# 5 Prime Finite Field Arithmetic

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 [64]. Furthermore, the ElGamal signature scheme [80] and the Digital Signature Standard (DSS) of the National Institute for Standards and Technology [90] 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 [35]. 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 modular algorithm on hardware platforms. We draw our material from computer arithmetic books [352, 138, 370, 187], collection of articles [75, 335], and journal and conference articles on hardware structures for performing the modular multiplication and exponentiations [288, 185, 322, 135, 34, 179, 180, 181, 365].

Therefore, in the remainder of this Chapter we will study algorithms for computing efficiently the most basic modular arithmetic operations. 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.

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 [337]. 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.

## 5.1 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:

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

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.

### 5.1.1 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 $i$th bits of the integers $A$ and $B$, respectively, and $C_i$ is the carry bit received by the $i$th 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

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

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 $i$th 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. Fig. 5.1 illustrates the FA and HA cells.

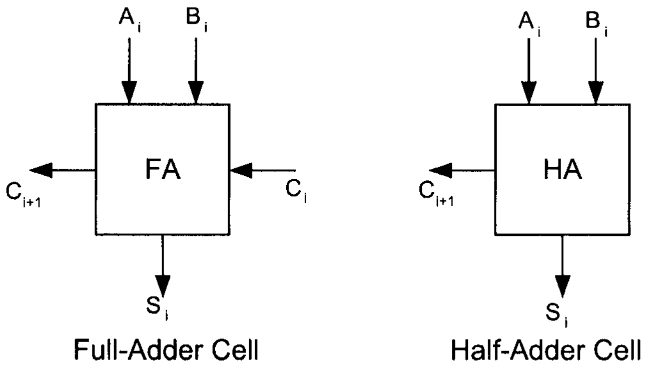

**Fig. 5.1.** Full-Adder and Half-Adder Cells

### 5.1.2 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 in Fig. 5.2 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 $i$th cell needs to receive the correct value

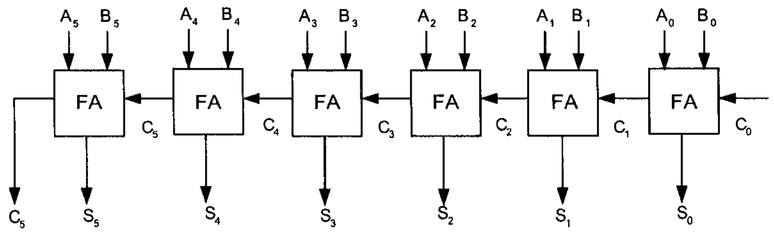

**Fig. 5.2.** Carry Propagate Adder

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 RSA computations, 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.

### 5.1.3 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 in Fig. 5.3.

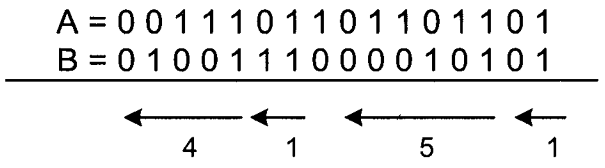
A = 0 0 1 1 1 0 1 1 0 1 1 0 1 1 0 1
B = 0 1 0 0 1 1 1 0 0 0 0 1 0 1 0 1
4
1
5
1

**Fig. 5.3.** Carry Completion Sensing Adder

A statistical analysis shows that the average longest carry sequence is approximately 4.6 for a 40-bit adder [108]. 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.

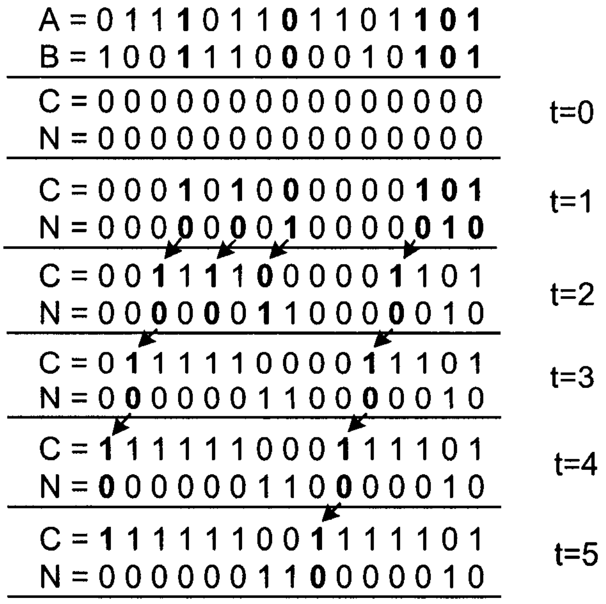
A = 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1
B = 1 0 0 1 1 1 0 0 0 0 1 0 1 1 0 1
C = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
N = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
t=0
C = 0 0 0 1 0 1 0 0 0 0 0 0 1 1 0 1
N = 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0
t=1
C = 0 0 1 1 1 1 0 0 0 0 0 1 1 1 0 1
N = 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0
t=2
C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

**Fig. 5.4.** Detecting Carry Completion

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 $i$th position is computed using the values of $A$ and $B$ for the $i$th position, and the previous $C$ and $N$ values, as follows:

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

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. The procedure described above is illustrated in Fig. 5.4.

## 5.1.4 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

$$C_2 = G_1 + C_1 P_1.$$

When we substitute $C_1$ in the above equation with the value of $C_0$ in the preceding equation, we obtain $C_2$ in terms $G_0$, $G_1$, $P_0$, $P_1$, and $C_0$ as

$$C_2 = G_1 + C_1 P_1 = G_1 + (G_0 + C_0 P_0) P_1 = G_1 + G_0 P_1 + C_0 P_0 P_1.$$

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 $C_i$s in advance, and then feeds these values to an array of EXOR gates to compute the sum vector $S$. The $i$th 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 in Fig. 5.5.

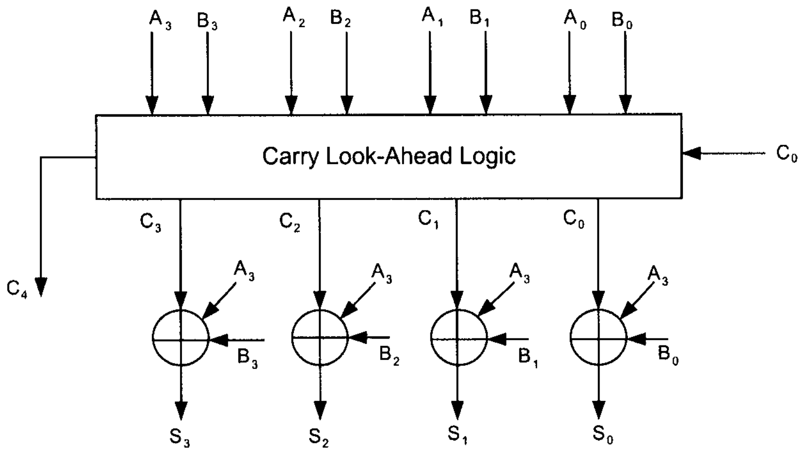

**Fig. 5.5.** Carry Look-Ahead Adder

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 [138, 370, 187].
- 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 [32, 188, 196].

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 [196, 197]. 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.

### 5.1.5 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:

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

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

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

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 in Fig. 5.6 for $k = 6$.

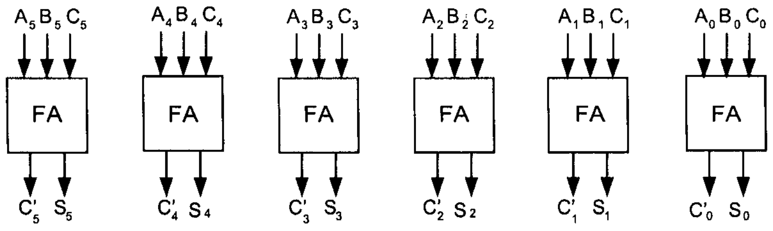

**Fig. 5.6.** Carry Save Adder

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.

$$
\begin{array}{rcrcccccccl}
A & = & -18 & = & & 1 & 0 & 1 & 1 & 1 & 0 & \\
B & = & 19 & = & & 0 & 1 & 0 & 0 & 1 & 1 & \\
C & = & 6 & = & & 0 & 0 & 0 & 1 & 1 & 0 & \\
\hline
S & = & -5 & = & & 1 & 1 & 1 & 0 & 1 & 1 & \\
C' & = & 12 & = & 0 & 0 & 0 & 1 & 1 & 0 & & \\
\hline
 & & & & & \mathbf{1} & & & & & & (1\ \mathrm{MSB}) \\
 & & & & & \mathbf{1} & 1 & & & & & (2\ \mathrm{MSB}) \\
 & & & & & \mathbf{0} & 0 & 0 & & & & (3\ \mathrm{MSB}) \\
 & & & & & \mathbf{0} & 0 & 0 & 1 & & & (4\ \mathrm{MSB}) \\
 & & & & & \mathbf{0} & 0 & 0 & 1 & 1 & & (5\ \mathrm{MSB}) \\
 & & & & & \mathbf{0} & 0 & 0 & 1 & 1 & 1 & (6\ \mathrm{MSB}) \\
\hline
\end{array}
$$

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.

### 5.1.6 Carry Delayed Adder

The carry delayed adder is a two-level carry save adder. As we will see in §5.3.6, 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:

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

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

$$\begin{array}{r@{\;=\;}r}
A = 40 & 1\,0\,1\,0\,0\,0 \\
B = 25 & 0\,1\,1\,0\,0\,1 \\
C = 20 & 0\,1\,0\,1\,0\,0 \\
\hline
S = 37 & 1\,0\,0\,1\,0\,1 \\
C = 48 & 0\,1\,1\,0\,0\,0\,0 \\
\hline
T = 21 & 0\,1\,0\,1\,0\,1 \\
D = 64 & 1\,0\,0\,0\,0\,0\,0 \\
\hline
\end{array}$$

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

$$\begin{array}{r@{\;=\;}r}
T = 21 & 0\,1\,0\,1\,0\,1 \\
D = 64 & 1\,0\,0\,0\,0\,0\,0 \\
\hline
T_iD_{i+1} & 0\,0\,0\,0\,0\,0 \\
\hline
\end{array}$$

We will explore this property in § 5.3.6 to design an efficient modular multiplier which was introduced by Brickell [33]. Fig. 5.7 illustrates the carry delayed adder for $k = 6$.

