RSA Hardware Implementation

Çetin Kaya Koç
Koc@ece.orst.edu

RSA Laboratories
RSA Data Security, Inc.
100 Marine Parkway, Suite 500
Redwood City, CA 94065-1031

Version 1.0 – August 1995

Contents

1 RSA Algorithm

The RSA algorithm, invented by Rivest, Shamir, and Adleman [25], is one of the simplest public-key cryptosystems. The parameters are n, p and q, e, and d. The modulus n is the product of the distinct large random primes: $n = pq$. The public exponent e is a number in the range $1 < e < \phi(n)$ such that

$$gcd(e, \phi(n)) = 1 \ ,$$

where $\phi(n)$ is Euler's totient function of n, given by

$$\phi(n) = (p-1)(q-1) \ .$$

The private exponent d is obtained by inverting e modulo $\phi(n)$, i.e.,

$$d = e^{-1} \bmod \phi(n) \ ,$$

using the extended Euclidean algorithm [11, 21]. Usually one selects a small public exponent, e.g., $e = 2^{16} + 1$. The encryption operation is performed by computing

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

where M is the plaintext such that $0 \leq M < n$. The number C is the ciphertext from which the plaintext M can be computed using

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

As an example, let $p = 11$ and $q = 13$. We compute $n = pq = 11 \cdot 13 = 143$ and

$$\phi(n) = (p-1)(q-1) = 10 \cdot 12 = 120 \ .$$

The public exponent e is selected such that $1 < e < \phi(n)$ and

$$\gcd(e, \phi(n)) = \gcd(e, 120) = 1 \ .$$

For example, $e = 17$ would satisfy this constraint. The private exponent d is obtained by inverting e modulo $\phi(n)$ as

$$\begin{aligned} d &= 17^{-1} \pmod{120} \\ &= 113 \ , \end{aligned}$$

which can be computed using the extended Euclidean algorithm. The user publishes the public exponent and the modulus: $(e, n) = (13, 143)$, and keeps the following private: $d = 113$, $p = 11$, $q = 13$. Let $M = 50$ be the plaintext. It is encrypted by computing $C = M^e \pmod{n}$ as

$$\begin{aligned} C &= 50^{17} \pmod{143} \\ &= 85 \ . \end{aligned}$$

The ciphertext $C = 85$ is decrypted by computing $M = C^d \pmod{n}$ as

$$\begin{aligned} M &= 85^{113} \pmod{143} \\ &= 50 \ . \end{aligned}$$

The RSA algorithm can be used to send encrypted messages and to produce digital signatures for electronic documents. It provides a procedure for signing a digital document, and verifying whether the signature is indeed authentic. The signing of a digital document is somewhat different from signing a paper document, where the same signature is being produced for all paper documents. A digital signature cannot be a constant; it is a function of the digital document for which it was produced. After the signature (which is just another piece of digital data) of a digital document is obtained, it is attached to the document for anyone wishing the verify the authenticity of the document and the signature. We refer the reader to the technical reports Answers to Frequently Asked Questions About Today's Cryptography and Public Key Cryptography Standards published by the RSA Laboratories [26, 27] for answers to certain questions on these issues.

2 Computation of Modular Exponentiation

Once the modulus and the private and public exponents are determined, the senders and recipients perform a single operation for signing, verification, encryption, and decryption. The operation required is the computation of $M^e \pmod{n}$, i.e., the modular exponentiation. The modular exponentiation operation is a common operation for scrambling; it is used in several cryptosystems. For example, the Diffie-Hellman key exchange scheme requires modular exponentiation [6]. Furthermore, the ElGamal signature scheme [7] and the Digital Signature Standard (DSS) of the National Institute for Standards and Technology [22] also require the computation of modular exponentiation. However, we note that the exponentiation process in a cryptosystem based on the discrete logarithm problem is slightly different: The base (M) and the modulus (n) are known in advance. This allows some precomputation since powers of the base can be precomputed and saved [5]. In the exponentiation process for the RSA algorithm, we know the exponent (e) and the modulus (n) in advance but not the base (M); thus, such optimizations are not likely to be applicable.

In the following sections we will review techniques for implementation of the modular exponentiation operation in hardware. We will study techniques for exponentiation, modular multiplication, modular addition, and addition operations. We intend to cover mathematical and algorithmic aspects of the modular exponentiation operation, providing the necessary knowledge to the hardware designer who is interested implementing the RSA algorithm using a particular technology. We draw our material from computer arithmetic books [32, 10, 34, 17], collection of articles [31, 30], and journal and conference articles on hardware structures for performing the modular multiplication and exponentiations [24, 16, 28, 9, 4, 13, 14, 15, 33]. For implementing the RSA algorithm in software, we refer the reader to the companion report High-Speed RSA Implementation published by the RSA Laboratories [12].

3 RSA Operations and Parameters

The RSA algorithm requires computation of the modular exponentiation which is broken into a series of modular multiplications by the application of exponentiation heuristics. Before getting into the details of these operations, we make the following definitions:

- The public modulus n is a k-bit positive integer, ranging from 512 to 2048 bits.
- The secret primes p and q are approximately $k/2$ bits.
- The public exponent e is an h-bit positive integer. The size of e is small, usually not more than 32 bits. The smallest possible value of e is 3.
- The secret exponent d is a large number; it may be as large as $\phi(n) - 1$. We will assume that d is a k-bit positive integer.

After these definitions, we will study algorithms for modular exponentiation, exponentiation, modular multiplication, multiplication, modular addition, addition, and subtraction operations on large integers.

4 Modular Exponentiation Operation

The modular exponentiation operation is simply an exponentiation operation where multiplication and squaring operations are modular operations. The exponentiation heuristics developed for computing M^e are applicable for computing $M^e \pmod{n}$. In the companion report [12], we review several techniques for the exponentiation operation. In the domain of hardware implementation, we will mention a couple of details, and refer the reader to the companion report [12] for more information on exponentiation heuristics.

The binary method for computing $M^e \pmod{n}$ given the integers M, e, and n has two variations depending on the direction by which the bits of e are scanned: Left-to-Right (LR) and Right-to-Left (RL). The LR binary method is more widely known:

LR Binary Method
Input: M, e, n
Output: $C := M^e \bmod n$
1. **if** $e_{h-1} = 1$ **then** $C := M$ **else** $C := 1$
2. **for** $i = h - 2$ **downto** 0
2a. $\quad C := C \cdot C \pmod{n}$
2b. $\quad$ **if** $e_i = 1$ **then** $C := C \cdot M \pmod{n}$
3. **return** C

The bits of e are scanned from the most significant to the least significant, and a modular squaring is performed for each bit. A modular multiplication operation is performed only if the bit is 1. An example of LR binary method is illustrated below for $h = 6$ and $e = 55 = (110111)$. Since $e_5 = 1$, the LR algorithm starts with $C := M$, and proceeds as

i	e_i	Step 2a (C)	Step 2b (C)
4	1	$(M)^2 = M^2$	$M^2 \cdot M = M^3$
3	0	$(M^3)^2 = M^6$	M^6
2	1	$(M^6)^2 = M^{12}$	$M^{12} \cdot M = M^{13}$
1	1	$(M^{13})^2 = M^{26}$	$M^{26} \cdot M = M^{27}$
0	1	$(M^{27})^2 = M^{54}$	$M^{54} \cdot M = M^{55}$

The RL binary algorithm, on the other hand, scans the bits of e from the least significant to the most significant, and uses an auxiliary variable P to keep the powers M.

