Design and Implementation of Scalable Low-Power Montgomery Multiplier

Hee-Kwan Son
Multimedia Lab, SoC R&D Center
Samsung Electronics Co., Korea
martin.son@samsung.com

Sang-Geun Oh
Multimedia Lab, SoC R&D Center
Samsung Electronics Co., Korea
stephen.oh@samsung.com

Abstract

In this paper, an efficient Montgomery multiplier is introduced for the modular exponentiation operation, which is fundamental to numerous public-key cryptosystems. Four aspects are considered: performance, power, reliability, and scalability. To increase performance, the architecture is based on the radix-4 Carry-Save Adder (CSA). To lower power consumption, we devised several effective techniques for reducing the spurious transitions and the Expected Switching Activity (ESA) of high fan-out signals. To achieve scalability, we implement a 4-fold nested loop for the whole data processing flow. It is compatible with the multiple-precision digit-serial arithmetic as well as the data transfer to/from an external memory. Lastly, to make sure that the arithmetic operation runs correctly without inducing data overflow error, we find out the optimum numbers of bits for all vectors appearing in the operation through a mathematical analysis and a logic simulation. In the evaluation of hardware implemented using 0.18 um CMOS standard library and 4 metal layers, area and current consumption are 59 K gates and 0.4 mA/MHz at 1.8 V supply voltage, respectively. The presented low power techniques save more than 20% of power consumed.

1. Introduction

Modular multiplication is widely used in secure communications. For example, the RSA [1] and the Diffie-Hellman [2] public-key cryptosystems require modular exponentiation, and this can be computed using a series of modular multiplications. Montgomery multiplication algorithm [3], [4], [13] is an efficient method for modular multiplication. The algorithm uses simple divisions by a power of two instead of divisions by a modulus, which are used in a conventional modular operation.

A number of Montgomery multipliers have been suggested in papers [5], [6], [7]. Their hardware architectures were designed to deal with a specific maximum number of bits. However, as computers' processing power increases, to provide the same "effective" security level, cryptosystems will require security parameters to be increased. To be able to cope with such coming requests, the architecture of cryptographic hardware has to be scalable.

Recently, A.F. Tenca and C.K. Koc published several papers [8], [9] about the scalable Montgomery multipliers. They introduced so-called word-based Montgomery multiplication algorithm to implement the scalability. However, they did not consider their hardware from the low power consumption point of view. Their algorithm cannot be applicable to the case when the data-path size is greater than the memory data bus width due to 2-fold nested loop structure, either.

To improve the performance by reducing the number of iterations to a half, a modified Booth recoding scheme is used for Montgomery multiplication [10]. Through this recoding scheme, the authors can achieve on-the-fly and simple calculation of a partial product. However, since they still use a triple of modulus, storing the value after pre-calculation or preparing an additional hardware to calculate it on the fly is needed. To solve this problem, a negative of modulus is used [11], instead of a triple of modulus. However, our mathematical analysis and simulation with random data prove that implementing the multiple-precision operation for scalability with a negative of modulus causes a data overflow at the intermediate result of multiplication.

In this paper, we propose an efficient architecture for a Montgomery multiplier. It is very unique in that it is developed considering all kinds of design factors such as performance, power consumption, scalability and reliability. Reliability is guaranteed through analyzing the minimum bit lengths of all internal vectors as well as input and output data.

This paper is organized as follows. Section 2 explains the digit-serial radix-4 Montgomery multiplication algorithm with only full-precision

numbers. In this section, we also introduce a specialized recoding scheme as well as the well-known Booth recoding scheme. Section 3 explains a method that allows our hardware to be scalable. Section 4 presents the structure of accumulator in detail. To guarantee reliable arithmetic calculation by preventing data overflow error Section 5 shows the optimum number of bits to be allocated to all of the vectors appearing in operation. Section 6 describes several techniques to decrease power dissipation and then explains the block diagram of data path. Experimental results, with power, area, and time measurements are given in Section 7. Finally Section 8 concludes the paper.

2. Radix-4 Montgomery Multiplication

The basic full-precision algorithm for a radix-4 digit-serial interleaved Montgomery multiplication [13] is given below:

[Inputs]
k : digit-length of M
$M = (m_{k-1},\cdots,m_1,m_0)_4 : 2 < M < 4^k$, M is odd
$A = (a_{k-1},\cdots,a_1,a_0)_4 : 0 \le A < M$
$B = (b_{k-1},\cdots,b_1,b_0)_4 : 0 \le B < M$
[Output]
$S = (s_{k-1},\cdots,s_1,s_0)_4 : 0 \le S < M$
[Method]
$S \leftarrow 0$
for i=0 **to** (k–1)
$q_i \leftarrow ((s_0 + b_i a_0)m_0') \bmod 4$
$S \leftarrow (S + b_i A + q_i M) \text{ div } 4$
endfor
if ($S \ge M$) $S \leftarrow (S - M)$ **endif**

where $R = 4^k$ and $m_0' = -m_0^{-1} \bmod 4$. Inputs A, B, and M denote multiplicand, multiplier, and modulus of the modular multiplication, respectively. The calculated result S is expressed as $S \equiv (AB + QM)R^{-1} \equiv ABR^{-1} \pmod M$. In this algorithm, q_i is the i-th significant digit of so-called Montgomery quotient $Q = (q_{k-1}, \cdots,q_1,q_0)_4$. Its value is calculated in such a way as to make the least significant digit of $(S + b_i A + q_i M)_4$ zero at each iteration. From now on, we use *PP* and *MM* to represent a partial product ($b_i A$) and a multiple of modulus ($q_i M$), respectively.

In the case of radix-4, b_i and q_i are 2-bit numbers. Thus, the value sets of *PP* and *MM* are as follows:

$$PP \in \{0, A, 2A, 3A\}, \quad MM \in \{0, M, 2M, 3M\}$$

To calculate 3A and 3M on the fly, we need two extra adders. To remove the burden of calculating 3A in the *PP*'s value set, a modified Booth recoding scheme is popularly used. Let the $b_{i,1}$ and $b_{i,0}$ are the two bits in the i-th significant digit of B. Radix-4 modified Booth recoding scheme takes a bit stream $(b_{i,1},b_{i,0},b_{i-1,1})_2$ as input and generates a recoded *PP* according to Table 1, where $b_{-1,1}$ is defined to be 0 and qa_i the recoded quotient digit for a *PP* at the i-th iteration.

TABLE 1
Booth Recoding Scheme

Three input bits			Recoded quotient for *PP*	Recoded *PP*
$b_{i,1}$	$b_{i,0}$	$b_{i-1,1}$	qa_i	*PP*
0	0	0	0	0
0	0	1	+1	+A
0	1	0	+1	+A
0	1	1	+2	+2A
1	0	0	–2	–2A
1	0	1	–1	–A
1	1	0	–1	–A
1	1	1	0	0

TABLE 2
Montgomery Recoding Scheme

Three input bits			Recoded quotient for *MM*	Recoded *MM*
$sp_{0,1}$	$sp_{0,0}$	$m_{0,1}$	qm_i	*MM*
0	0	0	0	0
0	0	1	0	0
0	1	0	–1	–M
0	1	1	+1	+M
1	0	0	+2	+2M
1	0	1	+2	+2M
1	1	0	+1	+M
1	1	1	–1	–M

Booth recoding scheme transforms the value set of *PP* into {–2A, –A, 0, +A, +2A}. All elements in the set are calculated by simple operations such as bit-inversion and/or bit-shift. However, 3M still remains in the value set of *MM*, and this problem cannot be solved by the Booth recoding scheme. In this paper, we adopt a specialized method named 'Montgomery recoding scheme' [11] to transform the original value set of *MM* into the one that also has easily obtainable elements only. Let $(sp_{0,1},sp_{0,0})_2$ be the 2 bits in the least significant digit (LSD) of $SP = S + PP$ and $(m_{0,1},m_{0,0})_2$ be the 2 bits in the LSD of M. According to the input condition that M has to be odd, $m_{0,0}$ is always '1'. Then, Montgomery recoding scheme takes a bit stream $(sp_{0,1},sp_{0,0},m_{0,1})_2$ as input and generates a recoded *MM*

according to Table 2, where qm_i is the recoded quotient digit for a *MM* at the *i*-th iteration.

Montgomery recoding scheme transforms the value set of *MM* into {–*M*, 0, +*M*, +2*M*}. Due to adopting the two recoding schemes, it is easy to calculate all the elements in the value sets of *PP* and *MM*. But the recoding schemes cause one negative effect: that is, all operands except for *M* are changed to the signed numbers that are more complicated to deal with than the unsigned ones.