## 5.2 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

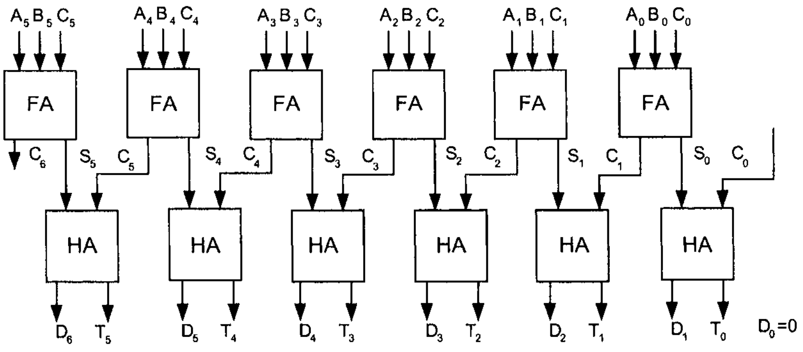

**Fig. 5.7.** Carry Delayed Adder

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.

### 5.2.1 Omura's Method

An efficient method computing the modular addition, which especially useful for multioperand modular addition was proposed by Omura in [260]. 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 $k$th 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:

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

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

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

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:

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

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

## 5.3 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

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

The result $P$ is a $k$-bit or $s$-word number.

The reduction is accomplished by dividing $P'$ by $n$, however, we are not interested in the quotient; we only need the remainder. The steps of the division algorithm can be somewhat simplified in order to speed up the process.

### 5.3.1 Standard Multiplication Algorithm

Let $A$ and $B$ be two $s$-digit ($s$-word) numbers expressed in radix $W$ as:

$$A = (A_{s-1}A_{s-2}\cdots A_0) = \sum_{j=0}^{s-1} A_i W^i,$$

$$B = (B_{s-1}B_{s-2}\cdots B_0) = \sum_{j=0}^{s-1} B_i W^i,$$

where the digits of $A$ and $B$ are in the range $[0, W-1]$. In general $W$ can be any positive number. For reconfigurable hardware implementations, we often select $W = 2^w$ where $w$ is the word-size or granularity of the device, e.g., $w = 4$. The standard (pencil-and-paper) algorithm for multiplying $A$ and $B$ produces the partial products by multiplying a digit of the multiplier ($B$) by the entire number $A$, and then summing these partial products to obtain the final number $2s$-word number $P'$. Let $P'_{ij}$ denote the (Carry,Sum) pair produced from the product $A_i \cdot B_j$. For example, when $W = 10$, and $A_i = 7$ and $B_j = 8$, then $P'_{ij} = (5, 6)$. The $P'_{ij}$ pairs can be arranged in a table as

$$\begin{array}{cccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 \\
\times & & & & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & & P'_{03} & P'_{02} & P'_{01} & P'_{00} \\
 & & & P'_{13} & P'_{12} & P'_{11} & P'_{10} & \\
 & & P'_{23} & P'_{22} & P'_{21} & P'_{20} & & \\
+ & P'_{33} & P'_{32} & P'_{31} & P'_{30} & & & \\
\hline
P'_7 & P'_6 & P'_5 & P'_4 & P'_3 & P'_2 & P'_1 & P'_0
\end{array}$$

The last row denotes the total sum of the partial products, and represents the product as an $2s$-word number. The standard algorithm for multiplication essentially performs the above digit-by-digit multiplications and additions. In order to save space, a single partial product variable $P'$ is being used. The initial value of the partial product is equal to zero; we then take a digit of $B$ and multiply by the entire number $A$, and add it to the partial product $P'$. The partial product variable $P'$ contains the final product $A \cdot B$ at the end of the computation. Algorithm 5.1 shows the standard procedure for computing the product $A \cdot B$.

**Algorithm 5.1** The Standard Multiplication Algorithm

```
Require: $A, B$.
Ensure: $P' = A \cdot B$.
 1: Initially $P'_i := 0$ for all $i = 0, 1, \ldots, 2s - 1$.
 2: for $i = 0$ to $s - 1$ do
 3:    $C := 0$;
 4:    for $j = 0$ to $s - 1$ do
 5:       $(C, S) := P'_{i+j} + A_j \cdot B_i + C$;
 6:       $P'_{i+j} := S$;
 7:    end for
 8:    $P'_{i+s} := C$;
 9: end for
10: Return($P'_{2s-1} P'_{2s-2} \cdots P'_0$)
```

In the following, we show the steps of the computation of $A \cdot B = 348 \cdot 857$ using the standard algorithm.

| $i$ | $j$ | Step | $(C,S)$ | Partial $P'$ |
|---|---|---|---|---|
| 0 | 0 | $P'_0 + A_0 b_0 + C$ | $(0,*)$ | 000000 |
| | | $0+8\cdot 7+0$ | $(5,6)$ | 000006 |
| | 1 | $P'_1 + A_1 b_0 + C$ | | |
| | | $0+4\cdot 7+5$ | $(3,3)$ | 000036 |
| | 2 | $P'_2 + A_2 b_0 + C$ | | |
| | | $0+3\cdot 7+3$ | $(2,4)$ | 000436 |
| | | | | 002436 |
| 1 | 0 | $P'_1 + A_0 b_1 + C$ | $(0,*)$ | |
| | | $3+8\cdot 5+0$ | $(4,3)$ | 002436 |
| | 1 | $P'_2 + A_1 b_1 + C$ | | |
| | | $4+4\cdot 5+4$ | $(2,8)$ | 002836 |
| | 2 | $P'_3 + A_2 b_1 + C$ | | |
| | | $2+3\cdot 5+2$ | $(1,9)$ | 009836 |
| | | | | 019836 |
| 2 | 0 | $P'_2 + A_0 b_2 + C$ | $(0,*)$ | |
| | | $8+8\cdot 8+0$ | $(7,2)$ | 019236 |
| | 1 | $P'_3 + A_1 b_2 + C$ | | |
| | | $9+4\cdot 8+7$ | $(4,8)$ | 018236 |
| | 2 | $P'_4 + A_2 b_2 + C$ | | |
| | | $1+3\cdot 8+4$ | $(2,9)$ | 098236 |
| | | | | 298236 |

In order to implement this algorithm, we need to be able to execute Step 5 of Algorithm 5.1 as,

$$(C, S) := P'_{i+j} + A_j \cdot B_i + C,$$

where the variables $P'_{i+j}$, $A_j$, $B_i$, $C$, and $S$ each hold a single-word, or a $W$-bit number. This step is termed as an inner-product operation which is common in many of the arithmetic and number-theoretic calculations. The inner-product operation above requires that we multiply two $W$-bit numbers and add this product to previous 'carry' which is also a $W$-bit number and then add this result to the running partial product word $P'_{i+j}$. From these three operations we obtain a $2W$-bit number since the maximum value is

$$2^W - 1 + (2^W - 1)(2^W - 1) + 2^W - 1 = 2^{2W} - 1.$$

Also, since the inner-product step is within the innermost loop, it needs to run as fast as possible. Of course, the best thing is to have a single microprocessor instruction for this computation; unfortunately, none of the currently available microprocessors and signal processors offers such a luxury. A brief inspection of the steps of this algorithm reveals that the total number of inner-product steps is equal to $s^2$. Since $s = k/w$ and $w$ is a constant on a given computer, the standard multiplication algorithm requires $O(k^2)$ bit operations in order to multiply two $k$-bit numbers.

## 5.3.2 Squaring is Easier

Squaring is an easier operation than multiplication since half of the single-precision multiplications can be skipped. This is due to the fact that $P'_{ij} = A_i \cdot A_j = P'_{ji}$.

$$
\begin{array}{ccccccccc}
 & & & & & A_3 & A_2 & A_1 & A_0 \\
\times & & & & & A_3 & A_2 & A_1 & A_0 \\
\hline
 & & & & & P'_{03} & P'_{02} & P'_{01} & P'_{00} \\
 & & & & P'_{13} & P'_{12} & P'_{11} & P'_{01} & \\
 & & & P'_{23} & P'_{22} & P'_{12} & P'_{02} & & \\
+ & & P'_{33} & P'_{23} & P'_{13} & P'_{03} & & & \\
\hline
 & & & & & 2P'_{03} & 2P'_{02} & 2P'_{01} & P'_{00} \\
 & & & & 2P'_{13} & 2P'_{12} & P'_{11} & & \\
 & & & 2P'_{23} & P'_{22} & & & & \\
+ & & P'_{33} & & & & & & \\
\hline
 & P'_7 & P'_6 & P'_5 & P'_4 & P'_3 & P'_2 & P'_1 & P'_0
\end{array}
$$

Thus, we can modify the standard multiplication procedure as shown in Algorithm 5.2 to take advantage of this property of the squaring operation.

**Algorithm 5.2** The Standard Squaring Algorithm

```
Require: $A$.
Ensure: $P' = A \cdot A$.
 1: Initially $P'_i := 0$ for all $i = 0, 1, \ldots, 2s-1$.
 2: for $i = 0$ to $s-1$ do
 3:    $(C, S) := P'_{i+i} + A_i \cdot A_i$
 4:    for $j = i+1$ to $s-1$ do
 5:       $(C, S) := P'_{i+j} + 2 \cdot A_j \cdot A_i + C$;
 6:       $P'_{i+j} := S$;
 7:    end for
 8:    $P'_{i+s} := C$;
 9: end for
10: Return($P'_{2s-1}P'_{2s-2} \cdots P'_0$)
```

However, we warn the reader that the carry-sum pair produced by operation

$$(C, S) := P'_{i+j} + 2 \cdot A_j \cdot A_i + C$$

in Step 5 of Algorithm 5.2 may be 1 bit longer than a single-precision number which requires $w$ bits. Since

$$(2^w - 1) + 2(2^w - 1)(2^w - 1) + (2^w - 1) = 2^{2w+1} - 2^{w+1}$$

and

$$2^{2w} - 1 < 2^{2w+1} - 2^{w+1} < 2^{2w+1} - 1,$$

the carry-sum pair requires $2w+1$ bits instead of $2w$ bits for its representation. Thus, we need to accommodate this 'extra' bit during the execution of the operations in Steps 5, 6, and 7 of Algorithm 5.2. The resolution of this carry may depend on the way the carry bits are handled by the particular processor's architecture. This issue, being rather implementation-dependent, will not be discussed here.