RL Binary Method
Input: M, e, n
Output: $C := M^e \bmod n$
1. $C := 1$; $P := M$
2. **for** $i = 0$ **to** $h - 2$
2a. **if** $e_i = 1$ **then** $C := C \cdot P \pmod{n}$
2b. $P := P \cdot P \pmod{n}$
3. **if** $e_{h-1} = 1$ **then** $C := C \cdot P \pmod{n}$
4. **return** C

The RL algorithm starts with $C =: 1$ and $P := M$, proceeds to compute M^{55} as follows:

i	e_i	Step 2a (C)	Step 2b (P)
0	1	$1 \cdot M = M$	$(M)^2 = M^2$
1	1	$M \cdot M^2 = M^3$	$(M^2)^2 = M^4$
2	1	$M^3 \cdot M^4 = M^7$	$(M^4)^2 = M^8$
3	0	M^7	$(M^8)^2 = M^{16}$
4	1	$M^7 \cdot M^{16} = M^{23}$	$(M^{16})^2 = M^{32}$
Step 3: $e_5 = 1$, thus $C := M^{23} \cdot M^{32} = M^{55}$			

We compare the LR and RL algorithm in terms of time and space requirements below:

- Both methods require $h - 1$ squarings and an average of $\frac{1}{2}(h-1)$ multiplications.
- The LR binary method requires two registers: M and C.
- The RL binary method requires three registers: M, C, and P. However, we note that P can be used in place of M, if the value of M is not needed thereafter.
- The multiplication (Step 2a) and squaring (Step 2b) operations in the RL binary method are independent of one another, and thus these steps can be parallelized. Provided that we have two multipliers (one multiplier and one squarer) available, the running time of the RL binary method is bounded by the total time required for computing $h - 1$ squaring operations on k-bit integers.

The advanced exponentiation algorithms are often based on word-level scanning of the digits of the exponent e. As mentioned, the companion technical report [12] contains several advanced algorithms for computing the modular exponentiation, which are slightly faster than the binary method. The word-level algorithms, i.e., the m-ary methods, require some space to keep precomputed powers of M in order to reduce the running time. These algorithms may not be very suitable for hardware implementation since the space on-chip is already limited due to the large size of operands involved (e.g., 1024 bits). Thus, we will not study these techniques in this report.

The remainder of this report reviews algorithms for computing the basic modular arithmetic operations, namely, the addition, subtraction, and multiplication. We will assume that the underlying exponentiation heuristic is either the binary method, or any of the advanced m-ary algorithm with the necessary register space already made available. This assumption allows us to concentrate on developing time and area efficient algorithms for the basic modular arithmetic operations, which is the current challenge because of the operand size.

The literature is replete with residue arithmetic techniques applied to signal processing, see for example, the collection of papers in [30]. However, in such applications, the size of operands are very small, usually around 5–10 bits, allowing table lookup approaches. Besides the moduli are fixed and known in advance, which is definitely not the case for our application. Thus, entirely new set of approaches are needed to design time and area efficient hardware structures for performing modular arithmetic operations to be used in cryptographic applications.

5 Addition Operation

In this section, we study algorithms for computing the sum of two k-bit integers A and B. Let A_i and B_i for $i = 1, 2, \ldots, k-1$ represent the bits of the integers A and B, respectively. We would like to compute the sum bits S_i for $i = 1, 2, \ldots, k-1$ and the final carry-out C_k as follows:

$$\begin{array}{cccccc} & A_{k-1} & A_{k-2} & \cdots & A_1 & A_0 \\ + & B_{k-1} & B_{k-2} & \cdots & B_1 & B_0 \\ \hline C_k & S_{k-1} & S_{k-2} & \cdots & S_1 & S_0 \end{array}$$

We will study the following algorithms: the carry propagate adder (CPA), the carry completion sensing adder (CCSA), the carry look-ahead adder (CLA), the carry save adder (CSA), and the carry delayed adder (CDA) for computing the sum and the final carry-out.

5.1 Full-Adder and Half-Adder Cells

The building blocks of these adders are the full-adder (FA) and half-adder (HA) cells. Thus, we briefly introduce them here. A full-adder is a combinational circuit with 3 input and 2 outputs. The inputs A_i, B_i, C_i and the outputs S_i and C_{i+1} are boolean variables. It is assumed that A_i and B_i are the ith bits of the integers A and B, respectively, and C_i is the carry bit received by the ith position. The FA cell computes the sum bit S_i and the carry-out bit C_{i+1} which is to be received by the next cell. The truth table of the FA cell is as follows:

A_i	B_i	C_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

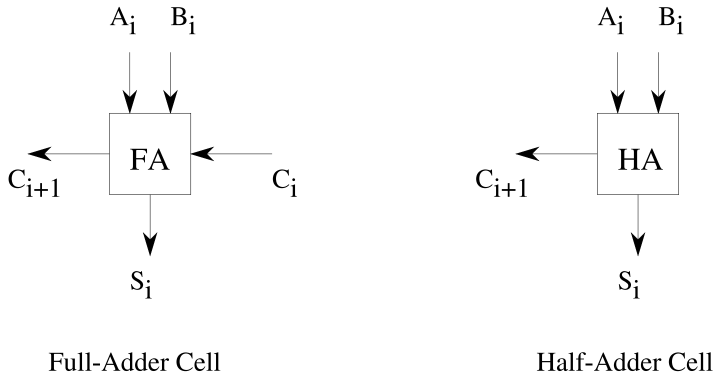
The boolean functions of the output values are as

$$\begin{aligned} C_{i+1} &= A_iB_i + A_iC_i + B_iC_i \ , \\ S_i &= A_i \oplus B_i \oplus C_i \ . \end{aligned}$$

Similarly, an half-adder is a combinational circuit with 2 inputs and 2 outputs. The inputs A_i, B_i and the outputs S_i and C_{i+1} are boolean variables. It is assumed that A_i and B_i are the ith bits of the integers A and B, respectively. The HA cell computes the sum bit S_i and the carry-out bit C_{i+1}. Thus, an half-adder is easily obtained by setting the third input bit C_i to zero. The truth table of the HA cell is as follows:

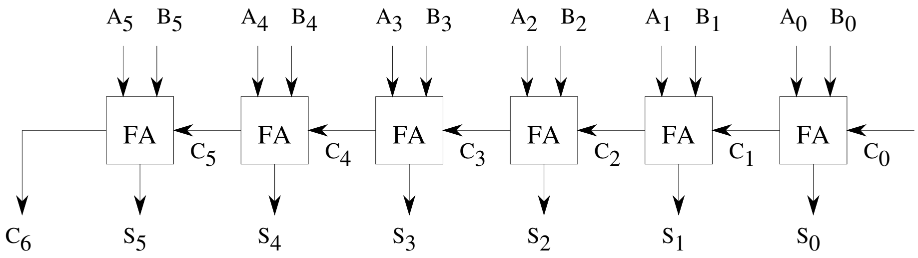
A_i	B_i	C_{i+1}	S_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The boolean functions of the output values are as $C_{i+1} = A_iB_i$ and $S_i = A_i \oplus B_i$, which can be obtained by setting the carry bit input C_i of the FA cell to zero. The following figure illustrates the FA and HA cells.



5.2 Carry Propagate Adder

The carry propagate adder is a linearly connected array of full-adder (FA) cells. The topology of the CPA is illustrated below for $k = 8$.



The total delay of the carry propagate adder is k times the delay of a single full-adder cell. This is because the ith cell needs to receive the correct value of the carry-in bit C_i in order to compute its correct outputs. Tracing back to the 0th cell, we conclude that a total of k full-adder delays is needed to compute the sum vector S and the final carry-out C_k. Furthermore, the total area of the k-bit CPA is equal to k times a single full-adder cell area. The CPA scales up very easily, by adding additional cells starting from the most significant.

The subtraction operation can be performed on a carry propagate adder by using 2's complement arithmetic. Assuming we have a k-bit CPA available, we encode the positive numbers in the range $[0, 2^{k-1}-1]$ as k-bit binary vectors with the most significant bit being 0. A negative number is then represented with its most significant bit as 1. This is accomplished as follows: Let $x \in [0, 2^{k-1}]$, then $-x$ is represented by computing $2^k - x$. For example, for $k = 3$, the positive numbers are $0, 1, 2, 3$ encoded as $000, 001, 010, 011$, respectively. The negative 1 is computed as $2^3 - 1 = 8 - 1 = 7 = 111$. Similarly, -2, -3, and -4 are encoded as 110, 101, and 100, respectively. This encoding system has two advantages which are relevant in performing modular arithmetic operations:

- The sign detection is easy: the most significant bit gives the sign.
- The subtraction is easy: In order to compute $x - y$, we first represent $-y$ using 2's complement encoding, and then add x to $-y$.

