Radix-4 Design of A Scalable Modular Multiplier

L.A. Tawalbeh

Abstract— **In this paper I am presenting an algorithm and architecture which is derived form A Scalable radix-2 Montgomery Multiplier architecture. Scalable means that a fixed-area multiplication module can handle operands of any size. In this design I applied pipelining methodology to utilize the concurrency in Montgomery Multiplication operation. In order to reduce the delay a design for radix-8 was presented in 2001. Although on one side this design reduces the delay significantly, but on the other side the design area increases. Experimental results are shown to prove that radix-4 Montgomery Multiplier has the same delay improvement gained by using radix-8 design, with approximately half the design area used .**

Keywords—**Modular Multiplier, Scalable Architecture, Cryptography, Montgomery Multiplication.**

I. Introduction

The arithmetic operations (i.e. addition, multiplication and inversion) have several applications in cryptography. The most important arithmetic operation is modular multiplication and it is used in many cryptographic algorithms such as RSA and Deffie-Helman key exchange[1]. The Montgomery Multiplier (MM) algorithm has many advantages in implementation of modular multiplication, for instance Montgomery Modular Multiplication algorithm has enabled considerable progress in speeding up of RSA cryptosystems. Perhaps the systolic array implementation stands out most in the history of its success. During its implementation in hardware, many aspects need to be considered in chip design. Among these are trade-offs between area and time, higher radix methods, carry propagation issues and communications both within the circuitry and with the rest of the world[4]. In this paper we are going to raise some concern about the trade-offs between area and time and the using of higher radix.

A. Digit Multipliers And Their Complexity

In order to get improved performance, high-radix algorithms have been proposed[3]. However, these high-radix algorithms usually are more complex and consume significant amounts of chip area, and it is not so evident whether the complex circuits derived from them provide the desired speed increase. The increase in the radix; forces the use of digit multipliers, and therefore more complex designs and longer clock cycle times are required. For this reason, low-radix designs are usually more attractive for hardware implementation[2]. Early modular multiplication designs treated radix-2, radix-8 separately at a gate level. With rapidly advancing technology, these have to be replaced by the generic radix-r, which is now essential for a better understanding of the general principles, for modular approach to design and for selecting from parameterized designs for optimal use of available chip area. Today's embedded cryptosystems are already using off-the-shelf 32-bit multipliers[4]. These r $\times$ r multipliers form the digit-by-digit products.

Of course, there is an immediate trade-off between time and area. Doubling the number of digit multipliers in a cryptotographic co-processor allows the parallel processing of twice the digits and thus reduces the time taken by half. This does not contradict the Area $\times$ Time^2 measure being constant for non-pipelined multipliers, although it appears to require less area than expected for the speed-up achieved. This indicates that choosing the largest radix possible for the given silicon area may not be the best policy; a pipelined multiplier or several rows of smaller multipliers may yield better throughput for a given area.

B. Scalability

Many designs for Montgomery Multiplication were proposed. Some of theses designs used a full precision arithmetic modules which resulted in limiting the design to a fixed degree. So, we need to design a scalable architecture which uses modules with the scalability property. Scalability of an arithmetic unit means that this unit can be reused or replicated in order to generate long precision results independently of the data path precision for which the unit was originally designed. When a need for a multiplication of larger precision arises,a new multiplier must be designed. Another way to avoid redesigning the module is to use software implementations and fixed precision multipliers[2]. However, software implementations are inefficient in utilizing inherent concurrency of the multiplication because of the inconvenient pipeline structure of the microprocessors being used. Furthermore,

Author is agraduate student at the Department of Electrical & Computer Engineering, Oregon State University, Corvallis, Oregon 97331. E-mail: tawalbeh@ece.orst.edu

software implementations on fixed digit multipliers are more complex and require excessive amount of effort in coding. Therefore, a scalable hardware module specifically tailored to take advantage of the concurrency of the Montgomery multiplication algorithm becomes extremely attractive. For example, we can reuse or replicate these modules in order to generate a long-precision results independently of the data path precision for which the module is originally designed[1]. Some of the proposed designs works on a certain field like $GF(P)$ or $GF(2^m)$, others were designed to be unified, so that they can be used in both fields.

Most proposed designs use radix-2. But the main disadvantage of using a scalable radix-2 multiplier is that it has a big delay in the critical path. This is due to using a carry propagate adders in computing the final result. So, using high radixes will minimize the delay, but as we mentioned above it will increase the area and the complexity of the design. For example, if we used radix-8 we may minimize the number of multiplications needed by one-third (since we are taking 3 bits at a time). But the design will become more complex.

