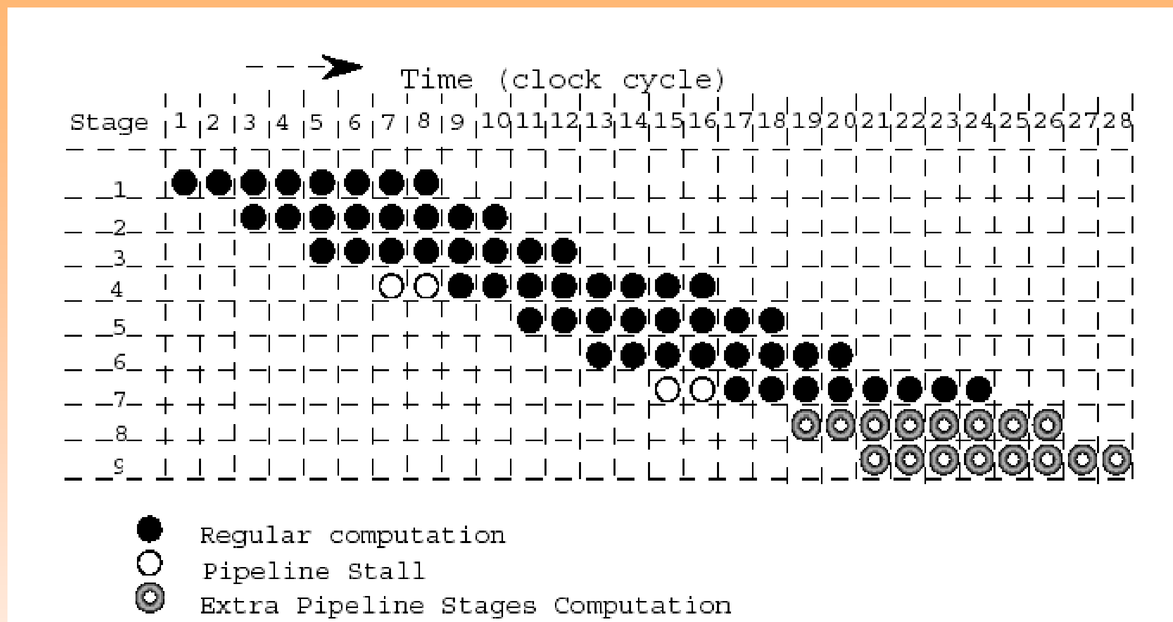


Pipelined Montgomery Multiplication