### 5.3.3 Modular Reduction

The multiply-and-reduce modular multiplication algorithm first computes the product $A \cdot B$ (or, $A \cdot A$) using one of the multiplication algorithms given above. The multiplication step is then followed by a division algorithm in order to compute the remainder. However, as we have mentioned before, we are not interested in the quotient; we only need the remainder. Therefore, the steps of the division algorithm can somewhat be simplified in order to speed up the process. The reduction step can be achieved by making one of the well-known sequential division algorithms. In the rest of this subsection, we describe the restoring and the nonrestoring division algorithms for computing the remainder of $P'$ when divided by $n$, where $n$ is a general modulus[1].

Division is the most complex of the four basic arithmetic operations. First of all, it has two results: the quotient and the remainder. Given a dividend $P'$ and a divisor $n$, a quotient $Q$ and a remainder $R$ have to be calculated in order to satisfy

$$P' = Q \cdot n + R \text{ with } R < n.$$

If $P'$ and $n$ are positive, then the quotient $Q$ and the remainder $R$ will be positive. The sequential division algorithm successively shifts and subtracts $n$ from $P'$ until a remainder $R$ with the property $0 \leq R < n$ is found. However, after a subtraction we may obtain a negative remainder. The restoring and nonrestoring algorithms take different actions when a negative remainder is obtained.

#### Restoring Division Algorithm

Let $R_i$ be the remainder obtained during the $i$th step of the division algorithm. Since we are not interested in the quotient, we ignore the generation of the bits of the quotient in the following algorithm. The procedure given below first left-aligns the operands $P'$ and $n$. Since $P'$ is $2k$-bit number and $n$ is a $k$-bit number, the left alignment implies that $n$ is shifted $k$ bits to the left, i.e., we start with $2^k n$. Furthermore, the initial value of $R$ is taken to be $P'$, i.e., $R_0 = P'$. We then subtract the shifted $n$ from $P'$ to obtain $R_1$; if $R_1$ is

[1] It is noted that Solinas proposed in [338] primes of special form for which the reduction step can be accomplished with high efficiency. However the material for Solinas special primes is not covered in this book. The interested reader may consult [37].

positive or zero, we continue to the next step. If it is negative the remainder is restored to its previous value as is shown in Algorithm 5.3 below.

**Algorithm 5.3** The Restoring Division Algorithm

```
Require: $P', n$.
Ensure: $R = P' \bmod n$.
1: $R_0 := t$;
2: $n := 2^k n$;
3: for $i = 1$ to $k$ do
4:     $R_i := R_{i-1} n$;
5:     if $R_i < 0$ then
6:         $R_i := R_{i-1}$;
7:     end if
8:     $n := n/2$
9: end for
10: Return($R_k$)
```

In Step 5 of Algorithm 5.3, we check the sign of the remainder; if it is negative, the previous remainder is taken to be the new remainder, i.e., a restore operation is performed. If the remainder $R_i$ is positive, it remains as the new remainder, i.e., we do not restore. The restoring division algorithm performs $k$ subtractions in order to reduce the $2k$-bit number $t$ modulo the $k$-bit number $n$. Thus, it takes much longer than the standard multiplication algorithm which requires $s = k/w$ inner-product steps, where $w$ is the word-size of granularity being employed.

In the following, we give an example of the restoring division algorithm for computing 3019 mod 53, where $3019 = (101111001011)_2$ and $53 = (110101)_2$. The result is $51 = (110011)_2$.

| | | |
|---|---|---|
| $R_0$ | 101111 001011 | $t$ |
| $n$ | 110101 | subtract |
| | − 000110 | negative remainder |
| $R_1$ | 101111 001011 | restore |
| $n/2$ | 11010 1 | shift and subtract |
| | + 10100 1 | positive remainder |
| $R_2$ | 10100 101011 | not restore |
| $n/2$ | 1101 01 | shift and subtract |
| | + 0111 01 | positive remainder |
| $R_3$ | 0111 011011 | not restore |
| $n/2$ | 110 101 | shift and subtract |
| | + 000 110 | positive remainder |
| $R_4$ | 000 110011 | not restore |
| $n/2$ | 11 0101 | shift |
| $n/2$ | 1 10101 | shift |
| $n/2$ | 110101 | shift and subtract |
| | + 000010 | negative remainder |
| $R_5$ | 110011 | restore |
| $R$ | 110011 | final remainder |

Also, before subtracting, we may check if the most significant bit of the remainder is 1. In this case, we perform a subtraction. If it is zero, there is no need to subtract since $n > R_i$. We shift $n$ until it is aligned with a nonzero most significant bit of $R_i$. This way we are able to skip several subtract/restore cycles. In the average, $k/2$ subtractions are performed.

### Nonrestoring Division Algorithm

The nonrestoring division algorithm allows a negative remainder. In order to correct the remainder, a subtraction or an addition is performed during the next cycle, depending on the whether the sign of the remainder is positive or negative, respectively. This is based on the following observation: Suppose $R_i = R_{i-1} - n < 0$, then the restoring algorithm assigns $R_i := R_{i-1}$ and performs a subtraction with the shifted $n$, obtaining

$$R_{i+1} = R_i - n/2 = R_{i-1} - n/2.$$

However, if $R_i = R_{i-1} - n < 0$, then one can instead let $R_i$ remain negative and add the shifted $n$ in the following cycle. Thus, one obtains

$$R_{i+1} = R_i + n/2 = (R_{i-1} - n) + n/2 = R_{i-1} - n/2,$$

which would be the same value. The steps of the nonrestoring algorithm, which implements this observation, are given in Algorithm 5.4.

Note that the nonrestoring division algorithm requires a final restoration cycle in which a negative remainder is corrected by adding the last value of $n$ back to it.

**Algorithm 5.4** The Nonrestoring Division Algorithm

```
Require: $P', n$.
Ensure: $R = P' \bmod n$.
 1: $R_0 := t$;
 2: $n := 2^k n$;
 3: for $i = 1$ to $k$ do
 4:    if $R_{i-1} > 0$ then
 5:       $R_i := R_{i-1} - n$;
 6:    else
 7:       $R_i := R_{i-1} + n$;
 8:    end if
 9:    $n := n/2$;
10:    if $R_k < 0$ then
11:       $R := R + n$;
12:    end if
13: end for
14: Return($R_k$)
```

In the following we compute $51 = 3019 \bmod 53$ using the nonrestoring division algorithm. Since the remainder is allowed to stay negative, we use 2's complement coding to represent such numbers.

| | | |
|---|---|---|
| $R_0$ | 0101111 001011 | $t$ |
| $n$ | 0110101 | subtract |
| $R_1$ | 1111010 | negative remainder |
| $n/2$ | 011010 1 | add |
| $R_2$ | 010100 1 | positive remainder |
| $n/2$ | 01101 01 | subtract |
| $R_3$ | 00111 01 | positive remainder |
| $n/2$ | 0110 101 | subtract |
| $R_4$ | 0000 110 | positive remainder |
| $n/2$ | 011 0101 | |
| $n/2$ | 01 10101 | |
| $n/2$ | 0 110101 | subtract |
| $R_5$ | 1 111110 | negative remainder |
| $n$ | 0 110101 | add (final restore) |
| $R$ | 0 110011 | Final remainder |

### 5.3.4 Interleaving Multiplication and Reduction

The interleaving algorithm has been known. The details of the method are sketched in papers [27, 334]. 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

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

This formulation yields the shift-add multiplication algorithm. Notice that we also reduce the partial product modulo $n$ at each step of Algorithm 5.5.

**Algorithm 5.5** The Interleaving Multiplication Algorithm

```
Require: $A, B, n$.
Ensure: $P = A \cdot B \bmod n$.
1: $P := 0$;
2: for $i = 0$ to $k - 1$ do
3:     $P := 2P + A \cdot B_{k-1-i}$;
4:     $P := P \bmod n$;
5: end for
6: 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:

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

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:

3a. $P := 2P$
3b. If carry-out then $P := P + m$
3c. $P := P + A \cdot B_j$
3d. 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 $n$.

### 5.3.5 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 in Algorithm 5.6.

**Algorithm 5.6** The Carry-Save Interleaving Multiplication Algorithm

```
Require: $A, B, n$.
Ensure: $P = A \cdot B \bmod n$.
1: $(C, S) := (0, 0)$;
2: for $i = 0$ to $k - 1$ do
3:    $(C, S) := 2C + 2S + A \cdot B_{k-1-i}$;
4:    $(C', S') := C + Sn$;
5:    if SIGN $\geq 0$ then
6:       $(C, S) := (C', S')$;
7:    end if
8: end for
9: 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 [185]. Here, we briefly review this algorithm, which is based on the addition of 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

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

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

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

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

**I.** $(\hat{C}, \hat{S}) := C + S - n.$
**J.** If $T(\hat{C}) + T(\hat{S}) \ge 0$ 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

$$0 \le C_0 + S_0 < (Q+1)n + 2^t$$

to the range

$$0 \le C_R + S_R < n + 2^t,$$

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.,

$$C_R + S_R \ge 0.$$

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 Algorithm 5.7.

