30 Years of Cryptographic Hardware Design

Çetin Kaya Koç

Department of Computer Science
University of California Santa Barbara

koc@cs.ucsb.edu
1976-1978: Invention of public-key cryptography:


1980: First hardware implementation:

Articles

14 A Description of a Single-Chip Implementation of the RSA Cipher
Ronald L. Rivest, Massachusetts Institute of Technology
This public-key encryption/decryption chip features a 512-bit ALU.

22 Tools for Verifying Integrated Circuit Designs
Clark M. Baker and Chris Terman, Massachusetts Institute of Technology
A valuable set of analysis tools for design rule checking, electrical rules checking, and simulation is described.

32 ASAP—Advanced Symbolic Artwork Preparation
Kent Hardage, Hewlett-Packard Company
ASAP is a design system that supports a structured design methodology for MOS and bipolar circuits. Documented results indicate a significant saving in design time, along with increased transistor packing densities when compared with traditional techniques.

40 Distributed Processing in a High-Performance Smart Image Memory
James H. Clark and Mark R. Hannah, Stanford University
A novel multi-processor organization and custom LSI combine to generate a very high-performance graphics capability at low cost.

46 An Algorithm for MOS Logic Simulation
Randal E. Bryant, Massachusetts Institute of Technology
An efficient simulator especially suited for MOS circuits is described. Details of the implementation are included.

55 Design and Market Potential for Gate Arrays
Robert F. Hartmann, I.C. Cost Consultants
The technology for gate arrays seems well-established, but poor CAD and lack of leadership within the industry offer obstacles to growth.

60 Problems and Promise of Analog MOS/LSI
Edmund K. Cheng, Intel Corporation
Clever circuit design has been largely responsible for the advances made by analog MOS/LSI in the last five years. The future seems bright for further improvement, but device scaling holds some problems.

71 IC Photography for Every Application
Melgar Photographers can produce integrated circuit photographs for any purpose, whether for circuit analysis or living room decoration. A portfolio of their work is displayed in this article.
A Description of a Single-Chip Implementation of the RSA Cipher

Ronald L. Rivest, Massachusetts Institute of Technology

In 1976 Diffie and Hellman introduced the revolutionary concept of a public-key cryptosystem (Diffie and Hellman, 1976). Unlike classical cryptosystems, in a public-key cryptosystem the encryption and decryption keys are different. Moreover, given just one of the two keys, it is computationally prohibitive to calculate the other, although it is simple to create the matched pair of keys in the first place. This paradoxical property enables one to create flexible and powerful key distribution methods and to implement true “digital signatures” that are invaluable in the design of modern information protection and authentication systems.

To illustrate these capabilities, consider the following implementation of “digital signatures.” Every user of a public-key cryptosystem publishes a decryption key and keeps the matching encryption key secret. To create a signature $S$ for a message $M$, he merely encrypts $M$ with his secret encryption key; the resulting ciphertext is $S$. Anyone else can verify the validity of a message-signature pair $(M, S)$ by checking that the signature $S$ decodes to $M$ under the signer’s public encryption key. It is not possible to forge signatures or modify a signed message, given that knowledge of the signer’s decryption key does not imply knowledge of the signer’s encryption key.

To ensure confidentiality of communications, each user $A$ of a communication system can publish his encryption key $E_A$ while keeping the corresponding decryption key $D_A$ secret. Whenever someone wishes to send $A$ a private message, he can encrypt the message $M$ with $A$’s published key to obtain the ciphertext $C = E_A(M)$. Even though an eavesdropper knows what the encryption key is, the nature of a public-key cryptosystem and the fact that the eavesdropper doesn’t know the decryption key prevent him from reading the mail. When $A$ receives the mail, he can decipher it using $D_A$.