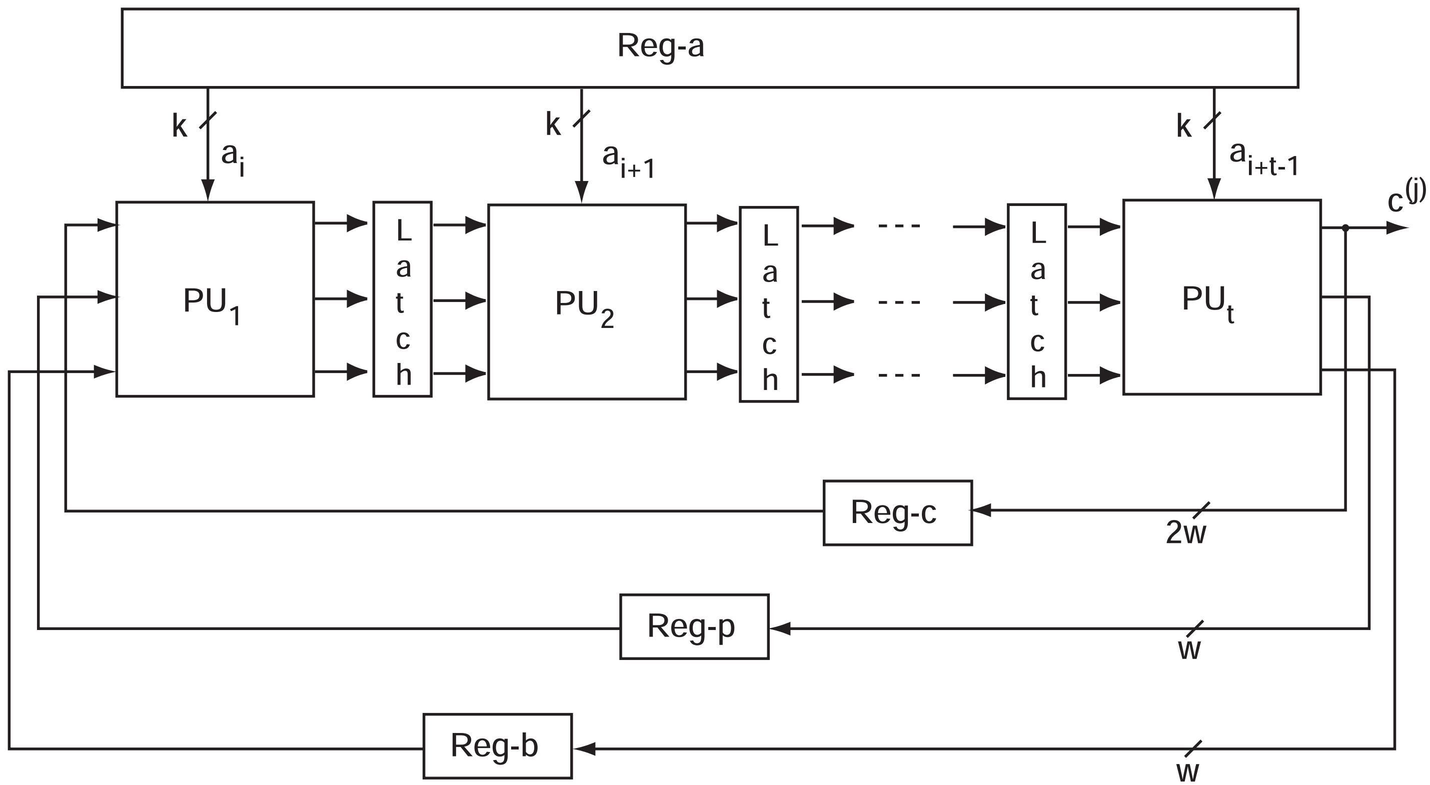
An example of pipeline computation for 7 bit operands
where w=1