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 7 of Algorithm 5.7, 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 4 of Algorithm 5.7, the range will be

**Algorithm 5.7** The Carry-Save Interleaving Multiplication Algorithm Revisited

```
Require: $A, B, n$.
Ensure: $P = A \cdot B \bmod n$.
1: Set $S^{(0)} := 0$ and $C^{(0)} := 0$.
2: for $i = 1$ to $k$ do
3:    $(C^{(i)}, S^{(i)}) := 2C^{(i-1)} + 2S^{(i-1)} + A_{n-i}B$;
4:    $(\hat{C}^{(i)}, \hat{S}^{(i)}) := C^{(i)} + S^{(i)} - 2n$;
5:    if $T(\hat{C}^{(i)}) + T(\hat{S}^{(i)}) \geq 0$ then
6:       $C^{(i)} := \hat{C}^{(i)}$ and $S^{(i)} := \hat{S}^{(i)}$.
7:    end if
8:    $(\hat{C}^{(i)}, \hat{S}^{(i)}) := C^{(i)} + S^{(i)} - n$;
9:    if $T(\hat{C}^{(i)}) + T(\hat{S}^{(i)}) \geq 0$ then
10:      $C^{(i)} := \hat{C}^{(i)}$ and $S^{(i)} := \hat{S}^{(i)}$;
11:   end if
12: end for
13: Return$(C^{(i)}, S^{(i)})$
```

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

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 $0 \leq \hat{P} = P - n < 2^t < n$ 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 |

### 5.3.6 Brickell's Method

This method is based on the use of a carry delayed integer introduced in §5.1.6. 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:

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

We can also reverse these steps to obtain:

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

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 [33] 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:

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

### 5.3.7 Montgomery's Method

In 1985, P. L. Montgomery introduced an efficient algorithm [238] for computing $R = A \cdot B \bmod n$ where $A$, $B$, and $n$ are $k$-bit binary numbers. The Montgomery reduction algorithm computes the resulting $k$-bit number $R$ without performing a division by the modulus $n$. Via an ingenious representation of the residue class modulo $n$, this algorithm replaces division by $n$ operation with division by a power of 2. This operation is easily accomplished on a computer since the numbers are represented in binary form. Assuming the modulus $n$ is a $k$-bit number, i.e., $2^{k-1} \leq n < 2^k$, let $r$ be $2^k$. The Montgomery reduction algorithm requires that $r$ and $n$ be relatively prime, i.e., $\gcd(r, n) = \gcd(2^k, n) = 1$. This requirement is satisfied if $n$ is odd. In the following we summarize the basic idea behind the Montgomery reduction algorithm.

Given an integer $A < n$, we define its $n$-residue with respect to $r$ as

$$\bar{A} = A \cdot r \bmod n.$$

It is straightforward to show that the set

$$\{\ i \cdot r \bmod n \mid 0 \leq i \leq n-1\ \}$$

is a complete residue system, i.e., it contains all numbers between 0 and $n-1$. Thus, there is a one-to-one correspondence between the numbers in the range 0 and $n-1$ and the numbers in the above set. The Montgomery reduction algorithm exploits this property by introducing a much faster multiplication routine which computes the $n$-residue of the product of the two integers whose $n$-residues are given. Given two $n$-residues $\bar{A}$ and $\bar{B}$, the *Montgomery product* is defined as the $n$-residue

$$\bar{R} = \bar{A} \cdot \bar{B} \cdot r^{-1} \bmod n$$

where $r^{-1}$ is the inverse of $r$ modulo $n$, i.e., it is the number with the property

$$r^{-1} \cdot r = 1 \bmod n.$$

The resulting number $\bar{R}$ is indeed the $n$-residue of the product

$$R = A \cdot B \bmod n$$

since

$$\begin{aligned}\bar{R} &= \bar{A} \cdot \bar{B} \cdot r^{-1} \bmod n \\ &= A \cdot r \cdot B \cdot r \cdot r^{-1} \bmod n \\ &= A \cdot B \cdot r \bmod n.\end{aligned}$$

In order to describe the Montgomery reduction algorithm, we need an additional quantity, $n'$, which is the integer with the property

$$r \cdot r^{-1} - n \cdot n' = 1.$$

The integers $r^{-1}$ and $n'$ can both be computed by the extended Euclidean algorithm [178]. The Montgomery product algorithm, which computes

$$u = \bar{A} \cdot \bar{B} \cdot r^{-1} \pmod{n}$$

given $\bar{A}$ and $\bar{B}$, is given in Algorithm 5.8 below.

**Algorithm 5.8** Montgomery Product

```
Require: $\bar{A}, \bar{B}, r, n$.
Ensure: $u$=MonPro($\bar{A}, \bar{B}$)=$\bar{A} \cdot \bar{B} \cdot r^{-1} \pmod{n}$.
1: $t := \bar{A} \cdot \bar{B}$;
2: $m := t \cdot n' \bmod r$;
3: $u := (t + m \cdot n)/r$;
4: if $u \geq n$ then
5:     Return($u - n$)
6: else
7:     Return($u$)
8: end if
```

The most important feature of the Montgomery product algorithm is that the operations involved are multiplications modulo $r$ and divisions by $r$, both of which are intrinsically fast operations since $r$ is a power 2. The MonPro Algorithm 5.9 can be used to compute the product of $A$ and $B$ modulo $n$, provided that $n$ is odd.

**Algorithm 5.9** Montgomery Modular Multiplication: Version I

```
Require: $A, B$, an odd number $n$.
Ensure: $x = A \cdot B \pmod{n}$.
1: Compute $n'$ using the extended Euclidean algorithm.
2: $\bar{A} := A \cdot r \bmod n$;
3: $\bar{B} := B \cdot r \bmod n$;
4: $\bar{x} :=$ MonPro($\bar{A}, \bar{B}$);
5: $x :=$ MonPro($\bar{x}, 1$);
6: Return($x$)
```

A better algorithm can be given by observing the property

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

which modifies Algorithm 5.9 as shown in Algorithm 5.10. However, the preprocessing operations, especially the computation of $n'$, are rather time-consuming.

**Algorithm 5.10** Montgomery Modular Multiplication: Version II

```
Require: $A, B$, an odd number $n$.
Ensure: $x = A \cdot B \pmod{n}$.
1: Compute $n'$ using the extended Euclidean algorithm.
2: $\bar{A} := A \cdot r \bmod n$;
3: $x :=$ MonPro$(\bar{A}, B)$;
4: Return($x$)
```

Nevertheless, there is an efficient algorithm for computing the single precision integer $n'_0$. The computation of $n'_0$ can be performed by a specialized Euclidean algorithm instead of the general extended Euclidean algorithm. Since $r = 2^{sw}$ and

$$r \cdot r^{-1} - n \cdot n' = 1,$$

we take modulo $2^w$ of the both sides, and obtain

$$-n \cdot n' = 1 \pmod{2^w},$$

or, in other words,

$$n'_0 = -n_0^{-1} \pmod{2^w},$$

where $n'_0$ and $n_0^{-1}$ are the least significant words (the least significant $w$ bits) of $n'$ and $n^{-1}$, respectively. In order to compute $-n_0^{-1} \pmod{2^w}$, we use algorithm 5.11 given below which computes $x^{-1} \pmod{2^w}$ for a given odd $x$.

**Algorithm 5.11** Specialized Modular Inverse

```
Require: an odd number $x$ and $w$.
Ensure: $y_w = x^{-1} \pmod{2^w}$.
1: $y_1 := 1$;
2: for $i = 2$ to $w$ do
3:     if $2^{i-1} < x \cdot y_{i-1} \pmod{2^i}$ then
4:         $y_i := y_{i-1} + 2^{i-1}$;
5:     else
6:         $y_i := y_{i-1}$;
7:     end if
8: end for
9: Return($y_w$)
```

The correctness of the algorithm follows from the observation that, at every step $i$, we have

$$x \cdot y_i = 1 \pmod{2^i}.$$

This algorithm is very efficient, and uses single precision addition and multiplications in order to compute $x^{-1}$. As an example, we compute $23^{-1} \pmod{64}$ using the above algorithm. Here we have $x = 23$, $w = 6$. The steps of the algorithm, the temporary values, and the final inverse are shown below:

| $i$ | $2^i$ | $y_{i-1}$ | $x \cdot y_{i-1} \pmod{2^i}$ | $2^{i-1}$ | $y_i$ |
|---|---|---|---|---|---|
| 2 | 4 | 1 | $23 \cdot 1 = 3$ | 2 | $1 + 2 = 3$ |
| 3 | 8 | 3 | $23 \cdot 3 = 5$ | 4 | $3 + 4 = 7$ |
| 4 | 16 | 7 | $23 \cdot 7 = 1$ | 8 | 7 |
| 5 | 32 | 7 | $23 \cdot 7 = 1$ | 16 | 7 |
| 6 | 64 | 7 | $23 \cdot 7 = 33$ | 32 | $7 + 32 = 39$ |

Thus, we compute $23^{-1} = 39 \pmod{64}$. This is indeed the correct value since

$$23 \cdot 39 = 14 \cdot 64 + 1 = 1 \pmod{64}.$$

Also, at every step $i$, we have $x \cdot y_i = 1 \pmod{2^i}$, as shown below:

| $i$ | $x \cdot y_i \bmod 2^i$ |
|---|---|
| 1 | $23 \cdot 1 = 1 \bmod 2$ |
| 2 | $23 \cdot 3 = 1 \bmod 4$ |
| 3 | $23 \cdot 7 = 1 \bmod 8$ |
| 4 | $23 \cdot 7 = 1 \bmod 16$ |
| 5 | $23 \cdot 7 = 1 \bmod 32$ |
| 6 | $23 \cdot 39 = 1 \bmod 64$ |

## Montgomery Exponentiation

The Montgomery product algorithm is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute a modular exponentiation, i.e., the computation of $M^e \bmod n$. Using one of the addition chain algorithms given in §5.4, we replace the exponentiation operation by a series of square and multiplication operations modulo $n$. This is where the Montgomery product operation finds its best use. In the following we summarize the modular exponentiation operation which makes use of the Montgomery product function MonPro. The exponentiation Algorithm 5.12 below uses the binary method.

Thus, we start with the ordinary residue $M$ and obtain its $n$-residue $\bar{M}$ using a division-like operation, which can be achieved, for example, by a series of shift and subtract operations. Additionally, Steps 2 and 3 of Algorithm 5.12 require divisions. However, once the preprocessing has been completed, the inner-loop of the binary exponentiation method uses the Montgomery product operations which performs only multiplications modulo $2^k$ and divisions by $2^k$. When the binary method finishes, we obtain the $n$-residue $\bar{x}$ of the quantity $x = M^e \bmod n$. The ordinary residue number is obtained from the $n$-residue by executing the MonPro function with arguments $\bar{x}$ and 1. This is easily shown to be correct since

$$\bar{x} = x \cdot r \bmod n$$

immediately implies that

$$x = \bar{x} \cdot r^{-1} \bmod n \ = \ \bar{x} \cdot 1 \cdot r^{-1} \bmod n \ := \ \text{MonPro}(\bar{x}, 1).$$

**Algorithm 5.12** Montgomery Modular Exponentiation

```
Require: $A, B$, and odd number $n$.
Ensure: $x = M^e \pmod{n}$.
1: Compute $n'$ using the extended Euclidean algorithm.
2: $\bar{M} := M \cdot r \bmod n$;
3: $\bar{x} := 1 \cdot r \bmod n$;
4: for $i = k - 1$ down to 0 do
5:       $\bar{x} := \text{MonPro}(\bar{x}, \bar{x})$;
6:    if $e_i = 1$ then
7:       $\bar{x} := \text{MonPro}(\bar{M}, \bar{x})$;
8:    end if
9: end for
10: $x := \text{MonPro}(\bar{x}, 1)$;
11: Return($x$)
```

The resulting algorithm is quite fast as was demonstrated by many researchers and engineers who have implemented it, for example, see [72, 200]. However, this algorithm can be refined and made more efficient, particularly when the numbers involved are multi-precision integers. For example, Dussé and Kaliski [72] gave improved algorithms, including a simple and efficient method for computing $n'$. We will describe these methods below.