From here, the idea of using radix-4 comes . The main point that I will try to make it in this paper is that we can get the same gain that we get from radix-8 in reducing the critical path delay, but with less area and less complexity.

II. Notation

. In this paper the following notation is used :

- M - modulus for modular multiplication;
- X - multiplier operand for modular multiplication;
- n - number of bits in the operands;
- w - word size;
- p - number of processing elements in the pipeline;
- e - number of words in an operand.

III. Multiple-Word Radix-2 Montgomery Multiplication (MWR2MM) Algorithm

The use of short precision words reduces the broadcast problem in the circuit implementation. The broadcast problem corresponds to the increase in the propagation delay of high-fanout signals[1]. Also, a word-oriented algorithm provides the support we need to develop scalable hardware units for the MM. Therefore, an algorithm which performs bit-level computations and produces word-level outputs would be the best choice. Let us consider w-bit words. For operands with m bits of precision, $e = \lceil \frac{n+1}{w} \rceil$ words are required. The extra bit used in the calculation of e is required since it is known that S (internal variable of the radix 2 algorithm) is in the range [0,2M-1], where M is the modulus. Thus the computations must be done with an extra bit of precision. The input operands will need an extra 0 bit value at the leftmost bit position in order to have the precision extended to the correct value[1]. We propose an algorithm in which the operand Y (multiplicand) is scanned word-by-word, and the operand X (multiplier) is scanned bit-by-bit. This decision enables us to obtain an efficient hardware implementation. We call it Multiple Word Radix-2 Montgomery Multiplication algorithm (MWR2MM. We make use of the following vectors:

$$M = (M^{(e-1)}....M^{(1)}.M^{(0)});$$
$$Y = (Y^{(e-1)}....Y^{(1)}.Y^{(0)});$$
$$X = (x_{m-1}....x_1.x_0);$$

where the words are marked with superscripts and the bits are marked with subscripts. The concatenation of vectors a and b is represented as (a,b). A particular range of bits in a vector from position i to position j, $j > i$ is represented as $a_{j..i}$. The bit position i of the k^{th} word of a is represented as a_i^k.

The details of the MWR2MM algorithm are given below.

$S = 0$ -initialize all words of S
for i = 0 to m-1
 $(C, S^{(0)}) := x_i Y^{(0)} + S^{(0)}$
 if $S_0^{(0)} = 1$ **then**
  $(C, S^{(0)}) := x_i Y^{(0)} + S^{(0)}$
  for j = 1 to e-1
   $(C, S^{(j)}) := C + x_i Y^{(j)} + M^{(j)} + S^{(j)}$
   $S^{(j-1)} := (S_0^{(j)}, S_{w-1..1}^{(j-1)}$
  $S^{(e-1)} := (C, S_{w-1..1}^{(e-1)}$
  else
   for j = 1 to e-1
    $(C, S^{(j)} := (C + x_i Y^{(j)} + S^{(j)}$
    $(S^{(j-1)}) := (S_0^{(j)}, S_{w-1..1}^{(j-1)}$
   $S^{(e-1)} := (C, S_{w-1..1}^{(e-1)}$

The MWR2MM algorithm computes a partial sum S for each bit of X, scanning the words of Y and M[1]. Once the precision is exhausted, another bit of X is taken, and the scan is repeated. Thus, the algorithm imposes no constraints to the precision of operands. The arithmetic operations are performed in precision w bits, and they are independent of the precision of operands. What varies is the number of loop iterations required to accomplish the modular multiplication. The carry variable C must be in the set $\{0, 1, 2\}$. This condition is imposed by the addition of three vectors S, M, and $x_i Y$.

IV. High-radix word-based Montgomery Algorithm

The notation used in this algorithm is shown in Table 1. This algorithm is a generalization of

(MWR2MM).