Pipelined Architecture with Fewer Units



Pipeline stalls when fewer processing units are available
m=7, w=1, k=3

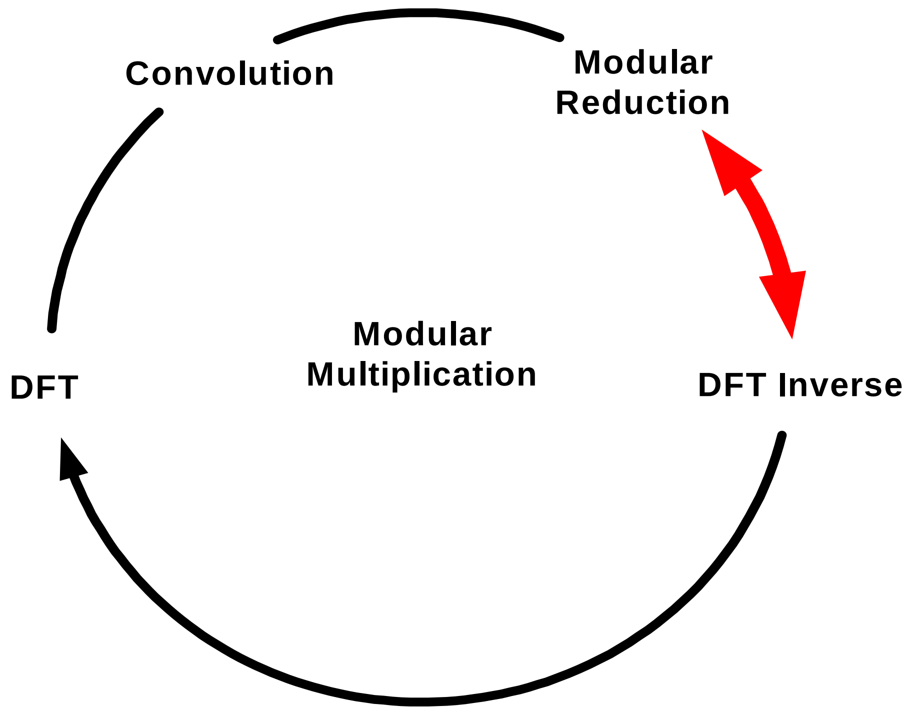
General Pipelined Architecture



Spectral Arithmetic

- We use FFT-based arithmetic to implement *modular multiplication*
- However, we are interested in performing the reduction inside the spectral (frequency) domain
- We utilize finite ring and field arithmetic (avoid real or complex arithmetic because of the roundoff errors in using floating-point or fixed-point arithmetic)
- We also want to bring down the break-even point of efficiency for FFT-based multiplication
- Furthermore, we utilize the properties of the DFT and Montgomery algorithm to perform *modular multiplication*

Spectral Arithmetic



DFT over a Finite Ring: Definition

Let ω be a primitive d-th root of unity in $\mathbb{Z}_q$ and, let $x(t)$ and $X(t)$ be polynomials of degree $d-1$ having entries in $\mathbb{Z}_q$. The DFT map over $\mathbb{Z}_q$ is an invertib le set map sending $x(t)$ to $X(t)$ given by

$$X_i = DFT_d^{\omega}(x(t)) := \sum_{j=0}^{d-1} x_j \omega^{ij} \bmod q,$$

with the inverse

$$x_i = IDFT_d^{\omega}(X(t)) := d^{-1} \cdot \sum_{j=0}^{d-1} X_j \omega^{-ij} \bmod q,$$

for $i, j = 0, 1, \ldots, d-1$.

DFT over a Finite Ring: Existence

We write

$$x(t) \quad \overset{\text{DFT}}{\longleftrightarrow} \quad X(t)$$

and say $x(t)$ and $X(t)$ are transform pairs; $x(t)$ is called a **time polynomial** and sometimes $X(t)$ is named as the **spectrum** of $x(t)$.

- (Convention) In the literature, DFT over a finite ring spectrum is also called as Number Theoretical Transform (NTT)

- (Existence) In order to have a DFT map over $\mathbb{Z}_q$:
  - The multiplicative inverse of DFT length d must exist in $\mathbb{Z}_q$ which requires that $\gcd(d, q) = 1$.
  - d has to divide $p - 1$ for every prime p divisor of q

DFT over a Finite Ring: Efficiency

In order to have simple arithmetic

- q should be chosen as
  a Mersenne number $q = 2^v - 1$, or
  a Fermat number $q = 2^v + 1$

- The principal root of unity ω should be selected as a power of 2 to simplify the multiplications with roots of unity

Properties of DFT

- Under certain conditions, the Fourier transform preserves some properties of the time sequences, e.g., linearity and convolution.

- The existence conditions of these properties differ when working in finite ring spectrums

- Let ϕ and Φ be operations on time and spectral domains respectively. We write

$$\phi \quad \overset{\text{DFT}}{\longleftrightarrow} \quad \Phi$$

  and say ϕ and Φ are transform pairs on $x(t)$ and sometimes declare that **the map DFT_d^ω respects the operation** ϕ on point $x(t)$ if following equation is satisfied

$$\phi(x(t)) = IDFT_d^\omega \circ \Phi \circ DFT_d^\omega(x(t))$$

Time-Frequency Dictionary

- **Time and frequency shifts** correspond to circular shifts Let

$$x(t) = x_0 + x_1 t + \ldots + x_{d-1} t^{d-1}$$

and

$$X(t) = X_0 + X_1 t + \ldots + X_{d-1} t^{d-1}$$

be a transform pair.