## An Example of Exponentiation

Here we show how to compute $x = 7^{10} \bmod 13$ using the Montgomery exponentiation algorithm.

- Since $n = 13$, we take $r = 2^4 = 16 > n$.
- Computation of $n'$:
  Using the extended Euclidean algorithm, we determine that $16 \cdot 9 - 13 \cdot 11 = 1$, thus, $r^{-1} = 9$ and $n' = 11$.
- Computation of $\bar{M}$:
  Since $M = 7$, we have $\bar{M} := M \cdot r \pmod{n} = 7 \cdot 16 \pmod{13} = 8$.
- Computation of $\bar{x}$ for $x = 1$:
  We have $\bar{x} := x \cdot r \pmod{n} = 1 \cdot 16 \pmod{13} = 3$.
- Steps 5 and 7 of the ModExp routine:

| $e_i$ | Step 5 | Step 7 |
|---|---|---|
| 1 | MonPro$(3,3) = 3$ | MonPro$(8,3) = 8$ |
| 0 | MonPro$(8,8) = 4$ | |
| 1 | MonPro$(4,4) = 1$ | MonPro$(8,1) = 7$ |
| 0 | MonPro$(7,7) = 12$ | |

  ∘ Computation of MonPro$(3,3) = 3$:
  $t := 3 \cdot 3 = 9$
  $m := 9 \cdot 11 \pmod{16} = 3$
  $u := (9 + 3 \cdot 13)/16 = 48/16 = 3$

  ∘ Computation of MonPro$(8,3) = 8$:
  $t := 8 \cdot 3 = 24$
  $m := 24 \cdot 11 \pmod{16} = 8$
  $u := (24 + 8 \cdot 13)/16 = 128/16 = 8$

- Computation of MonPro(8, 8) = 4:
  $t := 8 \cdot 8 = 64$
  $m := 64 \cdot 11 \pmod{16} = 0$
  $u := (64 + 0 \cdot 13)/16 = 64/16 = 4$
- Computation of MonPro(4, 4) = 1:
  $t := 4 \cdot 4 = 16$
  $m := 16 \cdot 11 \pmod{16} = 0$
  $u := (16 + 0 \cdot 13)/16 = 16/16 = 1$
- Computation of MonPro(8, 1) = 7:
  $t := 8 \cdot 1 = 8$
  $m := 8 \cdot 11 \pmod{16} = 8$
  $u := (8 + 8 \cdot 13)/16 = 112/16 = 7$
- Computation of MonPro(7, 7) = 12:
  $t := 7 \cdot 7 = 49$
  $m := 49 \cdot 11 \pmod{16} = 11$
  $u := (49 + 11 \cdot 13)/16 = 192/16 = 12$

- Step 7 of the ModExp routine: $x = \text{MonPro}(12, 1) = 4$
  $t := 12 \cdot 1 = 12$
  $m := 12 \cdot 11 \pmod{16} = 4$
  $u := (12 + 4 \cdot 13)/16 = 64/16 = 4$

Thus, we obtain $x = 4$ as the result of the operation $7^{10} \bmod 13$.

## Hardware Implementation of the Montgomery Method

In the rest of 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 $t = A \cdot B \cdot 2^{-k}$, as shown in Algorithm 5.13.

Procedure 5.13 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

**Algorithm 5.13** Add-and-Shift Montgomery Product

```
Require: $A, B$.
Ensure: $t = A \cdot B \cdot 2^{-k}$.
1: $t := 0$;
2: for $i = 0$ to $k - 1$ do
3:     $t := t + A_i \cdot B$;
4:     $t := t/2$;
5: end for
6: Return($t$)
```

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 shown in Algorithm 5.14.

**Algorithm 5.14** Binary Add-and-Shift Montgomery Product

```
Require: $A, B$, an odd number $n$.
Ensure: $u = A \cdot B \cdot 2^{-k} \pmod{n}$.
1: $u := 0$;
2: for $i = 0$ to $k - 1$ do
3:     $u := u + A_i \cdot B$;
4:     if $u$ is odd then
5:         $u := u + n$;
6:     end if
7:     $u := u/2$;
8: end for
9: Return($u$)
```

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.

### 5.3.8 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 $i$th 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 Fig. 5.8.

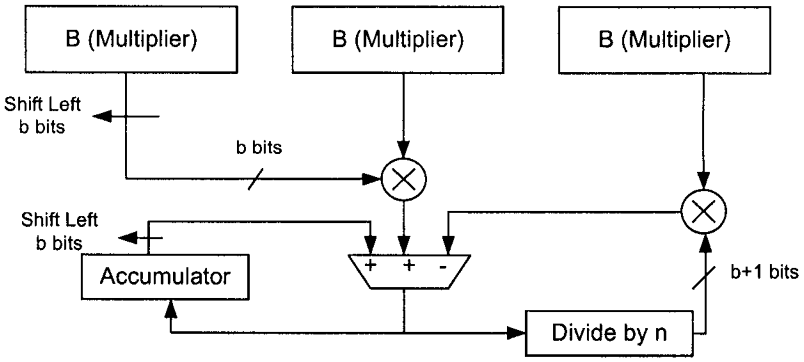

**Fig. 5.8.** High-Radix Interleaving Method

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.

### 5.3.9 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 in Algorithm 5.15.

**Algorithm 5.15** Word-Level Add-and-Shift Montgomery Product

```
Require: $A, B$, an odd number $n$, $k = sw$.
Ensure: $u = A \cdot B \cdot 2^{-k} \pmod{n}$.
1: $u := 0$;
2: for $i = 0$ to $s - 1$ do
3:     $u := u + A_i \cdot B$;
4:     $m := -u_0 \cdot n_0^{-1} \bmod 2^w$;
5:     $u := u + m \cdot n$;
6:     $u := u/2^w$;
7: end for
8: Return($u$)
```

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

## 5.4 Modular Exponentiation Operation

Modular exponentiation can be defined in terms of field multiplication as follows. Let $x$ be a positive integer in $[1, n]$. Let also $e$ be defined as an arbitrary positive integer. Then, we define modular exponentiation as the problem of finding the number $y$ such that,

$$y = x^e \bmod n \qquad (5.1)$$

Taking advantage of the linearity property of the modular operation, (5.1) can be evaluated by performing a reduction modulo $n$ at each step of the exponentiation thus guaranteeing that all the partial results will not grow larger than twice the length of the modulus. In the rest of this Section we will consider that every multiplication operation always includes a subsequent reduction step.

In general one can follow two strategies in order to optimize the computation of (5.1). One approach is to implement field multiplication, the main building block required for field exponentiation, as efficiently as possible. The other is to reduce the total number of multiplications needed to compute (5.1). In this Section we address the latter approach, assuming that arbitrary choices of the base $x$ are allowed but considering that the exponent $e$ has been previously fixed.

In this section, we include a brief review of the main deterministic heuristic proposed in the literature for computing field exponentiation.

## 5.4.1 Binary Strategies

Let $e$ be an arbitrary $m$-bit positive integer $e$, with a binary expansion representation given as, $e = (1e_{m-2} \dots e_1 e_0)_2 = 2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i$. Then,

$$\mathbf{y} = \mathbf{x}^e = \mathbf{x}^{2^{m-1}+\sum_{i=0}^{m-2} 2^i e_i} = x^{2^{m-1}} \cdot \prod_{i=0}^{m-2} \mathbf{x}^{2^i e_i} \tag{5.2}$$

Binary strategies evaluate (5.2) by scanning the bits of the exponent $e$ one by one, either from left to right (MSB-first binary algorithm) or from right to left (LSB-first binary algorithm) applying the so-called Horner's rule [2]. Both strategies require a total of $m - 1$ iterations. At each iteration a squaring operation is performed, and if the value of the scanned bit is one, a subsequent field multiplication is performed. Therefore, the binary strategy requires a total of $m - 1$ squarings and $H(e) - 1$ field multiplications, where $H(e)$ is the Hamming weight of the binary representation of $e$. The pseudo-code of the MSB-first and the LSB-first binary algorithms are shown in Figures 5.16 and 5.17, respectively. The computational complexity of the algorithm in Figure 5.16 is given as,

$$P(e,m) = m + H(e) - 2 = \lfloor \log_2(e) \rfloor + H(e) - 1 \tag{5.3}$$

[2] Horner's rule, named after W. G. Horner, ranks among the most efficient algorithms for the computation of $n$th degree polynomials of the form, $p(x) = p_n x^n + p_{n-1} x^n - 1 + \cdots + p_1 x + u_0, p_n \neq 0$, for fixed values of $x$. Horner's rule solves this problem by evaluating $p(x)$ as, $p(x) = (\dots(p_n x + p_{n-1})x + \cdots)x + p_0$. This elegant algorithm was discovered independently by Isaac Newton 150 years earlier than Horner and by the Chinese mathematician C. C. Chao in the 13th century [178]

**An Example.** Let us define $e = 1903 = (11101101111)_2$. Then $m = 11$ and $H(e) = 9$. According to (5.3) the computational complexity of the binary algorithm is given as,

$P(e) = m + H(e) - 2 = 11 + 9 - 2 = 18.$

After evaluating the algorithm of Figure 5.16, the resulting binary sequence is given as,

$$x^1 \to x^2 \to x^3 \to x^6 \to x^7 \to x^{14} \to x^{28} \to x^{29} \to x^{58}$$
$$\to x^{59} \to x^{118} \to x^{236} \to x^{237} \to x^{474} \to x^{475} \to x^{950}$$
$$\to x^{951} \to x^{1902} \to x^{1903}.$$

We compare the MSB-first and the LSB-first binary algorithms in terms of time and space requirements below:

- Both methods require $m - 1$ squarings and an average of $\frac{1}{2}(m - 1)$ multiplications.
- The MSB-first binary method requires two registers: $x$ and $y$.
- The LSB-first binary method requires three registers: $x$, $y$, 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 4) and squaring (Step 5) operations in the LSB-first 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 LSB-first binary method is bounded by the total time required for computing $h-1$ squaring operations on $k$-bit integers.