This first practical proposal for a public-key encryption algorithm was by Rivest, Shamir, and Adleman (Rivest, Shamir, and Adleman, 1978). This scheme, based on the difficulty of factoring large integers, has become known as the “RSA method” and has received extensive coverage in the popular and technical press (Gardner, 1977; Lempel, 1979; Diffie and Hellman, 1979; Blakely and Blakely, 1978-79, for example.) No way of “breaking” the code has been found to be quicker than factoring the large composite integer $n$ which is part of the key; yet factoring large composite integers remains one of the “computational impossibilities” of our time.

A potential disadvantage of the RSA method is that encryption and decryption are computationally demanding, requiring up to several hundred multiplication of several hundred bit numbers. A typical microprocessor-based implementation might achieve an encryption rate of ten bits per second, while a costly TTL implementation (e.g., 53K, 100 chips) might reach 6K bits second. Some interesting “hybrid” RSA/DES schemes have been developed which use RSA for key distribution only and DES for fast data encryption. This paper describes a cost-effective alternative: a special-purpose LSI chip to implement the RSA method. Shamir, Adleman, and I have designed such a “big-number ALU” chip which can support all of the usual big-number operations, including those needed to perform RSA encryption. This “RSA chip” has a 512-bit ALU and eight general-purpose 512-bit registers; it can perform RSA encryption at rates in excess of 1200 bits/second (even faster if keys shorter than the maximum length are used).

The RSA Method

The encryption method is only summarized here; the reader is referred to Rivesi, Shamir, and Adleman, 1978, for a full exposition.

An RSA encryption key consists of a pair of positive integers: $(e, n)$. A message $M$ to be encrypted must be an integer in the range $0$ to $n-1$. The ciphertext $C$ is obtained by encrypting $M$ as follows:

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

That is, $C$ is the remainder obtained when the $e$-th power of $M$ is divided by $n$.

Similarly, a decryption key is a pair of positive integers: $(d, n)$. Here, $n$ is the same as in the encryption key. The message $M$ can be obtained by decrypting the ciphertext $C$:

$$M = C^d \pmod{n}.$$  

Note that the encryption and decryption operations have a common form, which simplifies implementation.

The modulus $n$ is chosen to be the product of two large prime numbers, $p$ and $q$. (As it happens, large prime numbers are relatively common, and testing a large number for primality is not too difficult (Solovay and Strassen, 1977; Adleman, 1980)). A cryptanalyst could attempt to “break” the RSA method by factoring the modulus $n$. (Recall that in a public-key cryptosystem the encryption key $(e, n)$ may be public knowledge.) However, factoring large integers seems
A SHORT REPORT ON THE RSA CHIP

Ronald L. Rivest

MIT Laboratory for Computer Science
Cambridge, Mass. 02139

The nMOS "RSA chip" described in our article [1] was initially fabricated by Hewlett-Packard. Testing revealed that while the control portion of the chip worked correctly, the arithmetic section suffered from transient errors and was usually too unreliable to complete a full encryption. We tested a number of chips and found the same problem, enough to convince us that the cause was probably a design error and not a fabrication problem.
RSA Chips (Past/Present/Future)*
(Extended abstract)

Ronald L. Rivest
MIT Laboratory for Computer Science
Cambridge, Mass. 02139

Brief Abstract We review the issues involved in building a special-purpose chip for performing RSA encryption/decryption, and review a few of the current implementation efforts.

Eurocrypt 1984

* This research was supported by NSF grant MCS-80-0938.
There are two basically different exponentiation algorithms one may use: the left-to-right algorithm and the right-to-left algorithm. These algorithms examine the bits of the exponent in different orders. Suppose the exponent \( e \) has a binary representation of \( e_{k-1}e_{k-2}\ldots e_1e_0 \). Then the algorithms for computing a ciphertext \( C \) from a message \( M \) both begin by setting \( C \) to 1, and then proceed as follows:

- The **Left-to-Right Algorithm**: for \( i \) from \( k - 1 \) down to 0, this algorithm first sets \( C \) to \( C^2 \mod n \) and then, if \( e_i = 1 \), sets \( C \) to \( C \cdot M \mod n \).
- The **Right-to-Left Algorithm**: for \( i \) from 0 up to \( k - 1 \), this algorithm first sets \( C \) to \( C \cdot M \mod n \) if \( e_i = 1 \), and then (in any case) sets \( M \) to \( M^2 \mod n \).