The *one-term right circular shift* is defined as $x(t) \circlearrowleft 1$

$$x_1 + x_2 t + \ldots + x_{d-2} t^{d-1} + x_0 t^{d-1}$$
$$\Big\updownarrow \text{DFT}$$
$$X(t) \odot \Gamma(t)$$

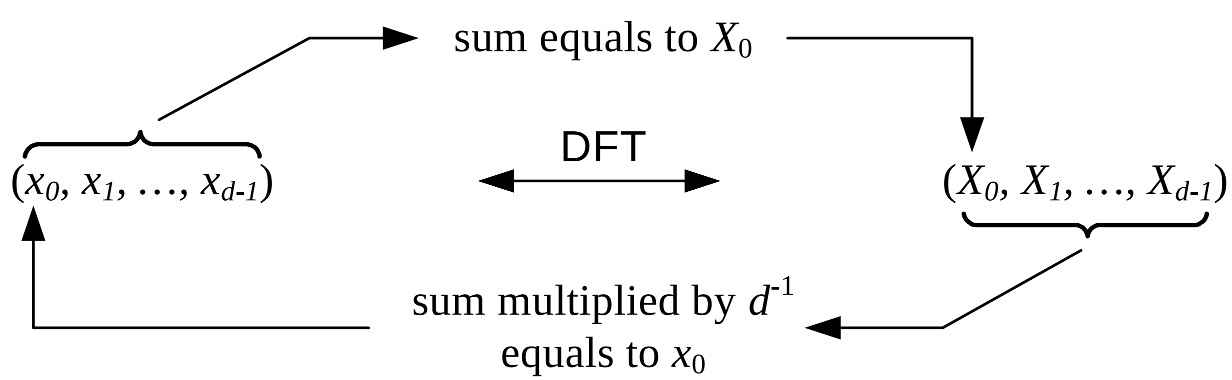
where $\odot$ stands for component-wise multiplication and

$$\Gamma(t) = 1 + \omega^{-1} t + \ldots + \omega^{-(d-1)} t^{d-1}$$

Time-Frequency Dictionary

- **Sum of sequence and first value:** The sum of the coefficients of a time polynomial equals to the zeroth coefficient of its spectral polynomial. Conversely the sum of the spectrum coefficients equals to d^{-1} times the zeroth coefficient of the time polynomial

$$x_0 = d^{-1} \cdot \sum_{i=0}^{d-1} X_i \omega^{-i} \quad \text{and} \quad X_0 = \sum_{i=0}^{d-1} x_i \omega^i$$



Time-Frequency Dictionary

- **Left and right logical shifts:** By using the previous properties, it is possible to perform logical left and right digit shifts $x(t) \ll 1$ as follows:

$$
\begin{array}{c}
(x(t) - x_0)/t = x_1 + \ldots + x_{d-1}t^{d-2} \\
\updownarrow \text{DFT} \\
(X(t) - x_0(t)) \odot \Gamma(t)
\end{array}
$$