3. Scalability by Multiple Precision

The basic algorithm in Section 2 uses not the multiple-precision numbers but the full-precision numbers, thus it is not scalable. And since our Montgomery multiplier exchanges its input/output data with a host such as CPU or DSP via a shared data memory, we have to consider data bus width of the memory as well as the multiple-precision operation. To make the algorithm scalable through multiple-precision operation and compatible with the data memory, we transform it into a 4-fold nested loop structure. Every operand in multiple-precision arithmetic has to be equally divided into *p* chunks, where *p* is an integer specifying the precision (1 for single precision, 2 for double precision, etc.). To help our explanation, several unit bit-lengths are introduced:

n : bit-length of modulus *M*
c : bit-length of a chunk
w : bit-length of a word

where $n = pc$, *c* is a multiple of *w*, and *w* is the data bus width of the memory. Since input/output operands have sign bits as explained in Section 2, the bit-lengths of them are extended to $n' \geq n+1$ and $n' = pc' = p(c+x)$, where c' (= $c+x$) is the bit-length of an extended chunk and *x* is a number decided by the memory's data bus configuration. In n' bits, $n'-n = p(c'-c)$ bits at the most significant side are sign bits. If the memory has 32-bit data bus and its transferable data units are 8-bit, 16-bit, and 32-bit, then $w = 32$ and $x = 8$, 16, or 32. Choosing smaller *x* leads to better performance but greater complexity in hardware. Following the trade-off rule, we choose $x = w/2$ in this paper.

To prevent the performance degradation due to carry propagation through the entire bits, the accumulator in our Montgomery multiplier has been designed using Carry-Save Adder (CSA) architecture. The accumulator sums up *PP* and *MM* generated at each iteration. We call this operation "*PPnMM accumulation*". Because of the CSA architecture, the accumulator's output is not in a conventional representation (CR) but in a redundant representation (RR) where a number is expressed as a sum of multiple numbers. Any type of Carry-Propagate Adder (CPA) can be used to convert a RR number into CR number by summation. We call this operation "*RR2CR conversion*".

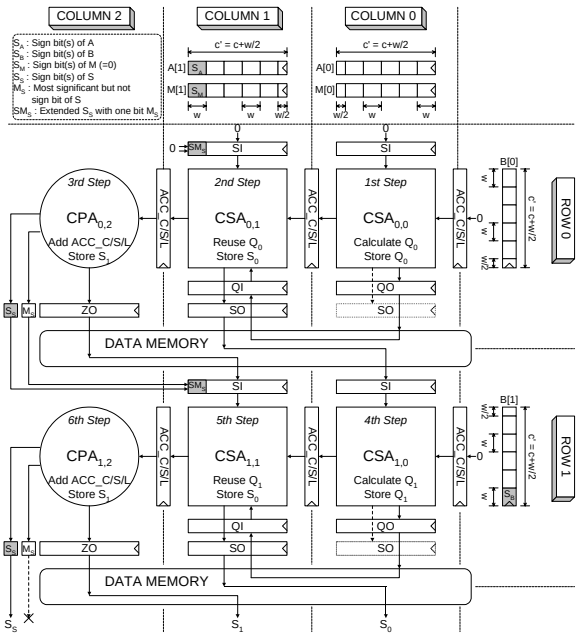

Fig 1. Processing Matrix

The operation flow diagram is shown in Fig. 1. To simplify explanation, although *p* can be any positive integer, we use $p = 2$ (i.e., double-precision case). We name the diagram "*processing matrix*" since it looks like a matrix with *p* rows and *p*+1 columns. In each row, *RR2CR conversion* is performed at the last column and *PPnMM accumulations* are performed at all others. At the first column in a row, an extended chunk of the row's Montgomery quotient *Q* is calculated and stored into a memory. At each non-first column in a row, an extended chunk of the row's result *S* is calculated and stored into the memory. The extended chunk of *Q* stored at the first column is reused in calculating each extended chunk of *S* at the non-first and non-last column in the corresponding row. Every non-first row reloads the previous row's result *SI* and uses it as the initial value of the accumulator. The *i*-th row uses the *i*-th extended chunk of *B* as well as full data of *A*, *M*, and *SI*. The *j*-th column other than the last one in a row uses the *j*-th extended chunk of *A*, *M*, and *SI*. The last row outputs the final result.

The actual data processing operation represents a 4-fold nested loop and can be explained by a pseudo code as follows:

```
/* iteration for each ext. chunk of B */
for row_idx = 0 to (p-1)
   clr_acc();
   /* iteration for each ext. chunk of A and M */
   for col_idx = 0 to (p-1)
      if (row_idx != 0)  ld_chnk_si(); endif
      ld_chnk_ai();
      ld_chnk_mi();
      ini_acc();
      /* iteration for each word of ext. chunk of B */
      for wrd_idx = -1 to (c/w)
         if (wrd_idx != -1) ld_word_bi(); endif
         if (col_idx != 0)  ld_word_qi(); endif
         /* iteration for each digit of  a word of B */
         for dgt_idx = 0 to (w/2-1)
            if ((wrd_idx == 0)&&(dgt_idx == 0))
               acc_sft_fb = 0;
            else
               acc_sft_fb = 1;
            endif
            booth_rec();
            pp_gen();
            sp_gen();
            montg_rec();
            mm_gen();
            ppnmm_acc();
            if (wrd_idx == -1) break; endif
            if (dgt_idx == w/4-1)
               if (((row_idx%2 == 0) &&
                     (wrd_idx == w/2-1)) ||
                   ((row_idx%2 == 1) &&
                     (wrd_idx == 0))) break;
               endif
            endif
         endfor
         if (wrd_idx == -1)  break; endif
         if (col_idx == 0)
            st_word_qo();
         else
            st_word_so();
         endif
      endfor
   endfor
   /* iteration for each word of Z */
   for wrd_idx = 0 to (c/w-1)
      rr2cr_cvt();
      st_word_zo();
   endfor
endfor
```