The CPA has several advantages but one clear disadvantage: the computation time is too long for our application, in which the operand size is in the order of several hundreds, up to 2048 bits. Thus, we need to explore other techniques with the hope of building circuits which require less time without significantly increasing the area.

5.3 Carry Completion Sensing Adder

The carry completion sensing adder is an asynchronous circuit with area requirement proportional to k. It is based on the observation that the average time required for the carry propagation process to complete is much less than the worst case which is k full-adder delays. For example, the addition of 15213 by 19989 produces the longest carry length as 5, as shown below:

A = 0 0 1 1 1 0 1 1 0 1 1 0 1 1 0 1

B = 0 1 0 0 1 1 1 0 0 0 0 1 0 1 0 1

4 1 5 1

A statistical analysis shows that the average longest carry sequence is approximately 4.6 for a 40-bit adder [8]. In general, the average longest carry produced by the addition of two k-bit integers is upper bounded by $\log_2 k$. Thus, we can design a circuit which detects the completion of all carry propagation processes, and completes in $\log_2 k$ time in the average.

A	0	1	1	**1**	0	**1**	1	**0**	1	1	0	1	**1**	**0**	**1**	
B	1	0	0	**1**	1	**1**	0	**0**	0	0	1	0	**1**	**0**	**1**	
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	t = 0
N	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
C	0	0	0	**1**	0	**1**	0	**0**	0	0	0	0	**1**	**0**	**1**	t = 1
N	0	0	0	**0**	0	**0**	0	**1**	0	0	0	0	**0**	**1**	**0**	
C	0	0	**1**	1	**1**	1	**0**	0	0	0	0	**1**	1	0	1	t = 2
N	0	0	**0**	0	**0**	0	**1**	1	0	0	0	**0**	0	1	0	
C	0	**1**	1	1	1	1	0	0	0	0	**1**	1	1	0	1	t = 3
N	0	**0**	0	0	0	0	1	1	0	0	**0**	0	0	1	0	
C	**1**	1	1	1	1	1	0	0	0	**1**	1	1	1	0	1	t = 4
N	**0**	0	0	0	0	0	1	1	0	**0**	0	0	0	1	0	
C	1	1	1	1	1	1	0	0	**1**	1	1	1	1	0	1	t = 5
N	0	0	0	0	0	0	1	1	**0**	0	0	0	0	1	0	

In order to accomplish this task, we introduce a new variable N in addition to the carry variable C. The value of C and N for ith position is computed using the values of A and B for the ith position, and the previous C and N values, as follows:

$$
\begin{aligned}
(A_i, B_i) = (0,0) &\implies (C_i, N_i) = (0,1) \\
(A_i, B_i) = (1,1) &\implies (C_i, N_i) = (1,0) \\
(A_i, B_i) = (0,1) &\implies (C_i, N_i) = (C_{i-1}, N_{i-1}) \\
(A_i, B_i) = (1,0) &\implies (C_i, N_i) = (C_{i-1}, N_{i-1})
\end{aligned}
$$

Initially, the C and N vectors are set to zero. The cells which produce C and N values start working as soon as the values of A and B are applied to them in parallel. The output of a cell (C_i, N_i) settles when its inputs (C_{i-1}, N_{i-1}) are settled. When all carry propagation processes are complete, we have either $(C_i, N_i) = (0, 1)$ or $(C_i, N_i) = (1, 0)$ for all $i = 1, 2, \ldots, k$. Thus, the end of carry completion is detected when all $X_i = C_i + N_i = 1$ for all $i = 1, 2, \ldots, k$, which can be accomplished by using a k-input AND gate.

5.4 Carry Look-Ahead Adder

The carry look-ahead adder is based on computing the carry bits C_i prior to the summation. The carry look-ahead logic makes use of the relationship between the carry bits C_i and the input bits A_i and B_i. We define two variables G_i and P_i, named as the generate and the propagate functions, as follows:

$$\begin{aligned} G_i &= A_i B_i \ , \\ P_i &= A_i + B_i \ . \end{aligned}$$

Then, we expand C_1 in terms of G_0 and P_0, and the input carry C_0 as

$$C_1 = A_0 B_0 + C_0(A_0 + B_0) = G_0 + C_0 P_0 \ .$$

Similarly, C_2 is expanded in terms G_1, P_1, and C_1 as

$$C_1 = G_1 + C_1 P_1 \ .$$

When we substitute C_1 in the above equation with the value of C_1 in the preceding equation, we obtain C_1 in terms G_0, G_1, P_0, P_1, and C_0 as

$$C_1 = G_1 + C_1 P_1 = G_1 + (G_0 + C_0 P_0) P_1 = G_1 + G_0 P_1 + C_0 P_0 P_1 \ .$$

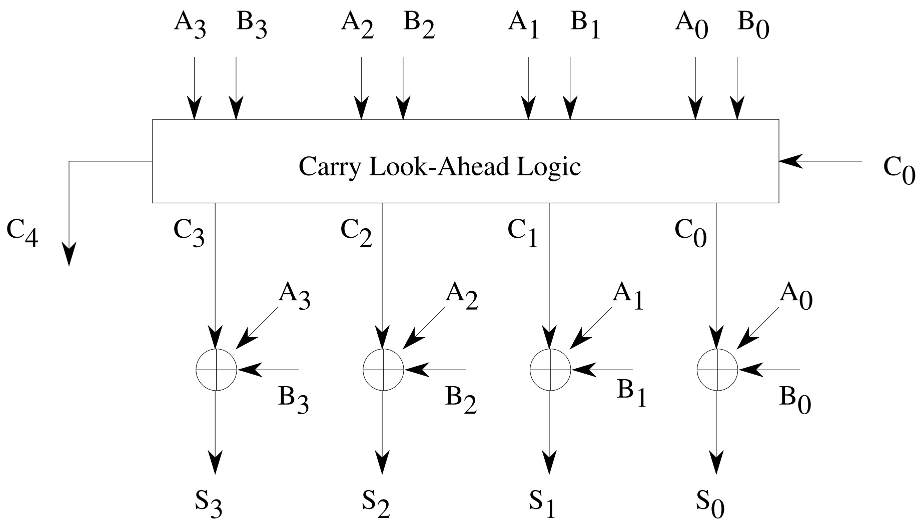
Proceeding in this fashion, we can obtain C_i as function of C_0 and $G_0, G_1, \ldots, G_i$ and $P_0, P_1, \ldots, P_i$. The carry functions up to C_4 are given below:

$$\begin{aligned} C_1 &= G_0 + C_0 P_0 \ , \\ C_2 &= G_1 + G_0 P_1 + C_0 P_0 P_1 \ , \\ C_3 &= G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2 \ , \\ C_4 &= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3 \ . \end{aligned}$$

The carry look-ahead logic uses these functions in order to compute all C_is in advance, and then feeds these values to an array of EXOR gates to compute the sum vector S. The $i$the element of the sum vector is computed using

$$S_i = A_i \oplus B_i \oplus C_i \ .$$

The carry look-ahead adder for $k = 3$ is illustrated below.



The CLA does not scale up very easily. In order to deal with large operands, we have basically two approaches:

- The block carry look-ahead adder: First we build small (4-bit or 8-bit) carry look-ahead logic cells with section generate and propagate functions, and then stack these to build larger carry look-ahead adders [10, 34, 17].

- The complete carry look-ahead adder: We build a complete carry look-ahead logic for the given operand size. In order to accomplish this task, the carry look-ahead functions are formulated in a way to allow the use of the parallel prefix circuits [2, 18, 19].

The total delay of the carry look-ahead adder is $O(\log k)$ which can be significantly less than the carry propagate adder. There is a penalty paid for this gain: The area increases. The block carry look-ahead adders require $O(k \log k)$ area, while the complete carry look-ahead adders require $O(k)$ area by making use of efficient parallel prefix circuits [19, 20]. It seems that a carry look-ahead adder larger than 256 bits is not cost effective, considering the fact there are better alternatives, e.g., the carry save adders. Even by employing block carry look-ahead approaches, a carry look-ahead adder with 1024 bits seems not feasible or cost effective.