**Algorithm 5.16** MSB-First Binary Exponentiation

```
Require: $\mathsf{x}, \mathsf{n}, e = (e_{m-1} \dots e_1 e_0)_2$.
Ensure: $\mathsf{y} = \mathsf{x}^e \bmod n$.
1: $\mathsf{y} = x$;
2: for $i = m - 2$ downto 0 do
3:     $\mathsf{y} = \mathsf{y}^2$ ;
4:     if $e_i == 1$ then
5:         $\mathsf{y} = \mathsf{y} \cdot \mathsf{x}$;
6:     end if
7: end for
8: Return(y)
```

### 5.4.2 Window Strategies

The binary method discussed in the preceding section can be generalized by scanning more than one bit at a time. Hence, the window method (first

**Algorithm 5.17** LSB-First Binary Exponentiation

```
Require: $x, n, e = (e_{m-1} \dots e_1 e_0)_2$.
Ensure: $y = x^e \bmod n$.
1: $p = x$ ; $y = 1$;
2: for $i = 0$ to $m - 1$ do
3:    if $e_i == 1$ then
4:       $y = y \cdot p$;
5:    end if
6:    $p = p^2$;
7: end for
8: Return(y)
```

described in [178]) scans $k$ bits at a time. The window method is based on a $k$-ary expansion of the exponent, where the bits of the exponent $e$ are divided into $k$-bit words or digits. The resulting words of $e$ are then scanned performing $k$ consecutive squarings and a subsequent multiplication as needed. In the following we describe the window method in a more formal way.

**Algorithm 5.18** MSB-First $2^k$-ary Exponentiation

```
Require: $x, n, e = (e_{m-1} \dots e_1 e_0)_2$, k divisor of $m$ such that $\Psi = m/k$.
Ensure: $y = x^e \bmod n$.
1: Pre-compute and store $x^j$ for all $j = 1, 2, 3, 4, \dots, 2^k - 1$.
2: Divide $e$ into $k$-bit words $W_i$ for $i = 0, 1, 2, \dots, \Psi - 1$.
3: $y = x^{W_{\Psi-1}}$;
4: for $i = \Psi - 2$ downto 0 do
5:    $y = y^{2^k}$;
6:    if $W_i \neq 0$ then
7:       $y = y \cdot x^{W_i}$;
8:    end if
9: end for
10: Return(y)
```

Let $e$ be an arbitrary $m$-bit positive integer $e$, with a binary expansion representation given as,

$$e = (1e_{m-2} \dots e_1 e_0)_2 = 2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i.$$

Let $k$ be a small divisor of $m$. Then this binary expansion of $e$ can be partitioned into $\Psi$ words of length $k$, such that $k\Psi = m$. If $k$ does not divide $m$, then the exponent must be padded with at most $k - 1$ zeros. Let us define $W_i \in [\![0, 2^k - 1]\!]$ as,

$$W_i = \left(e_{ik+(k-1)}e_{ik+(k-2)}\cdots e_{ik+1}e_{ik}\right)_2 = \sum_{j=0}^{k-1} 2^j e_{(ik+j)} \tag{5.4}$$

Then, we can equivalently represent $e$ as, $\sum_{i=0}^{\Psi-1} W_i \cdot 2^{id}$. Using the above definition we have,

$$\mathbf{y} = \mathbf{x}^e = \mathbf{x}^{\sum_{i=0}^{\Psi-1} 2^{id} W_i} = \prod_{i=0}^{\Psi-1} \mathbf{x}^{2^{id} W_i} \tag{5.5}$$

(5.5) is the basis of the window MSB-first procedure for exponentiation described in the pseudo-code of Figure 5.18. The window method first pre-computes the values of $x^j$ for $j = 1, 2, 3, \ldots, 2^k - 1$. Then, the exponent $e$ is scanned $k$ bits at a time from the most significant word ($W_{\Psi-1}$) to the least significant word ($W_0$). At each iteration the current partial result $y$ is raised to the $2^k$ power and multiplied with $x^{W_i}$, where $W_i$ is the current nonzero word being processed. Referring to Figure 5.18, it can be seen that,

- The first part of the algorithm consists on the pre-computation of the first $2^k$ powers of $\mathbf{x}$ at a cost of $2^k - 2$ preprocessing multiplications.
- At each iteration of the main loop, the power $\mathbf{y}^{2^k}$ can be computed by performing $k$ consecutive squarings. The total number of squarings is given by $(\Psi - 1)k = m - k$.
- At each iteration one multiplication is performed whenever the $i$-th word $W_i$ is different than zero. Since all but one of the $2^k$ different values of $W_i$ are nonzero, the average number of required multiplications is given as, $(\Psi - 1)(1 - 2^{-k}) = (\frac{m}{k} - 1)(1 - 2^{-k})$.

Thus, the average number of multiplications needed by the window method in order to compute an $m$-bit field exponentiation is given as,

$$P(m, k) = (2^k - 2) + (m - k) + (\frac{m}{k} - 1)(1 - 2^{-k}). \tag{5.6}$$

For $k = 1, 2, 3, 4$ the window method sketched at Figure 5.18 is called, respectively, *binary*, *quaternary*, *octary* and *hexa* MSB-first exponentiation method. In particular, note that by evaluating (5.6) for $k = 1$, the average number of multiplications for the binary algorithm can be found as $\frac{3}{2}(m - 1)$ field operations on average.

One obvious improvement of the strategy just outlined is that instead of calculating and storing all the $2^k$ first powers of $x$, one can just pre-compute the windows needed for a given exponent $e$, thus saving some operations. This last idea is illustrated in the examples below.

**Example.** Once again, let us consider the exponent $e = 1903 = (11101101111)_2$ with $m = 11$. Then, the window method computational complexity and resulting sequence using $k = 2, 3, 4$ can be found as,
**Quaternary**: $e = 1903 = (\mathbf{01\,11\,01\,10\,11\,11})_2$

$P(m,k) =$ 2 Pre-comp mults + 10 Sqrs + 5 mults $= 17$.
Precomp. Sequence: $x^1 \rightarrow x^2 \rightarrow x^3$.
Main sequence:

$$x^1 \rightarrow x^2 \rightarrow x^4 \rightarrow x^7 \rightarrow x^{14} \rightarrow x^{28} \rightarrow x^{29} \rightarrow x^{58} \rightarrow x^{116} \rightarrow x^{118} \rightarrow x^{236} \rightarrow x^{472} \rightarrow x^{475} \rightarrow x^{950} \rightarrow x^{1900} \rightarrow x^{1903}.$$

**Octal**: $e = 1903 = (011\,101\,101\,111)_2$
$P(m,k) =$ 4 Pre-comp mults + 9 Sqrs + 3 mults $= 16$.
Precomp. Sequence: $x^1 \rightarrow x^2 \rightarrow x^3 \rightarrow x^5 \rightarrow x^7$.
Main sequence:

$$x^3 \rightarrow x^6 \rightarrow x^{12} \rightarrow x^{24} \rightarrow x^{29} \rightarrow x^{58} \rightarrow x^{116} \rightarrow x^{232} \rightarrow x^{237} \rightarrow x^{474} \rightarrow x^{948} \rightarrow x^{1896} \rightarrow x^{1903}$$

**Hexa**: $e = 1903 = (0111\,0110\,1111)_2$
$P(m,k) =$ 6 Pre-comp mults + 8 Sqrs + 2 mults $= 16$.
Precomp. Sequence: $x^1 \rightarrow x^2 \rightarrow x^3 \rightarrow x^6 \rightarrow x^7 \rightarrow x^{14} \rightarrow x^{15}$.
Main sequence:

$$x^7 \rightarrow x^{14} \rightarrow x^{28} \rightarrow x^{56} \rightarrow x^{112} \rightarrow x^{118} \rightarrow x^{236} \rightarrow x^{472} \rightarrow x^{944} \rightarrow x^{1888} \rightarrow x^{1903}.$$

However, none of the above deterministic methods is able to find the shortest addition chain[3] for $e = 1903$.

### 5.4.3 Adaptive Window Strategy

The adaptive or sliding window strategy is quite useful for exponentiations with extremely large exponents (i.e. exponents with bit length greater than 128 bits) mainly because of its ability to adjust its method of computation according to the specific form of the exponent at hand. This adjustment is done by partitioning the input exponent into a series of variable-length zero and nonzero words called *windows*. As opposed to the traditional window method discussed in the previous section, the sliding window algorithm provides a performance tradeoff in the sense that allows the processing of variable-length zero and nonzero digits. The main goal pursued by this strategy is to try to maximize the number and length of zero words, while using relatively large values of $k$.

A sliding window exponentiation algorithm is typically divided into two phases: exponent partitioning and the field exponentiation computation itself.

[3] Addition chains are formally defined in §6.3.3.

In the first phase, the exponent $e$ is decomposed into zero and nonzero words (*windows*) $W_i$ of length $L(W_i)$ by using some partitioning strategy. Although in general it is not required that the window's lengths $L(W_i)$ must all be equal, all nonzero windows should have a length $L(W_i)$ smaller than a given number $k$. Let $Z$ be the number of zero windows and $NZ$ be the number of non-zero windows, so that their addition $\Psi$ represents the total number of windows generated by the partitioning phase, i.e.,

$$\Psi = Z + NZ \tag{5.7}$$

It is useful to force the least significant bit of a nonzero window $W_i$ to be equal to 1. In this way, when comparing with the standard window method discussed in the previous Section, the number of preprocessing multiplications are at least nearly halved, since $x^w$ must only be pre-computed for $w$ odd.