Note that *row_idx* and *col_idx* are the row index and the column index in the *processing matrix*, respectively, *wrd_idx* and *dgt_idx* are the index of a word in an extended chunk and the index of a digit in a word, respectively. In the pseudo code above, we use several data processing functions: *clr_acc()* clears the accumulator's registers, *booth_rec()* performs the Booth recoding, *pp_gen()* generates a *PP*, *sp_gen()* generates an *SP*, *montg_rec()* performs the Montgomery recoding, *mm_gen()* generates an *MM*, *ini_acc()* modifies the initial values of the accumulator's registers by summing up an extended chunk of *SI* to the accumulator, *ppnmm_acc()* accumulates a *PP* and a *MM*, and *rr2cr_cvt()* converts an RR number into a CR number. Also, there are several functions to load data from or to store data into memory: *ld_chnk_ai()*, *ld_chnk_mi()*, and *ld_chnk_si()* load an extended chunk of *A*, *M*, and *SI*, respectively, *ld_word_qi()* and *ld_word_bi()* load a word of *Q* and *B*, respectively, *st_word_qo()*, *st_word_so()*, and *st_word_zo()* store a word of *Q*, *S*, and *Z*, respectively. *Z* represents the output of *RR2CR conversion*.

4. Structure of Accumulator

The accumulator consists of 2-way 1-bit multiplexers (MUXs) and 4-2 compressors as well as three kinds of registers, ACC_C, ACC_S, and ACC_L. The bit-length of ACC_L is fixed to 3 bits. The other two registers' bit-lengths, however, are not fixed and depend on the bit-length of an extended chunk, *c'*. We will explain the optimal bit-lengths of ACC_C and ACC_S in Section 6. Among numerous circuit configurations for 4-2 compressors, we select the one depicted in Fig. 2 (a) since it consumes less power than the others [12].

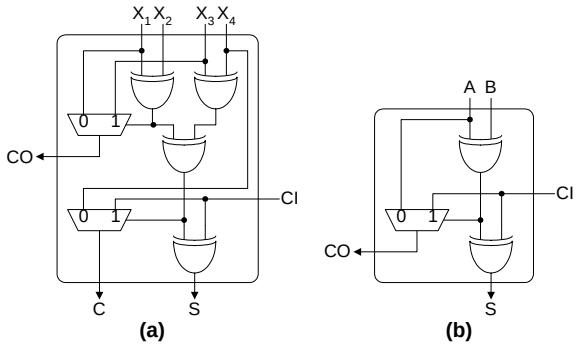

Fig 2. Circuit Diagram of (a) 4-2 Compressor and (b) Full Adder

The structure of the accumulator is illustrated in Fig. 4. Two MUXs and one compressor are located at each bit position in the accumulator. The role of the MUXs is selecting one between the shifted feedback input and the non-shifted feedback input. The feedback input

selection is controlled by ACC_SFT_FB signal: shifted one when it is '1' and non-shifted one when it is '0'. At the first cycle of each non-last column in a row, we sum up the previous row's result *SI* to the accumulator's registers by setting *PP*'s value to 0 and *MM*'s value to *SI*. This cycle is only for updating the initial values of the accumulator's registers, not for generating the row's calculation result. Thus at the next cycle, the non-shifted outputs of the accumulator's registers are fed back for that cycle's *PPnMM accumulation*. To say briefly, the accumulator uses two kinds of feedback inputs: 2-bit right shifted ones and non-shifted ones. Only at the second cycle of each non-last column, the accumulator uses non-shifted feedback inputs, and at all the other cycles, it uses shifted feedback inputs.

Except for its two lowest bit positions and three highest bit positions, the accumulator has a regular structure. All of the four 4-2 compressors at the highest bit positions use the sign bits (i.e., MSBs) of *PP* and *MM*. And the sign bits of ACC_C and ACC_S are suitably extended when they are fed back. At the two lowest bit positions, two more full adders (FAs) depicted in Fig. 2 (b) are attached to generate least significant three bits that are registered at ACC_L. The values in ACC_C and ACC_S are signed numbers but the value in ACC_L is an unsigned number. The registered value in the accumulator is 2*(ACC_C + ACC_S) + ACC_L.