5.5 Carry Save Adder

The carry save adder seems to be the most useful adder for our application. It is simply a parallel ensemble of k full-adders without any horizontal connection. Its main function is to add three k-bit integers A, B, and C to produce two integers C' and S such that

$$C' + S = A + B + C \ .$$

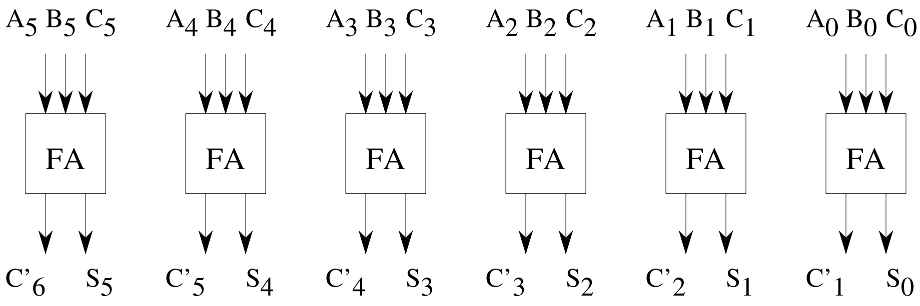
As an example, let $A = 40$, $B = 25$, and $C = 20$, we compute S and C' as shown below:

$$
\begin{array}{rcccccccc}
A = 40 & = & & 1 & 0 & 1 & 0 & 0 & 0 \\
B = 25 & = & & 0 & 1 & 1 & 0 & 0 & 1 \\
C = 20 & = & & 0 & 1 & 0 & 1 & 0 & 0 \\
\hline
S = 37 & = & & 1 & 0 & 0 & 1 & 0 & 1 \\
C' = 48 & = & 0 & 1 & 1 & 0 & 0 & 0 & \\
\hline
\end{array}
$$

The ith bit of the sum S_i and the $(i + 1)$st bit of the carry C'_{i+1} is calculated using the equations

$$
\begin{aligned}
S_i &= A_i \oplus B_i \oplus C_i \ . \\
C'_{i+1} &= A_i B_i + A_i C_i + B_i C_i \ ,
\end{aligned}
$$

in other words, a carry save adder cell is just a full-adder cell. A carry save adder, sometimes named a one-level CSA, is illustrated below for $k = 6$.



Since the input vectors A, B, and C are applied in parallel, the total delay of a carry save adder is equal to the total delay of a single FA cell. Thus, the addition of three integers to compute two integers requires a single FA delay. Furthermore, the CSA requires only k times the areas of FA cell, and scales up very easily by adding more parallel cells. The subtraction operation can also be performed by using 2's complement encoding. There are basically two disadvantages of the carry save adders:

- It does not really solve our problem of adding two integers and producing a single output. Instead, it adds three integers and produces two such that sum of these two is equal to the sum of three inputs. This method may not be suitable for application which only needs the regular addition.

- The sign detection is hard: When a number is represented as a carry-save pair (C, S) such that its actual value is $C + S$, we may not know the exact sign of total sum $C + S$. Unless the addition is performed in full length, the correct sign may never be determined.

We will explore this sign detection problem in an upcoming section in more detail. For now, it suffices to briefly mention the sign detection problem, and introduce a method of sign detection. This method is based on adding a few of the most significant bits of C and S in order to calculate (estimate) the sign. As an example, let $A = -18$, $B = 19$, $C = 6$. After the carry save addition process, we produce $S = -5$ and $C' = 12$, as shown below. Since the total sum $C' + S = 12 - 5 = 7$, its correct sign is 0. However, when we add the first most significant bits, we estimate the sign incorrectly.

A	=	-18	=		1	0	1	1	1	0	
B	=	19	=		0	1	0	0	1	1	
C	=	6	=		0	0	0	1	1	0	
S	=	-5	=		1	1	1	0	1	1	
C'	=	12	=	0	0	0	1	1	0		
					1						(1 MSB)
					1	1					(2 MSB)
					0	0	0				(3 MSB)
					0	0	0	1			(4 MSB)
					0	0	0	1	1		(5 MSB)
					0	0	0	1	1	1	(6 MSB)

The correct sign is computed only after adding the first three most significant bits. In the worst case, up to a full length addition may be required to calculate the correct sign.

5.6 Carry Delayed Adder

The carry delayed adder is a two-level carry save adder. As we will see in Section 7.3, a certain property of the carry delayed adder can be used to reduce the multiplication complexity. The carry delayed adder produced a pair of integers (D, T), called a carry delayed number, using the following set of equations:

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_i \ , \\ C_{i+1} &= A_i B_i + A_i C_i + B_i C_i \ , \\ T_i &= S_i \oplus C_i \ , \\ D_{i+1} &= S_i C_i \ , \end{aligned}$$

where $D_0 = 0$. Notice that C_{i+1} and S_i are the outputs of a full-adder cell with inputs A_i, B_i, and C_i, while the values D_{i+1} and T_i are the outputs of an half-adder cell.

An important property of the carry delayed adder is that $D_{i+1}T_i = 0$ for all $i = 0, 1, \ldots, k-1$. This is easily verified as

$$D_{i+1}T_i \ = \ S_iC_i(S_i \oplus C_i) \ = \ S_iC_i(\bar{S}_iC_i + S_i\bar{C}_i) \ = \ 0 \ .$$

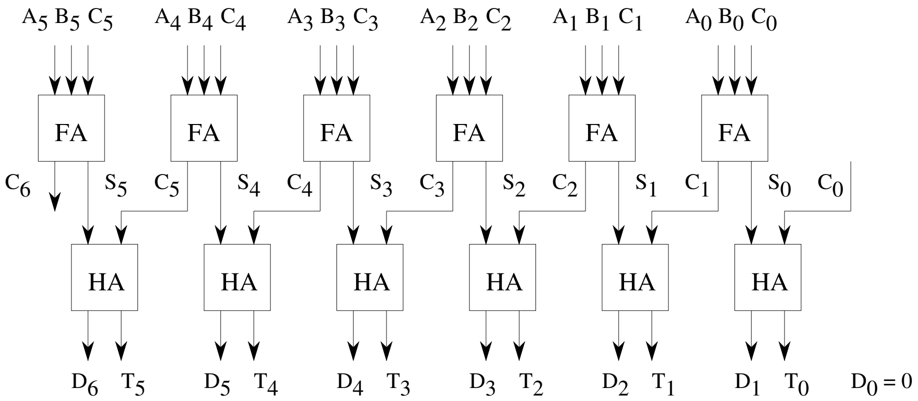
As an example, let $A = 40$, $B = 25$, and $C = 20$. In the first level, we compute the carry save pair (C, S) using the carry save equations. In the second level, we compute the carry delayed pair (D, T) using the definitions $D_{i+1} = S_iC_i$ and $T_i = S_i \oplus C_i$ as

$A = 40$	$=$		1	0	1	0	0	0
$B = 25$	$=$		0	1	1	0	0	1
$C = 20$	$=$		0	1	0	1	0	0
$S = 37$	$=$		1	0	0	1	0	1
$C = 48$	$=$	0	1	1	0	0	0	0
$T = 21$	$=$		0	1	0	1	0	1
$D = 64$	$=$	1	0	0	0	0	0	0

Thus, the carry delayed pair $(64, 21)$ represents the total of $A + B + C = 85$. The property of the carry delayed pair that $T_i D_{i+1} = 0$ for all $i = 0, 1, \ldots, k-1$ also holds.

$T = 21$	$=$		0	1	0	1	0	1
$D = 64$	$=$	1	0	0	0	0	0	0
$T_i D_{i+1}$	$=$		0	0	0	0	0	0

We will explore this property in Section 7.3 to design an efficient modular multiplier which was introduced by Brickell [3]. The following figure illustrates the carry delayed adder for $k = 6$.



6 Modular Addition Operation

The modular addition problem is defined as the computation of $S = A + B \pmod{n}$ given the integers A, B, and n. It is usually assumed that A and B are positive integers with $0 \leq A, B < n$, i.e., they are least positives residues. The most common method of computing S is as follows:

1. First compute $S' = A + B$.
2. Then compute $S'' = S' - n$.
3. If $S'' \geq 0$, then $S = S'$ else $S = S''$.

Thus, in addition to the availability of a regular adder, we need fast sign detection which is easy for the CPA, but somewhat harder for the CSA. However, when a CSA is used, the first two steps of the above algorithm can be combined, in other words, $S' = A + B$ and $S'' = A + B - n$ can be computed at the same time. Then, we perform a sign detection to decide whether to take S' or S'' as the correct sum. We will review algorithms of this type when we study modular multiplication algorithms.