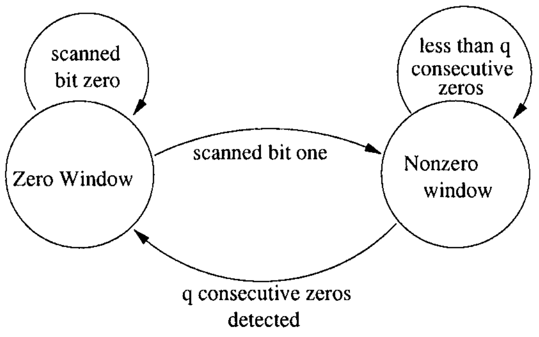

**Fig. 5.9.** Partitioning Algoritm

Several sliding window partitioning approaches have been proposed [116, 178, 191, 181, 30, 35]. Proposed techniques differ in whether the length of a nonzero window has to have a constant or a variable length. The partitioning algorithm instrumented in this work scans the exponent from the most significant to the least significant bit according to the finite state machine shown in Figure 5.9. Hence, at any moment the algorithm is either completing a zero window or a nonzero window. Zero windows are allowed to have an arbitrary length. However, the maximum length of any given nonzero window should not exceed the value of $k$ bits.

Starting from the Zero Window State (ZWS), the exponent bits are checked one by one. As long as the value of the current scanned bit is zero, the algorithm stays in ZWS accumulating as many consecutive zeros as possible. If the incoming bit is one, the finite state machine switches to the Nonzero Window State (NZWS). The automaton will stay there as long as $q$ consecutive zeros had not been collected. If this condition occurs the automaton switches to ZWS (usually $q$ is chosen to be a small number, namely, $q \in [2, 5]$).

Otherwise, if $k$ bits can been collected, the partitioning algorithm stores the new formed nonzero window and stays in NZWS in order to generate another nonzero window.

**Algorithm 5.19** Sliding Window Exponentiation

```
Require: $\mathbf{x}, \mathbf{n}, e = (e_{m-1} \dots e_1 e_0)_2$.
Ensure: $\mathbf{y} = \mathbf{x}^e \bmod n$.
 1: Pre-compute and store $x^j$ for at most all $j = 1, 2, 3, 4, \dots, 2^k - 1$.
 2: Divide $e$ into zero and nonzero windows $W_i$ of length $L(W_i)$ for
    $i = 0, 1, 2, \dots, \Psi - 1$.
 3: $\mathbf{y} = x^{W_{\Psi-1}}$;
 4: for $i = \Psi - 2$ downto 0 do
 5:   $\mathbf{y} = \mathbf{y}^{2^{L(W_i)}}$;
 6:   if $W_i \neq 0$ then
 7:     $\mathbf{y} = \mathbf{y} \cdot \mathbf{x}^{W_i}$;
 8:   end if
 9: end for
10: Return(y)
```

The pseudo-code for the sliding window exponentiation algorithm is shown in Figure 5.19. From that figure it can be seen that,

- The first part of the algorithm consists on the pre-computation of at most the first $2^k$ odd powers of $\mathbf{x}$ at a cost of no more than $2^{k-1} - 1$ preprocessing multiplications.
- At step 2, the exponent $e$ is partitioned using the strategy described above and depicted in Figure 5.9. As a consequence, a total of $Z$ zero windows and $NZ$ nonzero windows will be produced.
- At step 3, $\mathbf{y}$ is initialized using the value of the Most Significant Window as $\mathbf{y} = x^{W_{\Psi-1}}$. It is always assumed that $W_{\Psi-1} \neq 0$.
- At each iteration of the main loop, the power $\mathbf{y}^{2^{L(W_i)}}$ can be computed by performing $L(W_i)$ consecutive squarings. The total number of squarings is given by $m - L(W_{\Psi-1})$
- At each iteration one multiplication is performed whenever the $i$-th word $W_i$ is different than zero. Recall that $NZ$ represents the number of nonzero windows. Therefore, the number of multiplications required at this step of this algorithm is $NZ - 1$. Although the exact value of $NZ$ will depend on the partitioning strategy instrumented, our experiments show that an approximate value for $NZ$ using $q = 2, k = 5$, is about $0.15m$.

Thus, we find that the average number of multiplications needed to compute a field exponentiation for an $m$-bit exponent $e$ is given as,

$$\begin{aligned} P(m,k) &= (2^{k-1} - 1) + (m - L(W_{k-1})) + NZ - 1 \qquad (5.8) \\ &\approx 2^{k-1} - 1 + 1.15m - L(W_{k-1}). \end{aligned}$$

Due to the considerable high efficiency of the partitioning strategy for collecting zero words, the sliding window method significantly outperforms the standard window method when sufficiently large exponents are computed [181]. However, notice that the value of the parameter $k$ cannot be chosen too large due to the exponentially increasing cost of pre-computing the first $2^k$ odd powers of $\mathbf{x}$ (step 1 of Figure 5.19). In practice and depending on the value of $m$, $k \in [4, 8]$ is generally adopted.

After executing the above algorithm, it is found that the modular exponentiation operation $M^e \bmod n$ with $e = 1903$, can be computed by performing 9 field squarings and 6 field multiplications, according with the sequence shown below,

$$\begin{aligned} x^1 \rightarrow x^2 \rightarrow x^3 \rightarrow x^6 \rightarrow x^{12} \rightarrow x^{24} \rightarrow x^{25} \rightarrow x^{50} \\ \rightarrow x^{100} \rightarrow x^{200} \rightarrow x^{300} \rightarrow x^{600} \rightarrow x^{900} \rightarrow x^{1800} \\ \rightarrow x^{1900} \rightarrow x^{1903}. \end{aligned} \tag{5.9}$$

Each of the deterministic heuristics just described clearly sets an upper bound on the number of field operations required for computing the modular exponentiation operation. In particular, the theoretical cost of the binary algorithm given in (5.3) implies that $l(e) \leq m + H(e) - 1$. A lower bound for $l(e)$ was found in [321] as, $\log_2 e + \log_2 H(e) - 2.13$. Therefore we can write,

$$\log_2 e + \log_2 H(e) - 2.13 \leq l(e) \leq \lfloor log_2(e) \rfloor + H(e) - 1 \tag{5.10}$$

Let us suppose that we are interested in computing the modular exponentiation for several exponents of a given fixed bit-length, say, $m$. Then, as it was shown in [191], the minimum number of underlying field operations is a function of the Hamming weight $H(e)$. Indeed, one can expect that on average $l(e)$ will be smaller for both, $H(e)$ closer to 0 and for $H(e)$ closer to $m$. On the contrary, when $H(e)$ is close to $m/2$, i.e., for those $m$-bit exponents having a balanced number of zeros and ones, $l(e)$ happens to be maximal [191].

### 5.4.4 RSA Exponentiation and the Chinese Remainder Theorem

Let us recall from Chapter 2 that 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 how the RSA modular exponentiation can be greatly benefit by applying the Chinese Remainder Theorem to it.

## The Chinese Remainder Theorem

The Chinese Remainder Theorem(CRT) has a tremendous importance in cryptography. For instance, Quisquater and Couvreur proposed in [279] to use it for speeding up the RSA decryption primitive. It can be defined as follows.

Let $p_i$ for $i = 1, 2, \ldots, k$ be pairwise relatively prime integers, i.e.,

$$gcd(p_i, p_j) = 1 \text{ for } i \neq j.$$

Given $u_i \in [0, p_i - 1]$ for $i = 1, 2, \ldots, k$, the Chinese remainder theorem states that there exists a unique integer $u$ in the range $[0, P-1]$ where $P = p_1 p_2 \cdots p_k$ such that

$$u = u_i \pmod{p_i}.$$

In the case of RSA decryption primitive, The Chinese remainder theorem tells us that the computation of

$$M := C^d \pmod{p \cdot q},$$

can be broken into two parts as

$$\begin{aligned} M_1 &:= C^d \pmod{p}, \\ M_2 &:= C^d \pmod{q}, \end{aligned}$$

after which the final value of $M$ is computed (lifted) by the application of a Chinese remainder algorithm. There are two algorithms for this computation: The single-radix conversion (SRC) algorithm and the mixed-radix conversion (MRC) algorithm. Here, we briefly describe these algorithms, details of which can be found in [105, 355, 178, 209]. Going back to the general example, we observe that the SRC or the MRC algorithm computes $u$ given $u_1, u_2, \ldots, u_k$ and $p_1, p_2, \ldots, p_k$. The SRC algorithm computes $u$ using the summation

$$u = \sum_{i=1}^{k} u_i c_i P_i \pmod{P},$$

where

$$P_i = p_1 p_2 \cdots p_{i-1} p_{i+1} \cdots p_k = \frac{P}{p_i},$$

and $c_i$ is the multiplicative inverse of $P_i$ modulo $p_i$, i.e.,

$$c_i P_i = 1 \pmod{p_i}.$$

Thus, applying the SRC algorithm to the RSA decryption, we first compute

$$\begin{aligned} M_1 &:= C^d \pmod{p}, \\ M_2 &:= C^d \pmod{q}, \end{aligned}$$

However, applying Fermat's theorem to the exponents, we only need to compute

$$\begin{aligned} M_1 &:= C^{d_1} \pmod{p}, \\ M_2 &:= C^{d_2} \pmod{q}, \end{aligned}$$

where

$$\begin{aligned} d_1 &:= d \bmod (p-1), \\ d_2 &:= d \bmod (q-1). \end{aligned}$$

This provides some savings since $d_1, d_2 < d$; in fact, the sizes of $d_1$ and $d_2$ are about half of the size of $d$. Proceeding with the SRC algorithm, we compute $M$ using the sum

$$M = M_1 c_1 \frac{pq}{p} + M_2 c_2 \frac{pq}{q} \pmod{n} \;=\; M_1 c_1 q + M_2 c_2 p \pmod{n},$$

where $c_1 = q^{-1} \pmod{p}$ and $c_2 = p^{-1} \pmod{q}$. This gives

$$M = M_1 (q^{-1} \bmod p) q + M_2 (p^{-1} \bmod q) p \pmod{n}.$$

In order to prove this, we simply show that

$$\begin{aligned} M \pmod{p} &= M_1 \cdot 1 + 0 = M_1, \\ M \pmod{q} &= 0 + M_2 \cdot 1 = M_2. \end{aligned}$$