5. Reliable Arithmetic

To optimize the size of hardware without causing a data overflow error, the exact number of bits has to be allocated to each vector. The right numbers of bits for all vectors excluding rows' results and ACC_C and ACC_S registers are clear. Each non-last column in a row needs *c'* bits of *A* and *M*, each row needs *c'* bits of *B*. Both *PP* and *MM* need *c'*+2 bits, inputs *A* and *B* need *pc*+1 bits including their sign bits, input *M* needs *pc* bits, ACC_L needs 3 bits, etc. Thus, we have to determine the optimum numbers of bits for ACC_C and ACC_S registers and for rows' results *S*.

First, we show how to determine the minimum numbers of bits for rows' results. For convenience of explanation, we introduce an integer *r* indicating a row in the *processing matrix*, $r \in \{0,1,\cdots, p-1\}$.

In the final row (i.e., $r = p$-1), full bits of *A*, *B*, and *M* are used in calculating the final result S_F. Thus, the ranges of inputs are as follows:

$M : 0 \le M \le 2^{pc}-1$
$A : -M+1 \le A \le M-1$
$B : -2^{pc}+1 \le B \le 2^{pc}-1$

According to Montgomery's equation, $S_F = (BA + QM) / R$, where $R = 2^{pc'}$, $Q = \sum_{i=0}^{pc'/2-1} qm_i 4^i$, $qm_i \in \{-1,0,+1,+2\}$. The maximum and the minimum values of *Q* are as follows:

$$max(Q) = \sum_{i=0}^{pc'/2-1} (+2)4^i = +\frac{2}{3}\left(2^{pc'}-1\right)$$

$$min(Q) = \sum_{i=0}^{pc'/2-1} (-1)4^i = -\frac{1}{3}\left(2^{pc'}-1\right)$$

Using the value of *R* and the boundary values for *A*, *B*, and *Q*, we can calculate the maximum and the minimum values of S_F.

$$max(S_F) = \frac{\left(2^{pc}-1\right)(M-1)+\frac{2}{3}\left(2^{pc'}-1\right)M}{2^{pc'}}$$
$$= \left(\frac{1}{2^{p(c'-c)}}+\frac{1}{3}\left(2-\frac{5}{2^{pc'}}\right)\right)M-\left(\frac{1}{2^{p(c'-c)}}-\frac{1}{2^{pc'}}\right)$$
$$\cong \frac{2}{3}M < +M$$

$$min(S_F) = \frac{\left(-2^{pc}+1\right)(M-1)-\frac{1}{3}\left(2^{pc'}-1\right)M}{2^{pc'}}$$
$$= \left(-\frac{1}{2^{p(c'-c)}}-\frac{1}{3}\left(1-\frac{5}{2^{pc'}}\right)\right)M+\left(\frac{1}{2^{p(c'-c)}}-\frac{1}{2^{pc'}}\right)$$
$$\cong -\frac{1}{3}M > -M$$

Since the bit-length of *M* is *n* ($n = pc$), n+1 bits are needed to represent S_F including its sign bit.

In an intermediate row (i.e., $0 \le r < p$-1), partial bits of *B*, B_I, are used with full bits of *A* and *M* in calculating the intermediate result S_I. Thus, the ranges of inputs are as follows:

$M : 0 \le M \le 2^{pc}-1$
$A : -M+1 \le A \le M-1$
$B_I : -2^{(r+1)c'-1} \le B_I \le 2^{(r+1)c'-1}-1$

According to Montgomery's equation, $S_I = (B_I A + Q_I M) / R_I$, where $R_I = 2^{(r+1)c'}$, $Q_I = \sum_{i=0}^{pc'/2-1} qm_i 4^i$, $qm_i \in \{-1,0,+1,+2\}$. The maximum and the minimum values of Q_I are as follows:

$$max(Q) = \sum_{i=0}^{(r+1)c'/2-1} (+2)4^i = +\frac{2}{3}\left(2^{(r+1)c'} - 1\right)$$

$$min(Q) = \sum_{i=0}^{(r+1)c'/2-1} (-1)4^i = -\frac{1}{3}\left(2^{(r+1)c'} - 1\right)$$

Using the value of R_I and the boundary values for A, B_I, and Q_I, we can calculate the maximum and the minimum values of S_I.