6.1 Omura's Method

An efficient method computing the modular addition, which especially useful for multi-operand modular addition was proposed by Omura in [23]. Let $n < 2^k$. This method allows a temporary value to grow larger than n, however, it is always kept less than 2^k. Whenever it exceeds 2^k, the carry-out is ignored and a correction is performed. The correction factor is $m = 2^k - n$, which is precomputed and saved in a register. Thus, Omura's method performs the following steps given the integers $A, B < 2^k$ (but they can be larger than n).

1. First compute $S' = A + B$.
2. If there is a carry-out (of the kth bit), then $S = S' + m$, else $S = S'$.

The correctness of Omura's algorithm follows from the observations that

- If there is no carry-out, then $S = A + B$ is returned. The sum S is less than 2^k, but may be larger than n. In a future computation, it will be brought below n if necessary.
- If there is a carry-out, then we ignore the carry-out, which means we compute

  $$S' = A + B - 2^k \ .$$

  The result, which needs to be reduced modulo n, is in effect reduced modulo 2^k. We correct the result by adding m back to it, and thus, compute

  $$\begin{aligned} S &= S' + m \\ &= A + B - 2^k + m \\ &= A + B - 2^k + 2^k - n \\ &= A + B - n \ . \end{aligned}$$

After all additions are completed, a final result is reduced modulo n by using the standard technique. As an example, let assume $n = 39$. Thus, we have $m = 2^6 - 39 = 25 = (011001)$. The modular addition of $A = 40$ and $B = 30$ is performed using Omura's method as follows:

$$\begin{array}{lcrrl} A & = & 40 = & (101000) & \\ B & = & 30 = & (011110) & \\ S' & = & A + B = & 1(000110) & \text{Carry-out} \\ m & = & & (011001) & \\ S & = & S' + m = & (011111) & \text{Correction} \end{array}$$

Thus, we obtain the result as $S = (011111) = 31$ which is equal to $70 \pmod{39}$ as required. On the other hand, the addition of $A = 23$ by $B = 26$ is performed as

$$\begin{array}{rcrrl} A & = & 23 = & (010111) & \\ B & = & 26 = & (011010) & \\ S' & = & A + B = & 0(110001) & \text{No carry-out} \\ S & = & S' = & (110001) & \end{array}$$

This leaves the result as $S = (110001) = 49$ which is larger than the modulus 39. It will be reduced in a further step of the multioperand modulo addition. After all additions are completed, a final negative result can be corrected by adding m to it. For example, we correct the above result $S = (110001)$ as follows:

$$\begin{array}{rcrr} S & = & & (110001) \\ m & = & & (011001) \\ S & = & S + m = & 1(001010) \\ S & = & & (001010) \end{array}$$

The result obtained is $S = (001010) = 10$, which is equal to 49 modulo 39, as required.

7 Modular Multiplication Operation

The modular multiplication problem is defined as the computation of $P = AB \pmod{n}$ given the integers A, B, and n. It is usually assumed that A and B are positive integers with $0 \leq A, B < n$, i.e., they are the least positive residues. There are basically four approaches for computing the product P.

- Multiply and then divide.
- The steps of the multiplication and reduction are interleaved.
- Brickell's method.
- Montgomery's method.

The multiply-and-divide method first multiplies A and B to obtain the $2k$-bit number

$$P' := AB \ .$$

Then, the result P' is divided (reduced) by n to obtain the k-bit number

$$P := P' \ \% \ n \ .$$

We will not study the multiply-and-divide method in detail since the interleaving method is more suitable and also more efficient for our problem. The multiply-and-divide method is useful only when one needs the product P'.

7.1 Interleaving Multiplication and Reduction

The interleaving algorithm has been known. The details of the method are sketched in papers [1, 29]. Let A_i and B_i be the bits of the k-bit positive integers A and B, respectively. The product P' can be written as

$$\begin{aligned} P' &= A \cdot B = A \cdot \sum_{i=0}^{k-1} B_i 2^i = \sum_{i=0}^{k-1} (A \cdot B_i) 2^i \\ &= 2(\cdots 2(2(0 + A \cdot B_{k-1}) + A \cdot B_{k-2}) + \cdots) + A \cdot B_0 \end{aligned}$$

This formulation yields the shift-add multiplication algorithm. We also reduce the partial product modulo n at each step:

1. $P := 0$
2. **for** $i = 0$ **to** $k-1$
2a. $\quad P := 2P + A \cdot B_{k-1-i}$
2b. $\quad P := P \bmod n$
3. **return** P

Assuming that $A, B, P < n$, we have

$$\begin{aligned} P &:= 2P + A \cdot B_j \\ &\leq 2(n-1) + (n-1) \ = \ 3n - 3 \ . \end{aligned}$$

Thus, the new P will be in the range $0 \leq P \leq 3n-3$, and at most 2 subtractions are needed to reduce P to the range $0 \leq P < n$. We can use the following algorithm to bring P back to this range:

$P' := P - n$; If $P' \geq 0$ then $P = P'$
$P' := P - n$; If $P' \geq 0$ then $P = P'$

The computation of P requires k steps, at each step we perform the following operations:

- A left shift: $2P$
- A partial product generation: $A \cdot B_j$
- An addition: $P := 2P + A \cdot B_j$
- At most 2 subtractions:
  $P' := P - n$; If $P' \geq 0$ then $P = P'$
  $P' := P - n$; If $P' \geq 0$ then $P = P'$

The left shift operation is easily performed by wiring. The partial products, on the other hand, are generated using an array of AND gates. The most crucial operations are the addition and subtraction operations: they need to be performed fast. We have the following avenues to explore:

- We can use the carry propagate adder, introducing $O(k)$ delay per step. However, Omura's method can be used to avoid unnecessary subtractions:

  2a. $P := 2P$
  2b. If carry-out then $P := P + m$
  2c. $P := P + A \cdot B_j$
  2d. If carry-out then $P := P + m$

- We can use the carry save adder, introducing only $O(1)$ delay per step. However, recall that the sign information is not immediately available in the CSA. We need to perform fast sign detection in order to determine whether the partial product needs to be reduced modulo n.

7.2 Utilization of Carry Save Adders

In order to utilize the carry save adders in performing the modular multiplication operations, we represent the numbers as the carry save pairs (C, S), where the value of the number is the sum $C + S$. The carry save adder method of the interleaving algorithm is given below:

1. $(C, S) := (0, 0)$
2. **for** $i = 0$ **to** $k - 1$
2a. $(C, S) := 2C + 2S + A \cdot B_{k-1-i}$
2b. $(C', S') := C + S - n$
2c. **if** SIGN ≥ 0 **then** $(C, S) := (C', S')$
3. **return** (C, S)

The function SIGN gives the sign of the carry save number $C' + S'$. Since the exact sign is available only when a full addition is performed, we calculate an estimated sign with the SIGN function. A sign estimation algorithm was introduced in [15]. Here, we briefly review this algorithm, which is based on the addition o the most significant t bits of C and S to estimate the sign of $C + S$. For example, let $C = (011110)$ and $S = (001010)$, then the function SIGN produces

$$
\begin{array}{rcl}
C & = & 011110 \\
S & = & 001010 \\
(t=1) \ \text{SIGN} & = & \underline{0} \\
(t=2) \ \text{SIGN} & = & \underline{0}1 \\
(t=3) \ \text{SIGN} & = & \underline{1}00 \\
(t=4) \ \text{SIGN} & = & \underline{1}001 \\
(t=5) \ \text{SIGN} & = & \underline{1}0100 \\
(t=6) \ \text{SIGN} & = & \underline{1}01000 \ .
\end{array}
$$