where

$$x_0(t) = x_0 + x_0 t + x_0 t^2 + \ldots + x_0 t^{d-1}$$

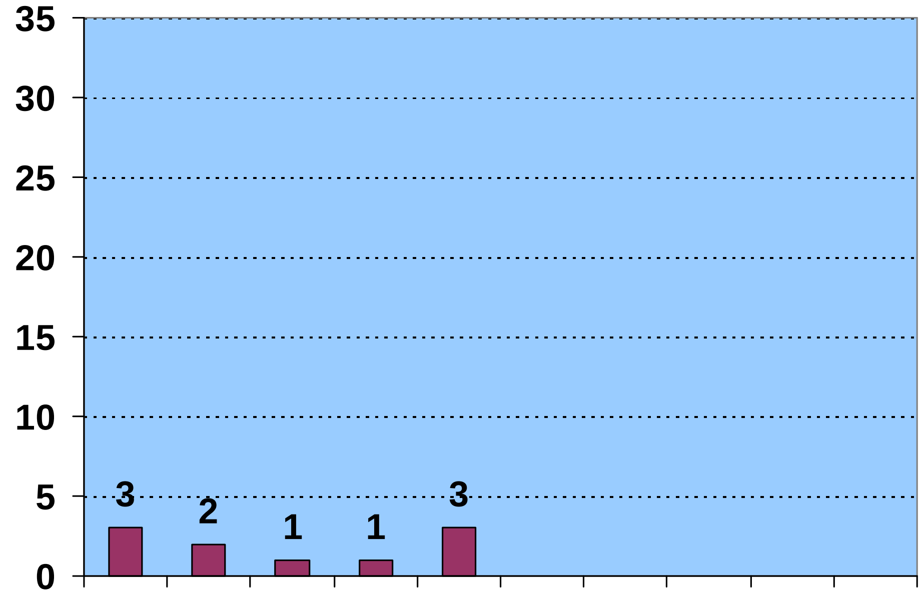
- The right shifts are similar, where one then uses the

$$\Omega(t) = 1 + \omega^1 t + \ldots + \omega^{(d-1)} t^{d-1}$$

polynomial instead of $\Gamma(t)$

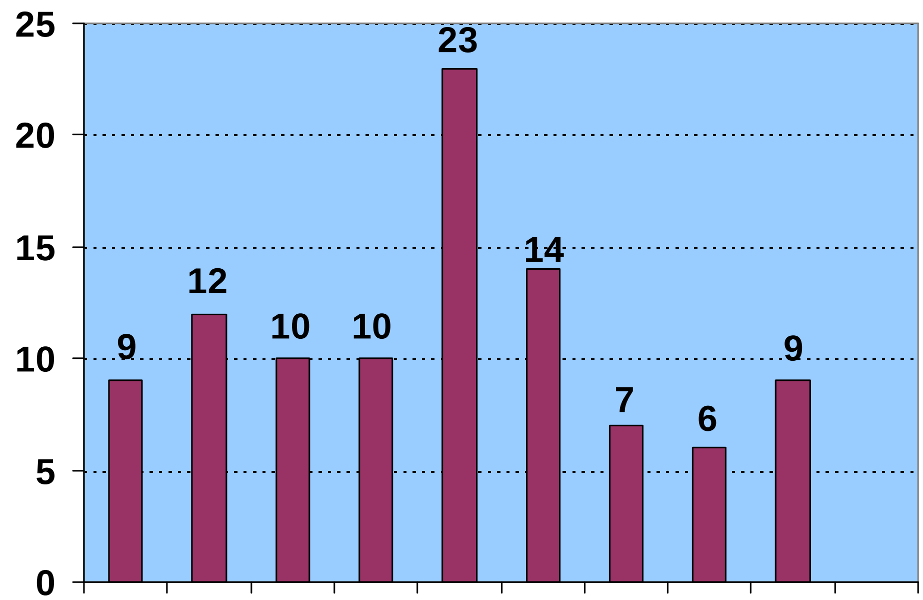
A Time Simulation for Spectral Modular Multiplication

We would like to compute $859^2 \cdot 4^{-9} \pmod{1337}$.
Signal $x(t)$ representing $859 = x(4)$ in base 4.