$$max(S_I) = \frac{\left(2^{(r+1)c'-1} - 1\right)(M-1) + \frac{2}{3}\left(2^{(r+1)c'} - 1\right)M}{2^{(r+1)c'}}$$

$$= \left(\frac{1}{2} + \frac{1}{3}\left(2 - \frac{5}{2^{(r+1)c'}}\right)\right)M - \left(\frac{1}{2} - \frac{1}{2^{(r+1)c'}}\right)$$

$$\cong \frac{7}{6}M < +2M$$

$$min(S_I) = \frac{\left(-2^{(r+1)c'-1}\right)(M-1) - \frac{1}{3}\left(2^{(r+1)c'} - 1\right)M}{2^{(r+1)c'}}$$

$$= \left(-\frac{1}{2} - \frac{1}{3}\left(1 - \frac{1}{2^{(r+1)c'}}\right)\right)M + \frac{1}{2}$$

$$\cong -\frac{5}{6} > -M$$

Since the bit-length of M is n (= pc), n+2 bits are needed to represent S_I including its sign bit.

In summary, n+1 bits are needed for the final result S_F, but n+2 bits are needed for the intermediate result S_I. The most significant two bits of a row's result are calculated at the end of the *RR2CR conversion* step. We denote the most significant bit and the next bit of a row's result SIGN_S and MS1B_S, respectively. Specifically, SIGN_S represents the sign bit of a row's result. Since S_F needs only n+1 bits, MS1B_S achieved at the final row can be dropped out.

Second, to determine the minimum number of bits to represent ACC_C and ACC_S, we take a simulation approach. A small sized data path and full set of input data are used in our simulation. Through this simulation analysis, we find out that at least c'+4 bits for ACC_C and c'+3 bits for ACC_S are needed to prevent the overflow error. It also means that we need c'+5 compressors in the accumulator.

Modular exponentiation, the mathematical model of RSA and Diffie-Hellman algorithms, consists of chained modular multiplications. If we use original Montgomery's algorithm for the modular multiplication, so-called "post-reduction step" is required after completing the iterating loop, as shown in Section 2. But for the Montgomery multiplier presented in this paper, such an additional step is not required because its output does not go beyond the value ranges of inputs. Only the last multiplication during the entire exponentiation process needs the post-reduction step.

6. Low Power Techniques

From the basis of fundamental idea explained at Section 2 to 5, we further improve our hardware to dissipate less power than the one implemented directly. Inevitably all digital devices have spurious transitions or glitches internally due to unbalanced path delays, which causes worthless dynamic power dissipation. Furthermore, if fan-outs of the glitchy signals are big, then the amount of worthlessly dissipated power is significant. Such signals are the outputs from the two circuit modules in charge of the Booth and Montgomery recodings. Because the modules comprise only combinational logic circuits according to Table 1 and Table 2, their outputs must have glitches. And the fan-outs of the outputs are c'+2 that is usually very large number. To reduce the glitching power dissipation, we put in some latches and force the outputs to pass through latches. If all flip-flops and registers capture their inputs at the clock's rising edge, then the latches are transparent when the clock is in a low state. If the outputs of the two recoding modules can reach their stable values before the clock's falling edge, none of the glitches can propagate to the fan-out modules driven by the outputs. We name these latches "*glitch blockers*". The *glitch blockers* are also very effective for reducing the glitches appearing in the accumulator since they synchronize the arrival of *PP* and *MM* at the accumulator's inputs.

Fig. 3 shows the block diagram of data path. PP generator makes a *PP* by modifying an extended chunk of *A* according to the Booth recoder's outputs, SEL_PP and EN_PP. MM generator makes a *MM* by modifying an extended chunk of *M* according to the Montgomery recoder's outputs, SEL_MM and EN_MM. Several *glitch blockers* are located at the outputs of the two recoders. To implement scalability, we have to add up *SI* at the first cycle of each non-last column in a row. Since the value set of *MM* has one less element than the one of *PP*, we put *SI* into the value set of *MM*. Thus, it is expanded into {*SI*, –*M*, 0, +*M*, +2*M*}. The meanings of the two recoders' outputs are as follows:

Booth Recoder's outputs
SEL_PP: selects *PP*'s value from {–2*A*,–*A*,+*A*,+2*A*}
EN_PP: sets *PP*'s value to 0 or not
NEG_PP: indicates that *PP*'s value is –2*A* or –*A*

Montgomery Recoder's outputs
SEL_MM: selects *MM*'s value from {*SI*,–*M*,+*M*,+2*M*}
EN_MM: sets *MM*'s value to 0 or not
NEG_MM: indicates that *MM*'s value is –*M*

When *PP* is zeroed by EN_PP, *PP* outputted from the PP generator does not depend on SEL_PP. Thus, keeping SEL_PP frozen at that time is effective for reducing power dissipation. Same reasoning also applies to SEL_MM. We place two 2-bit flip-flops and construct feedback loops for SEL_PP and SEL_MM to implement this idea.