In the worst case the exact sign is produced after adding all k bits. If the exact sign of $C + S$ is computed, we can obtain the result of the multiplication operation in the correct range $[0, N)$. If an estimation of the sign is used, then we will prove that the range of the result becomes $[0, N+\Delta)$, where Δ depends on the precision of the estimation. Furthermore, since the sign is used to decide whether some multiple of N should be subtracted from the partial product, an error in the decision causes only an error of a multiple of N in the partial product, which is corrected later. We define function $T(X)$ on an n-bit integer X as

$$T(X) = X - (X \bmod 2^t) \ ,$$

where $0 \leq t \leq n-1$. In other words, T replaces the first least significant t bits of X with t zeros. This implies

$$T(X) \leq X < T(X) + 2^t \ .$$

We reduce the pair (C, S) by performing the following operation Q times:

I. $(\hat{C}, \hat{S}) := C + S - N$.
J. If $T(\hat{C}) + T(\hat{S}) \geq 0$ then set $C := \hat{C}$ and $S := \hat{S}$.

In Step J, the computation of the sign bit R of $T(\hat{C}) + T(\hat{S})$ involves $n - t$ most significant bits of $\hat{C}$ and $\hat{S}$. The above procedure reduces a carry-sum pair from the range

$$0 \leq C_0 + S_0 < (Q+1)N + 2^t$$

to the range

$$0 \leq C_R + S_R < N + 2^t \ ,$$

where (C_0, S_0) and (C_R, S_R) respectively denote the initial and the final carry-sum pair. Since the function T always underestimates, the result is never over-reduced, i.e.,

$$C_R + S_R \geq 0 \ .$$

If the estimated sign in Step J is positive for all Q iterations, then QN is subtracted from the initial pair; therefore

$$C_R + S_R = C_0 + S_0 - QN < N + 2^t \ .$$

If the estimated sign becomes negative in an iteration, it stays negative thereafter to the last iteration. Thus, the condition

$$T(\hat{C}) + T(\hat{S}) < 0$$

in the last iteration of Step J implies that

$$T(\hat{C}) + T(\hat{S}) \leq -2^t \ ,$$

since $T(X)$ is always a multiple of 2^t. Thus, we obtain the range of $\hat{C}$ and $\hat{S}$ as

$$T(\hat{C}) + T(\hat{S}) \leq \hat{C} + \hat{S} < T(\hat{C}) + T(\hat{S}) + 2^{t+1} \ .$$

It follows from the above equations that

$$\hat{C} + \hat{S} < 2^{t+1} - 2^t = 2^t \ .$$

Since in Step I we perform $(\hat{C}, \hat{S}) := C + S - N$ and in the last iteration the carry-sum pair is not reduced (because the estimated sign is negative), we must have

$$C_R + S_R = \hat{C} + \hat{S} + N \ ,$$

which implies

$$C_R + S_R < N + 2^t \ .$$

The modular reduction procedure described above subtracts N from (C, S) in each of the Q iterations. The procedure can be improved in speed by subtracting $2^{k-j}N$ during iteration j, where $(Q+1) \leq 2^k$ and $j = 1, 2, 3, \ldots, k$. For example, if $Q = 3$, then $k = 2$ can be used. Instead of subtracting N three times, we first subtract $2N$ and then N. This observation is utilized in the following algorithm:

1. Set $S^{(0)} := 0$ and $C^{(0)} := 0$.
2. Repeat 2a, 2b, and 2c for $i = 1, 2, 3, \ldots, k$
 2a. $(C^{(i)}, S^{(i)}) := 2C^{(i-1)} + 2S^{(i-1)} + A_{n-i}B$.
 2b. $(\hat{C}^{(i)}, \hat{S}^{(i)}) := C^{(i)} + S^{(i)} - 2N$.
 If $T(\hat{C}^{(i)}) + T(\hat{S}^{(i)}) \geq 0$, then set $C^{(i)} := \hat{C}^{(i)}$ and $S^{(i)} := \hat{S}^{(i)}$.
 2c. $(\hat{C}^{(i)}, \hat{S}^{(i)}) := C^{(i)} + S^{(i)} - N$.
 If $T(\hat{C}^{(i)}) + T(\hat{S}^{(i)}) \geq 0$, then set $C^{(i)} := \hat{C}^{(i)}$ and $S^{(i)} := \hat{S}^{(i)}$.
3. End.

The parameter t controls the precision of estimation; the accuracy of the estimation and the total amount of logic required to implement it decreases as t increases. After Step 2c, we have

$$C^{(i)} + S^{(i)} < N + 2^t \ ,$$

which implies that after the next shift-add step the range of $C^{(i+1)} + S^{(i+1)}$ will be $[0, 3N + 2^{t+1})$. Assuming $Q = 3$, we have

$$3N + 2^{t+1} \leq (Q+1)N + 2^t = 4N + 2^t \ ,$$

which implies $2^t \leq N$, or $t \leq n - 1$. The range of $C^{(i+1)} + S^{(i+1)}$ becomes

$$0 \leq C^{(i+1)} + S^{(i+1)} < 3N + 2^{t+1} \leq 3N + 2^n \leq 2^{n+2} \ ,$$

and after Step 2b, the range will be

$$-2^{n+1} \leq -2N \leq C^{(i+1)} + S^{(i+1)} < N + 2^n < 2^{n+1} \ .$$

In order to contain the temporary results, we use $(n+3)$-bit carry save adders which can represent integers in the range $[-2^{n+2}, 2^{n+2})$. When $t = n - 1$, the sign estimation technique checks 5 most significant bits of $\hat{C}^{(i)}$ and $\hat{S}^{(i)}$ from the bit locations $n-2$ to $n+3$. This algorithm produces a pair of integers $(C,S)=(C^{(n)},S^{(n)})$ such that $P=C+S$ is in the range $[0,2N)$. The final result in the correct range $[0,N)$ can be obtained by computing $P=C+S$ and $\hat{P}=C+S-N$ using carry propagate adders. If $\hat{P}<0$, we have $P=\hat{P}+N<N$, and thus P is in the correct range. Otherwise, we choose $\hat{P}$ because $0\le\hat{P}=P-N<2^t<N$ implies $\hat{P}\in[0,N)$. The steps of the algorithm for computing $47\cdot 48 \pmod{50}$ are illustrated in the following figure. Here we have

$$\begin{aligned} k &= \lfloor \log_2(50) \rfloor + 1 \;=\; 6 \ , \\ A=47 &= (000101111) \ , \\ B=48 &= (000110000) \ , \\ N=50 &= (000110010) \ , \\ M=-N &= (111001110) \ . \end{aligned}$$

The algorithm computes the final result

$$(C,S)=(010111000,110000000)=(184,-128)$$

in $3k=18$ clock cycles. The range of $C+S=184-128=56$ is $[0,2\cdot 50)$. The final result is found by computing $C+S=56$ and $C+S-N=6$, and selecting the latter since it is positive.

		C	S	$\hat{C}$	$\hat{S}$	$T(\hat{C})+T(\hat{S})$	R
$i=0$		000000000	000000000	–	–	–	–
$i=1$	2a	000000000	000110000	–	–	–	–
	2b	000000000	000110000	000100000	110101100	111000000	1
	2c	000000000	000110000	000000000	111111110	111100000	1
$i=2$	2a	000000000	001100000	–	–	–	–
	2b	000000000	001100000	000000000	111111100	111100000	1
	2c	010000000	110101110	010000000	110101110	000100000	0
$i=3$	2a	000100000	001101100	–	–	–	–
	2b	001011000	111010000	001011000	111010000	000000000	0
	2c	001011000	111010000	110110000	001000110	111100000	1
$i=4$	2a	101100000	100100000	–	–	–	–
	2b	001000000	111011100	001000000	111011100	000000000	0
	2c	001000000	111011100	110011000	001010010	111000000	1
$i=5$	2a	101100000	100001000	–	–	–	–
	2b	101100000	100001000	000010000	111110100	111100000	1
	2c	010010000	110100110	010010000	110100110	000100000	0
$i=6$	2a	001000000	001011100	–	–	–	–
	2b	010111000	110000000	010111000	110000000	000100000	0
	2c	010111000	110000000	100010000	011110110	111100000	1