If the left-to-right algorithm is used, then the number of modular multiplications required in the worst case can be reduced from \( 2 \cdot k \) to \( k + \left( \frac{k}{2} \right) \) by precomputing a table of \( M^1, \ldots, M^{2^{n-1}} \) (i.e. by modifying the left-to-right algorithm to consider the exponent \( e \) in radix \( 2^t \) instead of radix 2).

If the right-to-left algorithm is used, then by using twice as much hardware one can obtain a two-fold speed-up, since each squaring modular multiplication can be performed in parallel with the “accumulation” modular multiplication.

We note that the above two optimization techniques are incompatible, since they require different underlying exponentiation algorithms.

An elegant approach for speeding up the computation is to perform modular multiplication directly, rather than first performing an integer multiplication and then reducing the result modulo \( n \) as a separate step. This can yield a six-fold (approximately) speed-up, since the modular multiplication of two \( k \)-bit numbers can now be performed in approximately \( k \) clock cycles instead of approximately \( 6 \cdot k \). (see [Br82]).

**IV. Overview of Existing/Planned Chips**

In this section we review briefly six designs for RSA chips. These reviews are brief, and only intended to give the reader a feel for the kinds of chips possible with today’s technology. For more details the reader should consult the references. Also, there are other chips in the design stage for which no references exist; these chips are not listed here.

**IV.A. The “first” RSA chip**

This chip was designed by Rivest, Shamir, and Adleman, and is described in [Ri80].

It was a single-chip nMOS design; using 4-micron design rules, the chip occupied 42 \( \text{mm}^2 \). It contained a 512-bit ALU in bit-slice design with eight 512-bit registers for storage of intermediate results, carry-save adder logic, and up-down shifter logic. The 224-word microcode ROM contained control routines for encryption, decryption, finding large primes, gcd, etc. It used a 5V supply, and drew approximately 1 watt of power. It contained approximately 40,000 transistors. It communicated with a host microprocessor using an 8-bit I/O port. The encryption rate was designed to be slightly in excess of 1200 bits/second. Due to an as yet undiagnosed error in the memory cell design, this chip never worked reliably.

**IV.B. The NEC/Miyaguchi Design**

This chip design was described in [Mi82]; I do not know if it was ever fabricated.

The design was for a cascadable chip set, with each chip having a 2-bit slice. (So 333 chips would be needed for a 200 decimal digit modulus.) Each chip would contain a 2 by 8 multiplier;
Evolution of Intel Microprocessors: 1971 to 2007