Booth recoding scheme has a feature that +2*A* cannot follow +*A* or +2*A* and –2*A* cannot follow –*A* or –2*A*. Utilizing this feature to decrease power consumption, we suggest a binary coding method for SEL_PP as follows:

SEL_PP for +*A* = ~(SEL_PP for +2*A*)
SEL_PP for –*A* = ~(SEL_PP for –2*A*)

where '~' denotes bit-inversion. This binary coding method minimizes the ESA of SEL_PP.

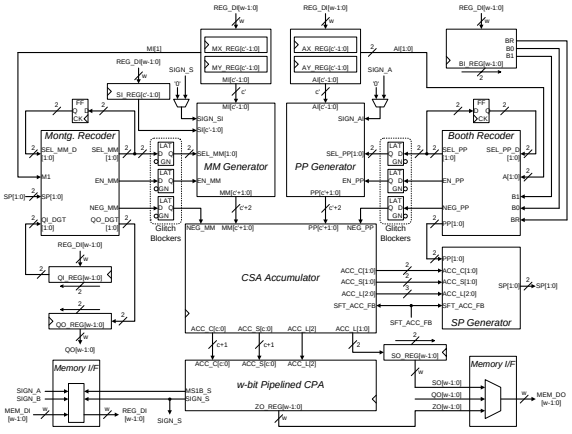

Fig 3. Block Diagram of Data Path

The whole *c'* bits of an extended chunks of *A* and *M* are used at every cycle in the non-last columns. And neighboring columns require different *c'* bits of *A* and *M*. To maximize the processing performance, we implement the *double buffering* scheme by preparing a pair of *c'*-bit registers for both *A* and *M*. While *c'* bits of *A* (and *M*) in one register is used for *PPnMM accumulations* at a column, the next *c'* bits of *A* (and *M*) is pre-loaded into the other register for operations at the next column. To the contrary, only 2 bits of inputs *B* and/or *Q* are needed and also only 2 bits of outputs *S* and/or *Q* are generated at a cycle in the non-last columns. Thus we prepare two *w*-bit parallel-in-serial-out (PISO) shift registers for inputs *B* and *Q* and two *w*-bit serial-in-parallel-out (SIPO) shift registers for outputs *S* and *Q*.

At the beginning of the last column in a row, higher significant bits of the row's result remain in the accumulator's registers without being stored into the memory. We implement a module that contains a pipelined *w*-bit CPA and undertakes three kinds of tasks: summing up the bits remaining in the accumulator's registers for the *RR2CR conversion*, storing lower significant *cp-c'* (*p*-1) bits of the sum (i.e., *Z*) into the memory, and registering next higher significant two bits of the sum (i.e., SIGN_S and MS1B_S) at flip-flops.

At each non-first cycle of the first column in a row, Montgomery recoder calculates a digit of *Q* according to Table 2. This calculation requires *SP* that is the LSD of the sum of *PP* and accumulator's feedback inputs. Since there are two kinds of feedback inputs, namely, shifted ones and none-shifted ones, determining *SP* depends on the feedback input selection. SP generator in Fig. 3 performs this task, as shown in the following pseudo code:

```
if (SFT_ACC_FB==1) /*shifted feedback inputs*/
   SP=PP[1:0]+ACC_C[1:0]+ACC_S[1:0]+ACC_L[2]
   ;
else                          /*non-shifted feedback inputs*/
   SP=PP[1:0]+ACC_L[1:0];
endif
```

7. Hardware Evaluation Results

We implement the presented Montgomery multiplier using a (512+16)-bit data path. Chunk size can be adjusted from 128-bit to 512-bit with a 32-bit step size. Using a 2-Kbyte data memory, it supports single, double, triple, and quadruple precisions. We use Synopsys' Design Compiler (DC) and Power Compiler (PC) for logic synthesis and power optimization with 0.18 um CMOS standard cell library and 4 metal layers. The size of implemented hardware is 59 K gates. It consumes 0.4 mA/MHz and runs up to 50 MHz at 1.8 V supply voltage. We show the evaluation results of current consumption in Table 3. The results prove that the low power techniques introduced in this paper are very efficient.

As a public-key cryptographic coprocessor, the presented Montgomery multiplier is embedded in the Smart Card IC implemented in Samsung Electronics. The measured time for 1024-bit RSA signing operation without using CRT (Chinese Remainder Theorem) on the Smart Card IC is 180 ms consuming 6 mA current, which is the upper limit of the GSM (Global System

for Mobile) Class-B SIM (Subscriber Identification Module) card.