7.3 Brickell's Method

This method is based on the use of a carry delayed integer introduced in Section 5.6. Let A be a carry delayed integer, then, it can be written as

$$A = \sum_{i=0}^{k-1} (T_i + D_i) \cdot 2^i \ .$$

The product $P = AB$ can be computed by summing the terms:

$$\begin{array}{c}
(T_0 \cdot B + D_0 \cdot B) \cdot 2^0 + \\
(T_1 \cdot B + D_1 \cdot B) \cdot 2^1 + \\
(T_2 \cdot B + D_2 \cdot B) \cdot 2^2 + \\
\vdots \\
(T_{k-1} \cdot B + D_{k-1} \cdot B) \cdot 2^{k-1}
\end{array}$$

Since $D_0 = 0$, we rearrange to obtain

$$\begin{array}{c}
2^0 \cdot T_0 \cdot B + 2^1 \cdot D_1 \cdot B + \\
2^1 \cdot T_1 \cdot B + 2^2 \cdot D_2 \cdot B + \\
2^2 \cdot T_2 \cdot B + 2^3 \cdot D_3 \cdot B + \\
\vdots \\
2^{k-2} \cdot T_{k-2} \cdot B + 2^{k-1} \cdot D_{k-1} \cdot B + \\
2^{k-1} \cdot T_{k-1} \cdot B
\end{array}$$

Also recall that either T_i or D_{i+1} is zero due to the property of the carry delayed adder. Thus, each step requires a shift of B and addition of at most 2 carry delayed integers:

- Either: $(P_d, P_t) := (P_d, P_t) + 2^i \cdot T_i \cdot B$
- Or: $(P_d, P_t) := (P_d, P_t) + 2^{i+1} \cdot D_{i+1} \cdot B$

After k steps $P = (P_d, P_t)$ is obtained. In order to compute $P \pmod{n}$, we perform reduction:

$$\begin{array}{c}
\text{If} \quad P \geq 2^{k-1} \cdot n \ \text{then} \quad P := P - 2^{k-1} \cdot n \\
\text{If} \quad P \geq 2^{k-2} \cdot n \ \text{then} \quad P := P - 2^{k-2} \cdot n \\
\text{If} \quad P \geq 2^{k-3} \cdot n \ \text{then} \quad P := P - 2^{k-3} \cdot n \\
\vdots \\
\text{If} \quad P \geq n \ \text{then} \quad P := P - n
\end{array}$$

We can also reverse these steps to obtain:

$$\begin{array}{rcl}
P & := & T_{k-1} \cdot B \cdot 2^{k-1} \\
P & := & P + T_{k-2} \cdot B \cdot 2^{k-2} + D_{k-1} \cdot B \cdot 2^{k-1} \\
P & := & P + T_{k-3} \cdot B \cdot 2^{k-3} + D_{k-2} \cdot B \cdot 2^{k-2} \\
 & \vdots & \\
P & := & P + T_1 \cdot B \cdot 2^1 + D_2 \cdot B \cdot 2^2 \\
P & := & P + T_0 \cdot B \cdot 2^0 + D_1 \cdot B \cdot 2^1
\end{array}$$

Also, the multiplication steps can be interleaved with reduction steps. To perform the reduction, the sign of $P - 2^i \cdot n$ needs to be determined (estimated). Brickell's solution [3] is essentially a combination of the sign estimation technique and Omura's method of correction. We allow enough bits for P, and whenever P exceeds 2^k, add $m = 2^k - n$ to correct the result. 11 steps after the multiplication procedure started, the algorithm starts subtracting multiples of n. In the following, P is a carry delayed integer of $k + 11$ bits, m is a binary integer of k bits, and t_1 and t_2 control bits, whose initial values are $t_1 = t_2 = 0$.

1. Add the most significant 4 bits of P and $m \cdot 2^{11}$.
2. If overflow is detected, then $t_2 = 1$ else $t_2 = 0$.
3. Add the most significant 4 bits of P and the most significant 3 bits of $m \cdot 2^{10}$.
4. If overflow is detected and $t_2 = 0$, then $t_1 = 1$ else $t_1 = 0$.

The multiplication and reduction steps of Brickell's algorithm are as follows:

$$
\begin{aligned}
B' &:= T_i \cdot B + 2 \cdot D_{i+1} \cdot B \\
m' &:= t_2 \cdot m \cdot 2^{11} + t_1 \cdot m \cdot 2^{10} \\
P &:= 2(P + B' + m') \\
A &:= 2A \ .
\end{aligned}
$$

7.4 Montgomery's Method

The Montgomery algorithm computes

$$
\text{MonPro}(A, B) = A \cdot B \cdot r^{-1} \bmod n
$$

given $A, B < n$ and r such that $\gcd(n, r) = 1$. Even though the algorithm works for any r which is relatively prime to n, it is more useful when r is taken to be a power of 2, which is an intrinsically fast operation on general-purpose computers, e.g., signal processors and microprocessors. To find out why the above computation is useful for computing the modular exponentiation, we refer the reader to the companion report [12]. In this section, we introduce an efficient binary add-shift algorithm for computing MonPro(A, B), and then generalize it to the m-ary method. We take $r = 2^k$, and assume that the number of bits in A or B is less than k. Let $A = (A_{k-1}A_{k-2} \cdots A_0)$ be the binary representation of A. The above product can be written as

$$
2^{-k} \cdot (A_{k-1}A_{k-2} \cdots A_0) \cdot B = 2^{-k} \cdot \sum_{i=0}^{k-1} A_i \cdot 2^i \cdot B \pmod{n} \ .
$$

The product $t = (A_0 + A_1 2 + \cdots A_{k-1} 2^{k-1}) \cdot B$ can be computed by starting from the most significant bit, and then proceeding to the least significant, as follows:

1. $t := 0$
2. for $i = k-1$ to 0
2a. $t := t + A_i \cdot B$
2b. $t := 2 \cdot t$

The shift factor 2^{-k} in $2^{-k} \cdot A \cdot B$ reverses the direction of summation. Since

$$2^{-k} \cdot (A_0 + A_1 2 + \cdots A_{k-1} 2^{k-1}) \ = \ A_{k-1} 2^{-1} + A_{k-2} 2^{-2} \cdots A_0 2^{-k} \ ,$$

we start processing the bits of A from the least significant, and obtain the following binary add-shift algorithm to compute $t = A \cdot B \cdot 2^{-k}$.

1. $t := 0$
2. for $i = 0$ to $k-1$
2a. $t := t + A_i \cdot B$
2b. $t := t/2$

The above summation computes the product $t = 2^{-k} \cdot A \cdot B$, however, we are interested in computing $u = 2^{-k} \cdot A \cdot B \pmod{n}$. This can be achieved by subtracting n during every add-shift step, but there is a simpler way: We add n to u if u is odd, making new u an even number since n is always odd. If u is even after the addition step, it is left untouched. Thus, u will always be even before the shift step, and we can compute

$$u := u \cdot 2^{-1} \pmod{n}$$

by shifting the even number u to the right since $u = 2v$ implies

$$u := 2v \cdot 2^{-1} = v \pmod{n} \ .$$

The binary add-shift algorithm computes the product $u = A \cdot B \cdot 2^{-k} \pmod{n}$ as follows:

1. $u := 0$
2. for $i = 0$ to $k-1$
2a. $u := u + A_i \cdot B$
2b. If u is odd then $u := u + n$
2c. $u := u/2$

We reserve a $(k+1)$-bit register for u because if u has k bits at beginning of an add-shift step, the addition of $A_i \cdot B$ and n (both of which are k-bit numbers) increases its length to $k+1$ bits. The right shift operation then brings it back to k bits. After k add-shift steps, we subtract n from u if it is larger than n.