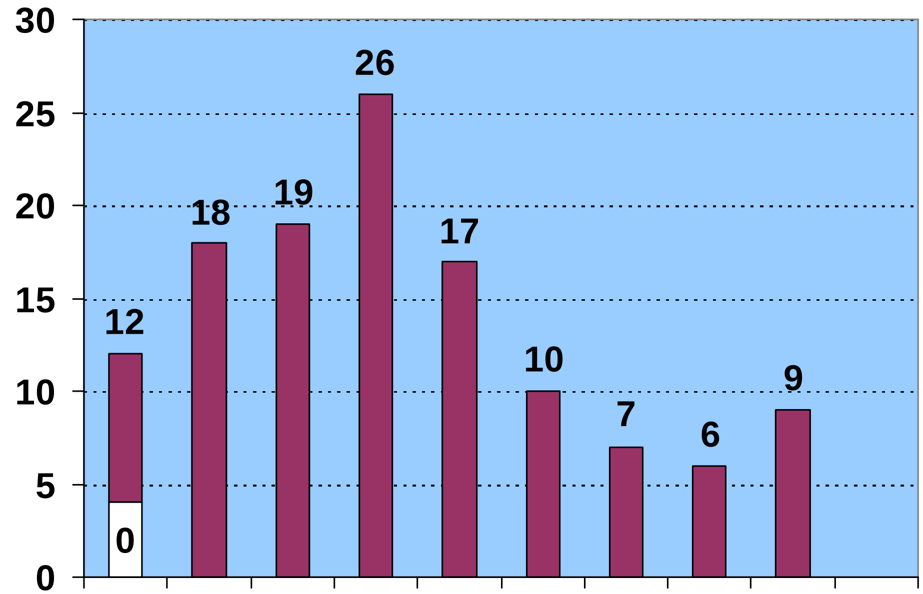
A Time Simulation for SMP

Convolving $x(t)$ with itself, we find $x^2(t) = 859^2 = 737881$.



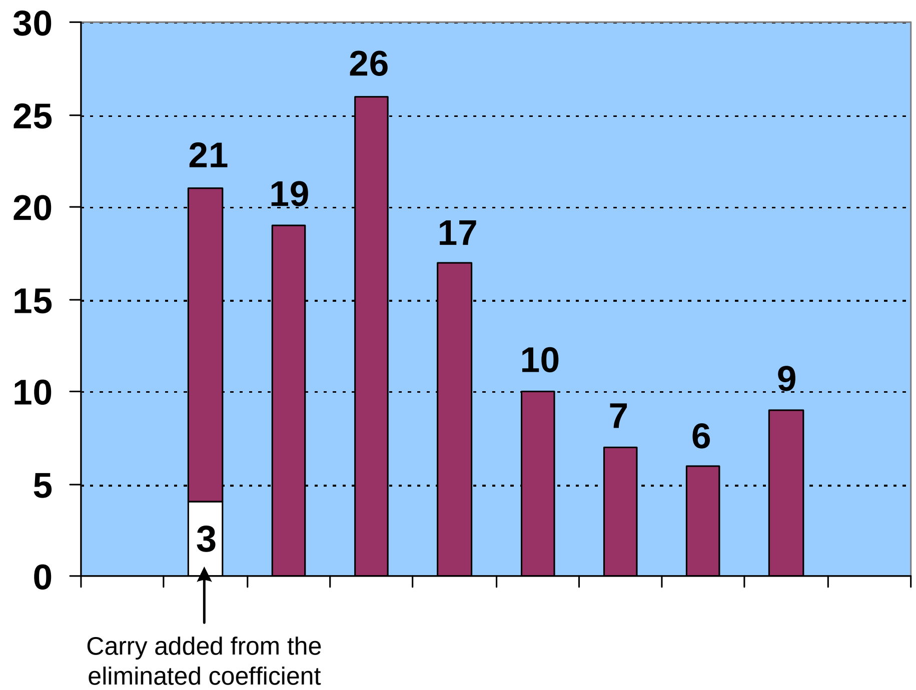
A Time Simulation for SMP

The modulus $m = 1337$ is represented as $m = 1 + 2t + 3t^2 + t^4 + t^5$.We add $3m$ to the sum to anhilate the least significant b bits of the least digit.