- **X - multiplier operand for modular multiplication;**
- x_j **- a single bit of X at position j;**
- X_j**- a single radix-r digit of X at position j;**
- **Y - multiplicand operand for modular multiplication;**
- **N - number of bits in the operands;**
- **r - Radix** $(r = 2^k)$**;**
- **S -partial product in the multiplication process;**
- **k - number of bits per digit in radix r;**
- qY_j **- coefficient that determines a multiple of Y which is added to the partial product S in the** j^th **iteration of the computational loop;**
- qM_j **- coefficient that determines a multiple of the modulus M which is added to the partial product S in the** j^{th} **iteration of the computational loop;**
- **BPW - number of bits in a word of either Y , M or S;**
- $NW = \lceil \frac{n+1}{BPW} \rceil$ **- number of words in either Y , M or S;**
- **NS - number of stages;**
- C_a, C_b **- carry bits;**
- $(Y^{(NW-1)}, ..., Y^{(1)}, Y^{(0)})$ **- operand Y represented as multiple words;**
- $S^{(i)}_{k-1..0}$ **- bits k - 1 to 0 of the** i^{th} **word of S.**

Table 1:

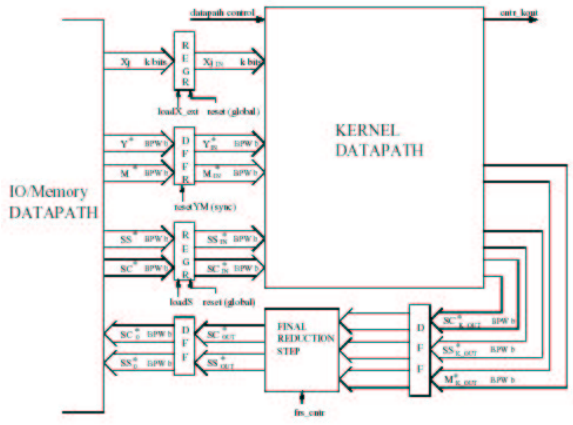
The parameter k changes depending on how many bits of the multiplier X are scanned during each loop, or in the other words, the Radix of the computation $(r = 2^k)$**. Each loop iteration (computational loop) scans k-bits of X (a radix-r digit** X_i**) and determines the value** qY**, according to Booth encoding. Booth encoding is applied to a bit vector to reduce the complexity of multiple generation in the hardware[3]. I will talk about how I used booth encoding to encode the digits of the multiplier.**

Ca **and** Cb **represent two carry bits that are propagated from the computation of one word to the computation of the next word. In order to make the least-significant** k**-bits of S all zeros,** qM_jM **is added to the partial product. This is required to avoid losing bits in the shift operation performed in Step 10[3]. In step 11 and 12 the most significant (MS) word of S is generated and sign extended. The use of Booth encoding may cause intermediate values of S to be negative. The final result in S, when Step 13 (final reduction step) is reached, is always positive and it can be a number greater than the modulus M. Its purpose is to reduce the result to a number less than the modulus. M is chosen as** $2^{N-1} < M < 2^N$ **and the result is bounded as** $0 \leq S < 2M$**. Therefore, a single subtraction of the modulus will assure that** $S < M$**, just in the case when the final result in S is greater than or equal to the modulus[3].**

V. High-radix Montgomery Multiplier - System level

For high-precision computation it is beneficial to divide the multiplicand Y, the modulus M and the result S into words[3]. The approach keeps the gates and the wire delays inside reasonable boundaries. With operands precision of thousands of bits, a conventional design to multiply all the bits at once would have a high number of pins, increased fan-in for the gates, high gate loads, and gate outputs driving long wires. The multiplications $(qY * Y)$ **and** $(qM * M)$ **shown in the MWR2kMM algorithm can be implemented by multiplexers (MUXes) and adders. The shifting operation in Step 10 is simple in hardware. Additions can be done using Carry- Save Adders (CSA), and keeping S in redundant form. With this approach the carries generated during addition are not propagated but rather stored in a separate bit-vector along with a bit-vector for the sum bits. The most complex operations of finding the coefficients qY and** qM **(steps 3 and 5) can be executed by table look-up.** qY **is pre-computed before the computational cycle begins since it depends only on the least significant** k **bits of X. This observation leaves the computation of** qM**in the most critical part of the algorithm[3]. The architecture of a Montgomery multiplier implementing the MWR2kMM algorithm is shown in Fig 1. There are two main functional blocks: Kernel and IO. Only the data path is shown. The Kernels datapath is where the computation takes place according to the algorithm. A control block (not shown) supplies the signals to synchronize the system.**

Fig 1: System Level Diagram of Modular Multiplier.