The MRC algorithm, on the other hand, computes the final number $u$ by first computing a triangular table of values:

$$\begin{array}{lllll} u_{11} & & & & \\ u_{21} & u_{22} & & & \\ u_{31} & u_{32} & u_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ u_{k1} & u_{k2} & \cdots & \cdots & u_{k,k} \end{array}$$

where the first column of the values $u_{i1}$ are the given values of $u_i$, i.e., $u_{i1} = u_i$. The values in the remaining columns are computed sequentially using the values from the previous column according to the recursion

$$u_{i,j+1} = (u_{ij} - u_{jj}) c_{ji} \pmod{p_i},$$

where $c_{ji}$ is the multiplicative inverse of $p_j$ modulo $p_i$, i.e.,

$$c_{ji}p_j = 1 \pmod{p_i}.$$

For example, $u_{32}$ is computed as

$$u_{32} = (u_{31} - u_{11})c_{13} \pmod{p_3},$$

where $c_{13}$ is the inverse of $p_1$ modulo $p_3$. The final value of $u$ is computed using the summation

$$u = u_{11} + u_{22}p_1 + u_{33}p_1p_2 + \cdots + u_{kk}p_1p_2\cdots p_{k-1}$$

which does not require a final modulo $P$ reduction. Applying the MRC algorithm to the RSA decryption, we first compute

$$\begin{aligned} M_1 &:= C^{d_1} \pmod{p}, \\ M_2 &:= C^{d_2} \pmod{q}, \end{aligned}$$

where $d_1$ and $d_2$ are the same as before. The triangular table in this case is rather small, and consists of

$$\begin{array}{ll} M_{11} & \\ M_{21} & M_{22} \end{array}$$

where $M_{11} = M_1$, $M_{21} = M_2$, and

$$M_{22} = (M_{21} - M_{11})(p^{-1} \bmod q) \pmod{q}.$$

Therefore, $M$ is computed using

$$M := M_1 + [(M_2 - M_1) \cdot (p^{-1} \bmod q) \bmod q] \cdot p.$$

This expression is correct since

$$\begin{aligned} M \pmod{p} &= M_1 + 0 &= M_1, \\ M \pmod{q} &= M_1 + (M_2 - M_1) \cdot 1 &= M_2. \end{aligned}$$

The MRC algorithm is more advantageous than the SRC algorithm for two reasons:

- It requires a single inverse computation: $p^{-1} \bmod q$.
- It does not require the final modulo $n$ reduction.

The inverse value $(p^{-1} \bmod q)$ can be precomputed and saved. Here, we note that the order of $p$ and $q$ in the summation in the proposed public-key cryptography standard PKCS # 1 is the reverse of our notation. The data structure [194] holding the values of user's private key has the variables:

```
exponent1 INTEGER, -- d mod (p-1)
exponent2 INTEGER, -- d mod (q-1)
coefficient INTEGER, -- (inverse of q) mod p
```

Thus, it uses $(q^{-1} \bmod p)$ instead of $(p^{-1} \bmod q)$. Let $M_1$ and $M_2$ be defined as before. By reversing $p$, $q$ and $M_1$, $M_2$ in the summation, we obtain

$$M := M_2 + [(M_1 - M_2) \cdot (q^{-1} \bmod p) \bmod p] \cdot q.$$

This summation is also correct since

$$\begin{aligned} M \pmod q &= M_2 + 0 &= M_2, \\ M \pmod p &= M_2 + (M_1 - M_2) \cdot 1 &= M_1, \end{aligned}$$

as required. Assuming $p$ and $q$ are $(k/2)$-bit binary numbers, and $d$
is as large as $n$ which is a $k$-bit integer, we now calculate the total number of bit operations for the RSA decryption using the MRC algorithm. Assuming $d_1$, $d_2$, $(p^{-1} \bmod q)$ are precomputed, and that the exponentiation algorithm is the binary method, we calculate the required number of multiplications as

- Computation of $M_1$: $\frac{3}{2}(k/2)$ $(k/2)$-bit multiplications.
- Computation of $M_2$: $\frac{3}{2}(k/2)$ $(k/2)$-bit multiplications.
- Computation of $M$: One $(k/2)$-bit subtraction, two $(k/2)$-bit multiplications, and one $k$-bit addition.

Also assuming multiplications are of order $k^2$, and subtractions are of order $k$, we calculate the total number of bit operations as

$$2\,\frac{3k}{4}(k/2)^2 + 2(k/2)^2 + (k/2) + k \;=\; \frac{3k^3}{8} + \frac{k^2 + 3k}{2}.$$

On the other hand, the algorithm without the CRT would compute $M = C^d \pmod n$ directly, using $(3/2)k$ $k$-bit multiplications which require $3k^3/2$ bit operations. Thus, considering the high-order terms, we conclude that the CRT based algorithm will be approximately 4 times faster.

### 5.4.5 Recent Prime Finite Field Arithmetic Designs on FPGAs

In this Subsection, we show some of the most significant designs recently published in the open literature for modular exponentiation. All designs included in Table 5.1 were implemented either on VLSI or on reconfigurable hardware platforms. Notice also that there is a strong correlation between design's speed and the date of publication ,i.e., fastest designs tend to be the ones which have been more recently published.

Liu et al. presented in [210] a design based on the *distributed module cluster* microarchitecture especially designed to reduce long datapaths. The throughput achieved by their technique ranks as the fastest design published to date. Amanor et al. presented in [6] several designs based on different multiplier strategies. Their redundant interleaved multiplier can compute a 1024-bit RSA decryption exponentiation in just 6.1 mS. On the other hand, authors in [6] also essayed designs based on a Montgomery multiplier block,

**Table 5.1.** Modular Exponentiation Comparison Table

| Work | year | Platform | Cost | BRAMs, 18-bit M | Freq. MHz | 1024-bit time(mS) | Mult. Block Utilized |
|---|---|---|---|---|---|---|---|
| Liu et al.[210] | 2005 | 0,13$\mu m$ CMOS | 221K gates | None | 714 | 1.47 | DMC Mont. Mult. |
| Amanor et al.[6] | 2005 | Virtex | 4608 CLBs | None | 69.4 | 6.1 (est.) | Interleaved Mult. |
| Kelley et al.[170] | 2005 | Virtex II | 2847 LUTs | 5Kb, 32 | 102 | 6.6 | 16-bit Scal radix $2^{16}$ |
| Mukaida et al. [243] | 2004 | 0,11$\mu m$ CMOS | 61K gates | – | 250 | 7.3 | 64-bit Scal radix $2^4$ |
| Amanor et al.[6] | 2005 | Virtex | 8640 CLBs | None | 42.1 | 9.7 (est.) | CSA Mont. Mult. |
| Blum et al. [29] | 2001 | Virtex | 6613 CLBs | – | 45 | 12 | Mont. Mult. radix $2^4$ |
| Harris et al.[134] | 2005 | Virtex II Pro | 5598 LUTs | 5Kb, - | 144 | 16 | 16-bit Scal radix 2 |
| Kelley et al.[170] | 2005 | Virtex II | 780 LUTs | 5Kb, 8 | 102 | 22 | 16-bit Scal radix $2^{16}$ |
| Todorov[361] | 2000 | 0,5$\mu m$ CMOS | 28K gates | – | 64 | 46 | 16-bit Scal radix 8 |
| Tenca et al.[359] | 2003 | 0,5$\mu m$ CMOS | 28K gates | – | 80 | 88 | 8-bit Scal radix 2 |

but the timing performance obtained was somehow lesser than that of the interleaved multiplier. Kelley et al. presented in [170] a 16-bit Montgomery scalable multiplier of radix $2^{16}$, the highest radix for a Montgomery multiplier published to date. With that multiplier block, authors in [170] were able to achieve a 1024-bit exponentiation in just 6.6 mS. It is noted though, that the design by Kelley et al. utilized 32 embedded multipliers plus some 5K bit RAMs. Blum et al. designed in 2001 a high-radix Montgomery multiplier architecture able of achieving an exponentiation time of 12mS [29].

On the other side of the spectrum, designs by Todorov [361] and Tenca et al. [359] rank among the most economical of all high performance designs included in Table 5.1.

Due to the diversity of platforms and resources employed by the designs featured in Table 5.1, it results rather difficult to establish reasonable criteria for selecting the most efficient of all of them. Here, we say that a given design is efficient if it offers a great cost-benefit compromise. Nevertheless, the design by Mukaida et al. reported in [243] seems to be our best bet for this category. Utilizing a radix 16 multiplier implemented on ASIC at a clock speed of 250MHz, authors in [243] produced a design able to compute a 1024-bit exponentiation within 7.3mS at a hardware price of just 61K gates.

A final word about the performance comparison presented here. 1024-bit RSA exponentiation is one of the few major cryptographic primitives which shows a moderate performance speedup when hardware implementations of it are compared with its software counterparts. On this regard, Table 5.2 compares two RSA software designs against two of the fastest designs surveyed here.

As it can be seen, the speedup attained by the design in [210] is of 25.17 and 15.03 when compared with an XScale and a Pentium IV implementations, respectively.

**Table 5.2.** Modular Exponentiation: Software vs Hardware Comparison Table

| Work | year | Platform | Cost | Freq. MHz | 1024-bit time(mS) | Speedup |
|---|---|---|---|---|---|---|
| Liu et al.[210] | 2005 | 0,13$\mu m$ CMOS | 221K gates | 714 | 1.47 | 1 |
| Amanor et al.[6] | 2005 | Virtex | 4608 CLBs | 69.4 | 6.1 (est.) | 4.5 |
| Martínez-Silva et al.[219] | 2005 | IPAQ H5550 Intel XScale | – | 400MHz | 37 | 25.17 |
| López-Peza et al.[294] | 2004 | Intel Pentium IV | – | 2.4GHz | 22.10 | 15.03 |

## 5.5 Conclusions

In this Chapter we reviewed several relevant algorithms for performing efficient modular arithmetic on large integer numbers. Addition, modular addition, Reduction, modular multiplication and exponentiation were some of the operations studied throughout the material contained in this Chapter. Strong emphasis was placed on discussing the best strategies for implementing those algorithms on hardware platforms, either in the domain of ASIC designs or reconfigurable hardware platforms.

We intended to cover some of the most significant 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 the reconfigurable hardware technology.

The last Section of this Chapter contains a small survey of some of the most representative designs published in the open literature for modular exponentiation computation.