<table>
<thead>
<tr>
<th>Family</th>
<th>Trade Name (Code Name for Future Chips)</th>
<th>Clock Frequency in MegaHertz**</th>
<th>Millions of Instructions per Second</th>
<th>Date of Introduction</th>
<th>Number of Transistors</th>
<th>Design Rule (Pixel Size)</th>
<th>Address Bus Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>Projected Roadmap</td>
<td>24,000.0 MHz</td>
<td>+125,000. MIPS</td>
<td>2007</td>
<td>1 billion</td>
<td>0.045 micron</td>
<td>64 bit</td>
</tr>
<tr>
<td></td>
<td>(Northwood)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8086</td>
<td>(Madison)</td>
<td>3,000.0 MHz</td>
<td>TBA</td>
<td>2003</td>
<td>TBA</td>
<td>0.13 micron</td>
<td>64 bit</td>
</tr>
<tr>
<td>8086</td>
<td>(Deerfield)**</td>
<td>TBA</td>
<td>TBA</td>
<td>2003</td>
<td>TBA</td>
<td>0.13 micron</td>
<td>64 bit</td>
</tr>
<tr>
<td>8086</td>
<td>(McKinley)</td>
<td>1,000.0 MHz</td>
<td>TBA</td>
<td>2002Q2</td>
<td>TBA</td>
<td>0.18 micron</td>
<td>64 bit</td>
</tr>
<tr>
<td>80786</td>
<td>Itanium (Merced)</td>
<td>800.0 MHz</td>
<td>+2,500.00 MIPS</td>
<td>May 29, 2001</td>
<td>30 / 300 M</td>
<td>0.18 micron</td>
<td>64 bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80686</td>
<td>Pentium 4</td>
<td>1,500.0 MHz</td>
<td>*1,500.00 MIPS</td>
<td>November 20, 2000</td>
<td>42 million</td>
<td>0.18 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80686</td>
<td>Pentium III</td>
<td>1,000.0 MHz</td>
<td>*1,000.00 MIPS</td>
<td>March 1, 2000</td>
<td>28.1 million</td>
<td>0.18 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80686</td>
<td>P III Xeon</td>
<td>733.0 MHz</td>
<td>*733.00 MIPS</td>
<td>October 25, 1999</td>
<td>28.1 million</td>
<td>0.18 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80686</td>
<td>Mobile P II</td>
<td>400.0 MHz</td>
<td>*400.00 MIPS</td>
<td>June 14, 1999</td>
<td>27.4 million</td>
<td>0.18 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80686</td>
<td>P III Xeon</td>
<td>550.0 MHz</td>
<td>*550.00 MIPS</td>
<td>March 17, 1999</td>
<td>9.5 million</td>
<td>0.25 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80686</td>
<td>Pentium II</td>
<td>500.0 MHz</td>
<td>*500.00 MIPS</td>
<td>February 26, 1999</td>
<td>9.5 million</td>
<td>0.25 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80686</td>
<td>P II Xeon</td>
<td>400.0 MHz</td>
<td>*400.00 MIPS</td>
<td>June 29, 1998</td>
<td>7.5 million</td>
<td>0.25 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80686</td>
<td>Pentium II</td>
<td>333.0 MHz</td>
<td>*333.00 MIPS</td>
<td>January 26, 1998</td>
<td>7.5 million</td>
<td>0.25 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80686</td>
<td>Pentium II</td>
<td>300.0 MHz</td>
<td>*300.00 MIPS</td>
<td>May 7, 1997</td>
<td>7.5 million</td>
<td>0.35 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80586</td>
<td>Pentium Pro</td>
<td>200.0 MHz</td>
<td>*200.00 MIPS</td>
<td>November 1, 1995</td>
<td>5.5 million</td>
<td>0.35 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>90586</td>
<td>Pentium</td>
<td>133.0 MHz</td>
<td>*133.00 MIPS</td>
<td>June 1995</td>
<td>3.3 million</td>
<td>0.35 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80586</td>
<td>Pentium</td>
<td>90.0 MHz</td>
<td>*90.00 MIPS</td>
<td>March 1994</td>
<td>3.2 million</td>
<td>0.60 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80586</td>
<td>Pentium</td>
<td>60.0 MHz</td>
<td>*60.00 MIPS</td>
<td>March 22, 1993</td>
<td>3.1 million</td>
<td>0.80 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80486</td>
<td>80486 DX2</td>
<td>50.0 MHz</td>
<td>*50.00 MIPS</td>
<td>March 3, 1992</td>
<td>1.2 million</td>
<td>0.80 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80486</td>
<td>486 DX</td>
<td>25.0 MHz</td>
<td>20.00 MIPS</td>
<td>April 10, 1989</td>
<td>1.2 million</td>
<td>1.00 micron</td>
<td>32 bit</td>
</tr>
<tr>
<td>80386</td>
<td>386 DX</td>
<td>16.0 MHz</td>
<td>5.00 MIPS</td>
<td>October 17, 1985</td>
<td>275,000</td>
<td>1.50 micron</td>
<td>16 bit</td>
</tr>
<tr>
<td>80286</td>
<td>80286</td>
<td>6.0 MHz</td>
<td>0.90 MIPS</td>
<td>February 1982</td>
<td>134,000</td>
<td>1.50 micron</td>
<td>16 bit</td>
</tr>
<tr>
<td>8086</td>
<td>8086</td>
<td>5.0 MHz</td>
<td>0.33 MIPS</td>
<td>June 8, 1978</td>
<td>29,000</td>
<td>3.00 micron</td>
<td>16 bit</td>
</tr>
<tr>
<td>8080</td>
<td>8080</td>
<td>2.0 MHz</td>
<td>0.64 MIPS</td>
<td>April 1974</td>
<td>6,000</td>
<td>6.00 micron</td>
<td>8 bit</td>
</tr>
<tr>
<td>8008</td>
<td>8008</td>
<td>.2 MHz</td>
<td>2.06 MIPS</td>
<td>April 1972</td>
<td>3,500</td>
<td>10.00 micron</td>
<td>8 bit</td>
</tr>
<tr>
<td>4004</td>
<td>4004</td>
<td>.1 MHz</td>
<td>0.06 MIPS</td>
<td>November 15, 1971</td>
<td>2,300</td>
<td>10.00 micron</td>
<td>4 bit</td>
</tr>
</tbody>
</table>