TABLE 3
Current Consumption Results (@ 10 MHz)

Used EDA Tools	Current w/o Low Power Schemes	Current w/ Low Power Schemes	Current Saving Percent
DC only	5.77 mA	4.39 mA	23.9 %
DC and PC	5.00 mA	3.99 mA	20.2 %

8. Conclusion

We propose an efficient VLSI architecture and implementation methodology for a Montgomery multiplier. Scalability is achieved through the multiple-precision operation and the adjustable bit-length of a chunk. High performance is achieved through a radix-4 multiplier architecture based on CSA accumulation. Applying Booth and Montgomery recoding schemes reduces the amount of hardware resources and the length of critical path. Power optimization is achieved through decreasing the glitches and the ESA of high fan-out signals. Hardware evaluation results show that our low power techniques save more than 20% of power consumed. We also present the optimum bit-lengths for all the vectors in order to guarantee reliable calculations by preventing data overflow error. Due to its low-power and high-performance features, the presented Montgomery multiplier can be applicable to the mobile devices such as Smart Card IC and flash card IC successfully.

9. References

[1] R.L. Rivest, A. Sharmir, and L. Adleman, "A Method of Obtaining Digital Signature and Public-Key Cryptosystems," *Comm. ACM.*, vol. 21, no. 2, pp. 120-126, 1982

[2] W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Trans. Information Theory*, vol. 22, pp. 644-654, 1976

[3] P.L. Montgomery, "Modular Multiplication without Trial Division," *Math. Computing*, vol. 44, no. 170, pp. 519-521, Apr. 1985

[4] J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997

[5] C. Walter, "Systolic Modular Multiplication," *IEEE Trans. Computers*, vol. 42, no. 3, pp. 376-378, Mar. 1993

[6] P. Kornerup, "A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms," *IEEE Trans. Computers*, vol. 43, no. 8, pp. 892-898, Aug. 1994

[7] W.C. Tsai, C.B. Shung, and S.J. Wang, "Two Systolic Architecture for Modular Multiplication,", *IEEE Trans. VLSI*, vol. 8, no. 1, pp. 103-107, Feb. 2000

[8] A.F. Tenca and C.K. Koc, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm," *IEEE Trans. Computers*, vol. 52, no.9, pp. 1215-1221, Sep. 2003

[9] A.F. Tenca and C.K. Koc, "High-Radix Design of a Scalable Modular Multiplier," *Proc. Cryptographic Hardware and Embedded Systems (CHES 2001)*, pp. 189-205, May 2001

[10] J.J. Leu and A.Y. Wu, "Design Methodology for Booth-Encoded Montgomery Module Design for RSA Cryptpsystem," *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS-2000)*, pp. V.357-360, May 2000

[11] J.H. Hong and C.W. Wu, "Radix-4 Modular Multiplication and Exponentiation Algorithm for the RSA Public-key Cryptosystem," *ASP-DAC*, pp. 565-570, 2000

[12] S.F. Hsiao, M.R. Jiang and J.S. Yeh, "Design of high-speed low-power 3-2 counter and 4-2 compressor for fast multipliers," *Electronics Letters*, vol. 34, no. 4, pp. 341-343, Feb. 1998

[13] H. Orup, "Simplifying Quotient Determination in High - Radix Modular Multiplication," *Proc. 12th Symp. Computer Arithmetic*, pp. 193-199, July 1995

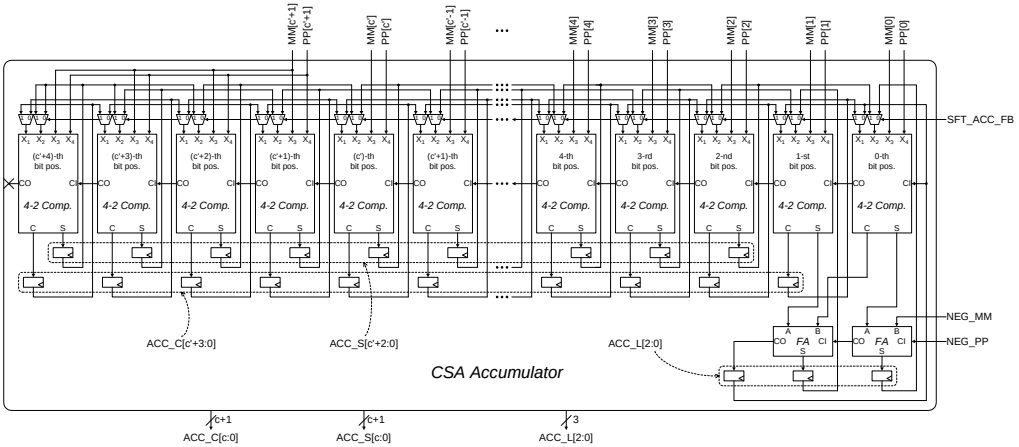

Fig 4. Circuit Diagram of Accumulator