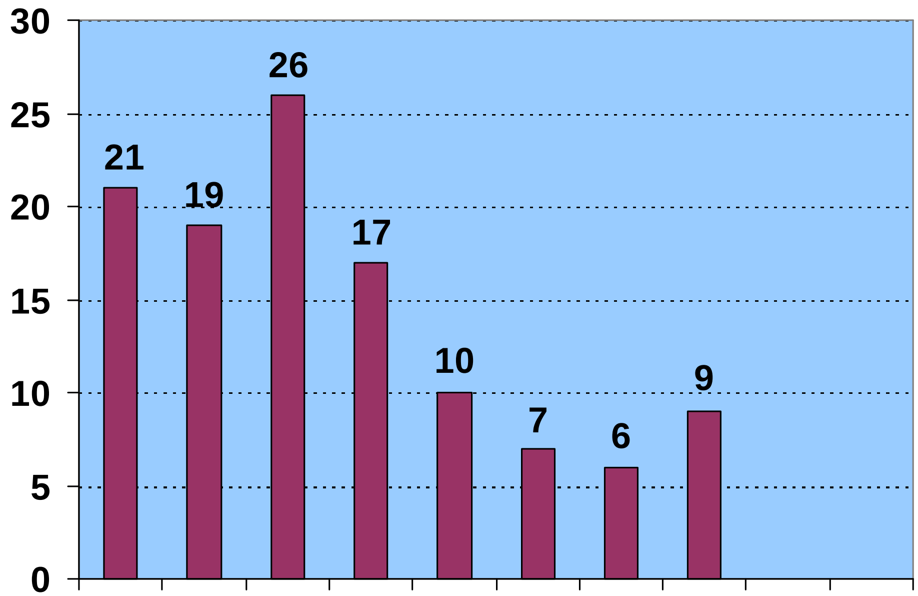
A Time Simulation for SMP

Carry goes to the next digit.



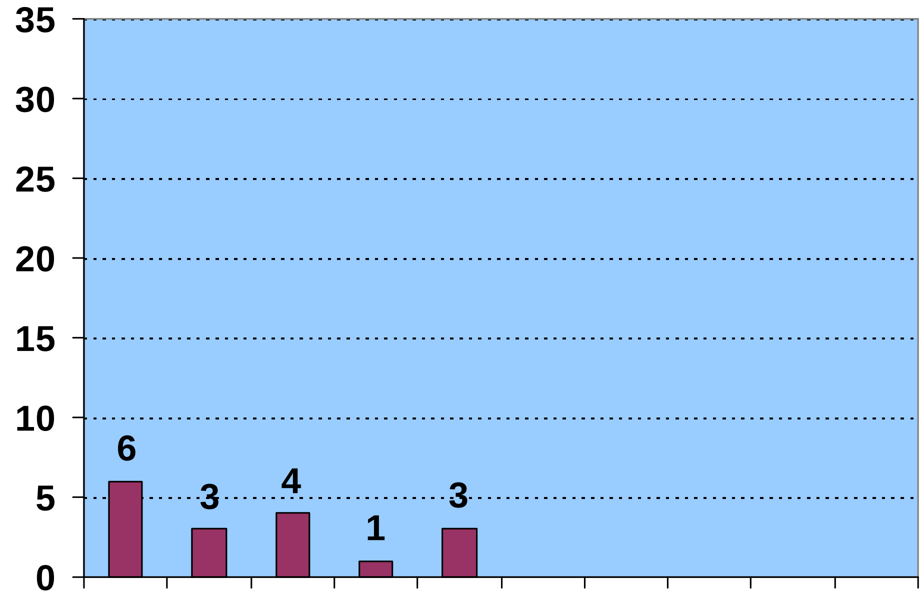
A Time Simulation for SMP

We then shift the digits.



A Time Simulation for SMP

After 9 iterations, we find the result: $914 \equiv 859^2 \cdot 4^{-9} \pmod{1337}$.



Unending Quest for Efficiency

- Conclusions?

- Challenges remain: Make faster but low-area and low-energy hardware for cryptography

- Platforms are diverse: Huge SSL and IPSec boxes versus tiny Bluetooth earphones, cellphones and PDAs

- New challenges: We need to build countermeasures in order to circumvent attacks by adversaries to obtain hardware-hidden secrets

- Questions?

  Email: koc@cryptocode.net