- Approximately one instruction per processor clock cycle // = starting with Itanium, the chips have multiple floating point processors per chip
- 1 KHz (KiloHertz) = 1 thousand cycles per second; 1 MegaHertz = 1 thousand KiloHertz; 100 KHz = 1 MHz; 1 GHz (GigaHertz) = 1 billion cycles per second; 1 GigaHertz = 1 thousand MegaHertz
- TBA To be announced, Pentium 4 was formerly code named Willamette. *** Deerfield is a low cost version of Madison.

Design Rule: because the wires and components, including transistors, on chips are drawn photographically, the pixel size of the imaging process determines the width of the wires and the size of the transistors. The size of the transistors determines how many will fit on a chip of a given size. (The optimal size of a chip depends on the chip manufacturing processes. In general, chip size increases slowly over time.) The smaller the transistors, the more will fit on the chip, determining the chip’s transistor budget. The size of the transistors also determines the transistor’s switching speed. Smaller transistors switch faster. Each micron is one one-millionth of a meter or about 40 millionths of an inch. Finally, the power required to switch smaller transistors is less, so smaller pixels in the design rules allow the batteries in laptop computers to last longer.

Number of Transistors: The number of transistors increases as the square of decrease in design rule size. Each reduction in design rule size is chosen to double the number of available transistors (the transistor budget). [For example: (.25/ .18 micron) x (.25 / .18) = 2.] The gradual increase in die size (the size of the chip) also increases the number of transistors. The Itanium chip has 30 million processor transistors and 300 million memory cache transistors.

Address Bus Bits: The address bus width in bits is based on the microprocessor chip family. (In the later chips of the 8086 family, some changes have been made to make more memory addressable under special circumstances, by using 36 bits to address 16 times as much memory as is possible with 32 address bits, but the generalized addressing structure is still 32 bits.) Each time a bit is added to the address bus width, the amount of memory (RAM; Random Access Memory) that can be addressed is doubled. 4 bit addresses allow the addressing of 16 bytes of memory (and extra work is necessary to address 256 bytes of memory). 8 bits allow the addressing of 256 bytes of memory (and extra work is necessary to address 65,536 bytes of memory). 16 bits can address 65,536 bytes of memory. 32 bits can address 4,294,967,296 bytes of memory (about 4 billion bytes). As memory prices drop, it becomes necessary to address over 4 billion bytes of memory. The 80786 family, (the Itanium) debuted May 29, 2001. It has a 64 bit address bus and will be able to address over 16 billion billion (16 quintillion) bytes of memory.

See also: http://www.intel.com/intel/museum/ Intel’s history of the microprocessor.

*MHz (MegaHertz) ( Millions of processor cycles per second) The number of times the processor goes through one cycle. The start of a processor cycle is determined by a pulse (tick) from the processor’s clock.