The final reduction functional block computes the final result in a suitable form for the multipliers output, implementing step 13 of the algorithm. The Kernels data path gets as inputs BPW-bit words of Y , M and S (represented in a Carry-Save form as SS and SC) and k bits of X. The outputs are BPW-bit words of the new partial product S. The superscript star (*) indicates that the signal is one word of the corresponding vector. For example, Y (*) represents one word of vector Y. These signals change every clock cycle. Depending on the kernel configuration (number of stages and word size) the operands must pass through the data path several times. The signal Xj is a k-bit signal. It provides the bits of X needed for Step 3 of the MWR2kMM algorithm. The IO block provides the interface with the user and the memory elements for the operands, modulus, and partial result. This block can be implemented in different ways depending on the application where the multiplier will be used and/or the systems architecture in which the multiplier will be integrated. The solution for this block can be flexible and the only requirement for it is to meet the timing specifications for the kernel. Therefore, the architecture of this functional unit is out of the scope of this work.

VI. Radix-4 design and analysis

The radix 8 scalable multiplier design proposed in [3] uses Booth recoding to recode the multiplier X from the digit set $\{0,1,2,3,4,5,6,7\}$ to the digit set $\{-4,-3,-2,-1,0,1,2,3,4\}$. After that the researcher generates zero, -1,1, -2,2 and -4,4 multiples of the multiplicand. These multiples cover all possible multiples of the multiplicand.

According to the multiple word Montgomery Multiplication Algorithm (MWMM), we need to add a multiples of the multiplicand at a certain step. But when dealing with radix-4 (the digit set is $\{0,1,2,3\}$, we need to decode the multiplier. So, I will use the Booth recoding to recode the multiplier into the digit set $\{-2,-1,0,1,2\}$ to get rid of the multiple 3 .

Booth recoding is done according to the following equation : Booth $(X_i, x_{i-1}) = -2x_{i+1} + x_i + x_{i-1}$. Where $X_i = (x_{i+1}, x_i)$. Each time we perform recoding we step the index by 2.

Example : if we have the $X = (01101)_2 = (13)_{10}$.then the recoded multiplier will be :
i = 0 : 010 $\Rightarrow 1$, I added 0 at position i = -1.
i = 2 : 110 $\Rightarrow \overline{1}$,
i = 4 : 001 $\Rightarrow 1$, I added 0 at position i = 5 .
So Booth(X)= $(1\overline{1}1)_4$. Where $\overline{1}$ means negative one .

It is possible to optimize the recoding circuit proposed in[3], and reduce its effect on the critical path delay. One important issue, related to adding a modulus M to the partial product during the computations according to the algorithm. This addition required to make the least significant k bits (in radix-4 we have k=2) of the partial product equals zero, so there will be no lost information during the shifting. In order to achieve this certain multiples qM of the modulus M should be added. But since we are using radix-4, we need to deal with two bits at a time. So, there will be four possible choices for the modulo multiples (qM) to be added. Choice should be made according to certain conditions and tests that must be performed.

The proposed approach can be used as follows: Let $S^{(0)}$ to be the least significant word of the partial product, and $M^{(0)}$ the least significant word of the modulus M. To find qM we need to test the least two significant bits of both $S^{(0)}$ and $M^{(0)}$(let us call them $S_{10}^{(0)}$ and $M_{10}^{(0)}$ respectively):
• If $S_{10}^{(0)} = 00$, so nothing will be added, so qM=0.
• If $S_{10}^{(0)} = 10$ then we need to add 2M, so qM = 2. (note that M is prime and so it is odd, so the least significant two bits should be either 01 or 11). In both cases we add 2M = 10 to $S_{10}^{(0)} = 10$ to give the required result 00
• If $S_{10}^{(0)} = 00$ or $S_{10}^{(0)} = 11$:

$S_{10}^{(0)}$	$M_{10}^{(0)} = 11$	$M_{10}^{(0)} = 01$
11	**Sub M**	**Add M**
01	**Add M**	**Sub M**

In addition we have qM =1, and in subtraction we have $qM = \overline{1}$. So, I suggest to encode the multiples of M into the digit set $\{-1,0,1,2\}$. By taking a look to the different combinations that we have above, we can see that we need to subtract the modulo M only when the XORING of the least two significant bits of $M^{(0)}$ and $M^{(0)}$ is = 00. So, we have $qM = \overline{1}$. What I am trying to say that we can apply an XORING test to the least significant two bits of $M^{(0)}$ and $M^{(0)}$, and then decide the value of the multiples of M that should be added .