Also note that Steps 2a and 2b of the above algorithm can be combined: We can compute the least significant bit u_0 of u before actually computing the sum in Step 2a. It is given as

$$u_0 := u_0 \oplus (A_i B_0) \ .$$

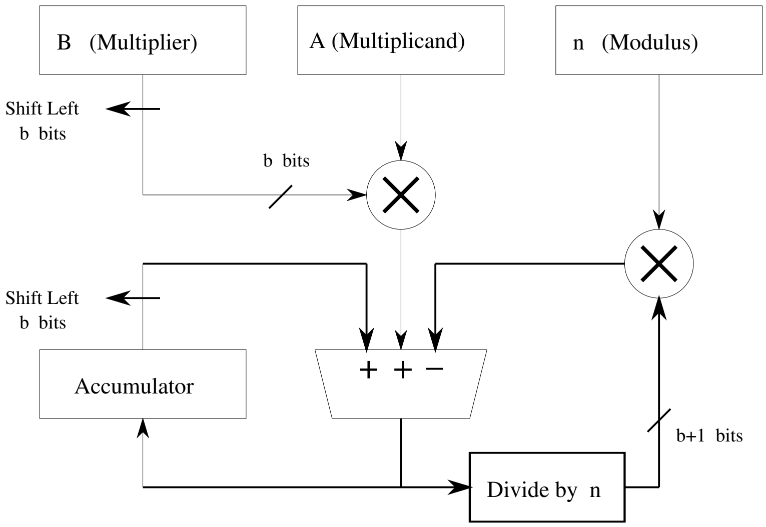
Thus, we decide whether u is odd prior to performing the full addition operation $u := u + A_i B$. This is the most important property of Montgomery's method. In contrast, the classical modular multiplication algorithms (e.g., the interleaving method) computes the entire sum in order to decide whether a reduction needs to be performed.

7.5 High-Radix Interleaving Method

Since the speed for radix 2 multipliers is approaching limits, the use of higher radices is investigated. High-radix operations require fewer clock cycles, however, the cycle time and the required area increases. Let 2^b be the radix. The key operation in computing $P = AB \pmod{n}$ is the computation of an inner-product steps coupled with modular reduction, i.e., the computation of

$$P := 2^b \cdot P + A \cdot B_i - Q \cdot n \ ,$$

where P is the partial product and B_i is the ith digit of B in radix 2^b. The value of Q determines the number of times the modulus n is subtracted from the partial product P in order to reduce it modulo n. We compute Q by dividing the current value of the partial product P by n, which is then multiplied by n and subtracted from the partial product during the next cycle. This implementation is illustrated in the following figure.



For the radix 2, the partial product generation is performed using an array of AND gates. The partial product generation is much more complex for higher radices, e.g., Wallace trees and generalized counters need to be used. However, the generation of the high-radix partial products does not greatly increase cycle time since this computation can be easily pipelined. The most complicated step is the reduction step, which necessitates more complex routing, increasing the chip area.

7.6 High-Radix Montgomery's Method

The binary add-shift algorithm is generalized to higher radix (m-ary) algorithm by proceeding word by word, where the wordsize is w bits, and $k = sw$. The addition step is performed by multiplying one word of A by B and the right shift is performed by shifting w bits to the right. In order to perform an exact division of u by 2^w, we add an integer multiple of n to u, so that the least significant word of the new u will be zero. Thus, if $u \neq 0 \pmod{2^w}$, we find an integer m such that $u + m \cdot n = 0 \pmod{2^w}$. Let u_0 and n_0 be the least significant words of u and n, respectively. We calculate m as

$$m = -u_0 \cdot n_0^{-1} \pmod{2^w} .$$

The word-level (m-ary) add-shift Montgomery product algorithm is given below:

1.	$u := 0$
2.	for $i = 0$ to $s-1$
2a.	$\quad u := u + A_i \cdot B$
2b.	$\quad m := -u_0 \cdot n_0^{-1} \bmod 2^w$
2c.	$\quad u := u + m \cdot n$
2d.	$\quad u := u/2^w$

This algorithm specializes to the binary case by taking $w = 1$. In this case, when u is odd, the least significant bit u_0 is nonzero, and thus, $m = -u_0 \cdot n_0^{-1} = 1 \pmod{2}$.

References

[1] G. R. Blakley. A computer algorithm for the product AB modulo M. *IEEE Transactions on Computers*, 32(5):497–500, May 1983.

[2] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31(3):260–264, March 1982.

[3] E. F. Brickell. A fast modular multiplication algorithm with application to two key cryptography. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *Advances in Cryptology, Proceedings of Crypto 82*, pages 51–60. New York, NY: Plenum Press, 1982.

[4] E. F. Brickell. A survey of hardware implementations of RSA. In G. Brassard, editor, *Advances in Cryptology — CRYPTO 89, Proceedings*, Lecture Notes in Computer Science, No. 435, pages 368–370. New York, NY: Springer-Verlag, 1989.

[5] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT 92*, Lecture Notes in Computer Science, No. 658, pages 200–207. New York, NY: Springer-Verlag, 1992.

[6] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[7] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[8] B. Gilchrist, J. H. Pomerene, and S. Y. Wong. Fast carry logic for digital computers. *IRE Transactions on Electronic Computers*, 4:133–136, 1955.

[9] F. Hoornaert, M. Decroos, J. Vandewalle, and R. Govaerts. Fast RSA-hardware: dream or reality? In C. G. Gunther, editor, *Advances in Cryptology — EUROCRYPT 88*, Lecture Notes in Computer Science, No. 330, pages 257–264. New York, NY: Springer-Verlag, 1988.

[10] K. Hwang. *Computer Arithmetic, Principles, Architecture, and Design.* New York, NY: John Wiley & Sons, 1979.

[11] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Reading, MA: Addison-Wesley, Second edition, 1981.

[12] Ç. K. Koç. High-Speed RSA Implementation. Technical Report TR 201, RSA Laboratories, November 1994.

[13] Ç. K. Koç and C. Y. Hung. Carry save adders for computing the product AB modulo N. *Electronics Letters*, 26(13):899–900, 21st June 1990.

[14] Ç. K. Koç and C. Y. Hung. Multi-operand modulo addition using carry save adders. *Electronics Letters*, 26(6):361–363, 15th March 1990.

[15] Ç. K. Koç and C. Y. Hung. Bit-level systolic arrays for modular multiplication. *Journal of VLSI Signal Processing*, 3(3):215–223, 1991.

[16] M. Kochanski. Developing an RSA chip. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pages 350–357. New York, NY: Springer-Verlag, 1985.

[17] I. Koren. *Computer Arithmetic Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[18] D. C. Kozen. *The Design and Analysis of Algorithms*. New York, NY: Springer-Verlag, 1992.

[19] R. Ladner and M. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[20] S. Lakshmivarahan and S. K. Dhall. *Parallelism in the Prefix Problem*. Oxford, London: Oxford University Press, 1994. In press.

[21] J. D. Lipson. *Elements of Algebra and Algebraic Computing*. Reading, MA: Addison-Wesley, 1981.

[22] National Institute for Standards and Technology. Digital signature standard (DSS). *Federal Register*, 56:169, August 1991.

[23] J. K. Omura. A public key cell design for smart card chips. In *International Symposium on Information Theory and its Applications*, pages 983–985, Hawaii, USA, November 27–30, 1990.

[24] R. L. Rivest. RSA chips (Past/Present/Future). In T. Beth, N. Cot, and I. Ingemarsson, editors, *Advances in Cryptology, Proceedings of EUROCRYPT 84*, Lecture Notes in Computer Science, No. 209, pages 159–165. New York, NY: Springer-Verlag, 1984.

[25] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[26] RSA Laboratories. Answers to Frequently Asked Questions About Today's Cryptography. RSA Data Security, Inc., October 1993.

[27] RSA Laboratories. The Public-Key Cryptography Standards (PKCS). RSA Data Security, Inc., November 1993.

[28] H. Sedlak. The RSA cryptography processor. In D. Chaum and W. L. Price, editors, *Advances in Cryptology — EUROCRYPT 87*, Lecture Notes in Computer Science, No. 304, pages 95–105. New York, NY: Springer-Verlag, 1987.

[29] K. R. Sloan, Jr. Comments on "A computer algorithm for the product AB modulo M". *IEEE Transactions on Computers*, 34(3):290–292, March 1985.

[30] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, editors. *Residue Arithmetic: Modern Applications in Digital Signal Processing.* New York, NY: IEEE Press, 1986.

[31] E. E. Swartzlander, editor. *Computer Arithmetic*, volume I and II. Los Alamitos, CA: IEEE Computer Society Press, 1990.

[32] N. S. Szabo and R. I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology.* New York, NY: McGraw-Hill, 1967.

[33] C. D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–378, March 1993.

[34] S. Waser and M. J. Flynn. *Introduction to Arithmetic for Digital System Designers.* New York, NY: Holt, Rinehart and Winston, 1982.