**GHz (GigaHertz) ( Billions of processors cycles per second), 1 thousand MHz = 1 GHz

MIPS: (Millions of Instructions per Second) with the introduction of the 80486 DX2, parallel instruction execution increased the number of instructions executed per processor cycle to approximately one instruction per cycle. Parallel instruction execution requires many more transistors, so the increase in the number of transistors has increased the number of instructions that can be executed per second faster than the clock cycle speed has increased. A larger transistor budget allows the addition of specialized instructions, which increase the microprocessor’s speed in processing specialized information such as graphics by increasing the amount of information processed per instruction.

*Multiprocessors on the chip produce more than one instruction per clock cycle.

GIPS: (GigaInstructions per Second) Billions of instructions per second. 1 thousand MIPS = 1 GIP

Steve Gilheany
In this talk, we give a brief summary of advanced algorithms for creating efficient hardware realizations of public-key cryptographic algorithms: Diffie-Hellman, RSA, and 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
- Scalable arithmetic, 1999
- Unified arithmetic, 2000
- Spectral arithmetic, 2007
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 \( M^e \pmod{n} \) 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

\[
\begin{align*}
R_A & := g^a \pmod{p} \\
R_B & := g^b \pmod{p} \\
R'_B & := R'_A = g^{ab} \pmod{p} \\
R'_A & := R'_{B} = g^{ba} \pmod{p}
\end{align*}
\]

between two parties, Alice & Bob

• These computations are also modular exponentiations, requiring modular addition, subtraction, and multiplication operations
The signature computation on $M$ and $k$ is the pair $(r, s)$

\[
\begin{align*}
    r & := (g^k \mod p) \mod q \\
    s & := (M + xr)k^{-1} \mod q \\
\end{align*}
\]

The signature verification

\[
\begin{align*}
    w & := s^{-1} \mod q \\
    u_1 & := Mw \mod q \\
    u_2 & := rw \mod q \\
    v & := (g^{u_1}y^{u_2} \mod p) \mod q \\
    \text{Check if} \quad r & = v
\end{align*}
\]
Elliptic 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 \( P = a \cdot b \mod n \)

- Multiply and reduce:
  Multiply: \( P' = a \cdot b \) (2\(k\)-bit number)
  Reduce: \( P = P' \mod n \) (\(k\)-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

\[ P' = a \cdot b = a \cdot \sum_{i=0}^{k-1} b_i 2^i = \sum_{i=0}^{k-1} (a \cdot b_i) 2^i \]

\[ = 2(\cdots 2(2(0 + a \cdot b_{k-1}) + a \cdot b_{k-2}) + \cdots) + a \cdot b_0 \]

1. \( P := 0 \)
2. \( \textbf{for } i = k - 1 \textbf{ downto } 0 \)
2a. \( P := 2P + a \cdot b_i \)
2b. \( P := P \mod n \)
3. \( \textbf{return } P \)

- 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: addition instead of division — 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

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

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. $u := u/2$

- Step 2b is only an addition! And, it 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 architectures, 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 $m = t \cdot n' \mod r$ and $u = (t + m \cdot n)/r$. Squaring can be optimized.

  SOS requires $2s + 2$ words of space
Advanced Montgomery Multiplication

• 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.

• 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.
Advanced Montgomery Multiplication

• 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.

• All methods require \( 2s^2 + s \) multiplications.

• Add, Read/Write and Space requirements are below

<table>
<thead>
<tr>
<th></th>
<th>Add</th>
<th>Read/Write</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOS</td>
<td>( 4s^2+4s+2 )</td>
<td>( 8s^2+13s+5 )</td>
<td>( 2s+2 )</td>
</tr>
<tr>
<td>FIPS</td>
<td>( 6s^2+2s+2 )</td>
<td>( 14s^2+16s+3 )</td>
<td>( s+3 )</td>
</tr>
<tr>
<td>FIOS</td>
<td>( 5s^2+3s+2 )</td>
<td>( 10s^2+9s+3 )</td>
<td>( s+3 )</td>
</tr>
<tr>
<td>CIHS</td>
<td>( 4s^2+4s+2 )</td>
<td>( 9.5s^2+11.5s+3 )</td>
<td>( s+3 )</td>
</tr>
<tr>
<td>CIOS</td>
<td>( 4s^2+4s+2 )</td>
<td>( 8s^2+12s+3 )</td>
<td>( s+3 )</td>
</tr>
</tbody>
</table>
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