XORING Result :
• 00 $\Rightarrow$ $qM = \overline{1}$.
• 10 $\Rightarrow$ $qM = 1$.
We notice that the first bit of the XORING result in this two cases is zero .

Now if we took $S_{10}^{(0)} = 00$, this is the case when we don't need to add any thing(qM = 0), The possible results of XORING it with the two possible values of $M_{10}^{(0)}$ are:
00 XOR 01 / 11 $\Rightarrow$ 01 / 11 .
Also by taking $S_{10}^{(0)} = 10$,this is the case when we need to add 2M.(qM = 2) we get the following XORING result :
10 XOR 01 / 11 $\Rightarrow$ 11 / 01.
The above results are summarized in the following table :

$S_{10}^{(0)}$	$M_{10}^{(0)}$	$XORINGResult$
00	**01**	**01**
00	**11**	**11**
10	**01**	**11**
10	**11**	**01**

Table shows that the first bit of the XORING result in the above two cases is always one. So, we can minimize this test. We need to look at the first bit of this XORING results. If it is 0, it means that (qM = 1 or = $\overline{1}$), and then we can take the second bit to decide which to choose (for instance we can use this 2^{nd} bit as a select control for a mux which has M and M complement as inputs).

Same way, if the 1^{st} bit = 1, then it means that we need to choose between 2M and 0M. So, again we use the 2nd bit to decide by using it as a mux select control that has 2M and zero as inputs. Note that 2M can be easily generated by shifting M one position to the left.

Using the information explained in the above discussion we can reduce the complexity of the design and make it fit into a specific area.

VII. Conclusion

In this paper we presented a new architecture for implementing the Montgomery multiplication. The main difference of our design from other proposed designs beside it is scalable to any operand size, it is using radix-4. and it can be adjusted to any available chip area. The proposed architecture is also flexible. In this paper, I proved that radix-4 Montgomery Multiplier reduces the critical path delay, by the same amount that we can get from the proposed radix-8 design. In addition to that radix-4 design area and complexity is almost half of radix-8 design. Figure2 shows a comparison of design area between radix-2 and radix-4 and radix-8.

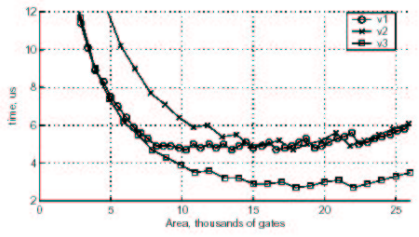


Fig 2: Area×time comparison between radix-2($v1$), radix-8($v2$), and radix-4($v3$)for 256-bit operands.

References

[1] A. F. Tenca and Ç. K. Koç, "A Scalable Architecture For Montgomery Multiplication," in *Cryptographic Hardware and Embedded Systems - CHES 1999* , Ç. K. Koç and C. Paar , Ed. 1999, Lecture Notes in Computer Science, No. 1717, pp. 94–108, Springer, Berlin, Germany.

[2] E. Savas,A. F. Tenca and Ç. K. Koç, "A Scalable and Unified Multiplier Architecture For Finite Fields GF(p) and GF(2^m)," in *Cryptographic Hardware and Embedded Systems - CHES 2000* , Ç. K. Koç and C. Paar , Ed. 2000, Lecture Notes in Computer Science, No. 1717, pp. 281–296, Springer, Berlin, Germany.

[3] A. F. Tenca and G. Todorov and C. K. Koc, "High Radix Design of a Scalable Modular Multiplier," in *Cryptographic Hardware and Embedded Systems - CHES 2001*,Ç. K. Koç and C. Paar , Ed. 2001, Lecture Notes in Computer Science, No. 1717, pp. 189–206, Springer, Berlin, Germany.

[4] Collin D. Walter, " Montgomery's Multiplication technique : How to make it smaller and faster," in *Cryptographic Hardware and Embedded Systems - CHES 1999* , Ç. K. Koç and C. Paar , Ed. 1999, Lecture Notes in Computer Science, No. 1717, pp. 80–93, Springer, Berlin, Germany.

[5] A. F. Tenca and C. K. Koc, "Word-based Algorithm and Scalable Architecture for Montogomery Multiplication,"

[6] H. Wu, " Montgomery Multiplier and Squarer in GF(2^m)," in *Cryptographic Hardware and Embedded Systems - CHES 2001*, Ç. K. Koç and C. Paar , Ed. 2001, Lecture Notes in Computer Science, No. 1717, pp. 107–120, Springer, Berlin, Germany.