\[ u(x) = \text{MonPro}(a(x), b(x)) = a(x) \cdot b(x) \cdot x^{-k} \mod p(x), \]

we use the steps below

1. \( u(x) := 0 \)
2. \( \text{for } i = 0 \text{ to } k - 1 \)
   2a. \( u(x) := u(x) + a_i \cdot b(x) \mod 2 \)
   2b. \( \text{if } u_0 \text{ is 1 then } u(x) := u(x) + p(x) \mod 2 \)
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 $GF(2^k)$.
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

Dept of EE @ UCLA - Jan 14, 2009
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

Let $\omega$ 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 invertible set map sending $x(t)$ to $X(t)$ given by

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

with the inverse

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

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

\[ x(t) \overset{\text{DFT}}{\longleftrightarrow} X(t) \]

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 \( q \)
In order to have simpler arithmetic, we need to select the ring more carefully.

For example, $q$ can be chosen as a Mersenne number $q = 2^v - 1$.

Or, $q$ can be chosen as a Fermat number $q = 2^v + 1$.

Arithmetic operations in Mersenne and Fermat rings are significantly more efficient (in time and area) than those in a ring chosen arbitrarily.

Furthermore, we suggest that the principal root of unity $\omega$ be selected as a power of 2 (or -2) in order to simplify the multiplications with the 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

• We need to create a dictionary of operations between the time domain and the spectral domain

• The purpose is to have the correct set of operations in the spectral domain which accomplish particular computational goals in the time domain
**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 \( x(t) \odot 1 \)

\[
x_1 + x_2 t + \ldots + x_{d-2} t^{d-1} + x_0 t^{d-1}
\]

\[
\uparrow_{\text{DFT}}
\]

\[
X(t) \circ \Gamma(t)
\]

where \( \odot \) stands for component-wise multiplication and

\[
\Gamma(t) = 1 + \omega^{-1} t + \ldots + \omega^{-(d-1)} t^{d-1}
\]
• **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.

\[
\sum_{i=0}^{d-1} x_i \omega^{-i} = d^{-1} x_0 \quad \text{and} \quad \sum_{i=0}^{d-1} x_i \omega^i = X_0
\]
**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:

$$(x(t) - x_0)/t = x_1 + \ldots + x_{d-1}t^{d-2}$$

$$\uparrow_{\text{DFT}}$$

$$(X(t) - x_0(t)) \odot \Gamma(t)$$

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 $\Gamma(t)$
We would like to compute $859^2 \cdot 4^{-9} \pmod{1337}$.

Signal $x(t)$ representing $859 = x(4)$ in base 4.
Convolving $x(t)$ with itself, we find $x^2(t) = 859^2 = 737881$. 
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.

Dept of EE @ UCLA - Jan 14, 2009
We then shift the digits.
After 9 iterations, we find the result: $914 \equiv 859^2 \cdot 4^{-9} \pmod{1337}$. 
Quest for Efficiency

- Platforms are diverse: Huge SSL and IPSec boxes versus tiny Bluetooth earphones, cellphones and PDAs. Low-to-moderate speed electronic communication versus Gb/sec photonic networks.

- Old challenges remain: Fast but low-area and low-energy hardware design for cryptography.

- New challenges:
  - Cryptography in “extreme” environments, for example, access to physical devices by adversaries (side-channel attacks).
  - Content protection for transient and stationary data.
  - Countermeasures to circumvent attacks by (local and remote) adversaries to obtain hardware-hidden secrets.