Efficient algorithm and implementation of Montgomery Multiplication Using Reconfigurable Hardware

L.A. Tawalbeh and M.H Sinky

Abstract— **In this paper we are presenting an efficient algorithm and architecture for Montgomery multiplication. This algorithm is derived from a Scalable high radix Montgomery Multiplication algorithm presented and proved to be correct in [1]. Radix-4 is widely used in arithmetic, and so we are presenting a radix-4 based scalable algorithm. Scalable means that a fixed-area multiplication module can handle operands of any size. In this design we applied a 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, on the other side the design area increases. In addition to exploring reconfigurable hardware techniques to implement the Montgomery Multiplier under discussion we explore other techniques to get around the area problems of a radix-8 design. The word-by-word algorithm used in the multiplier gives designer the freedom to select the level of parallelism according to the available area. Experimental results are shown to demonstrate that the proposed radix-4 Montgomery Multiplier design has better area/performance tradeoff than previous radix-2 and 8 scalable designs.**

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

I. Introduction

Modular arithmetic operations (i.e., addition, multiplication and inversion) are used in several cryptographic applications, such as decipherment operation of RSA algorithm [2], Diffie-Hellman key exchange algorithm [3], elliptic curve cryptography [4], and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm [5]. The most important of these three arithmetic operations is the modular multiplication operation since it is the core operation in many cryptographic functions.

Authors are graduate students at the Department of Electrical & Computer Engineering, Oregon State University, Corvallis, Oregon 97331. E-mail: `tawalbeh,sinky`@ece.orst.edu

Given the increasing demands on secure communications, cryptographic algorithms will be embedded in almost every application involving exchange of information. Some of these applications, such as smart cards [6] and hand-helds, require hardware restricted in area and power resources [7]. An efficient algorithm to implement modular multiplication is the Montgomery Multiplication algorithm [8], and it has many advantages over ordinary modular multiplication algorithms. The main advantage is that the division step in taking the modulus is replaced by shift operations which are easy to implement in hardware [7].

Cryptographic applications use large number of bits in order to be considered secure. Some of these applications use 256-bit precision operands, others use larger precision, up to 2048 or 4096, as in some exponentiation-based cryptographic applications.

Many of the proposed designs are fixed-precision [9] which uses operands of fixed size. Other designs are scalable [10], [1], and can take operands with an arbitrary precision. An important factor that should be taken into consideration is the area/time tradeoff [11]. In general the fastest design is better, but most of the fast designs use large area and more complicated logic.

II. Different Multipliers and Implementations

The scalability feature of the Montgomery Multiplier under discussion makes it ideal for FPGA implementations. An increase of change in the word size for a particular system simply requires reconfiguring the system with the appropriate number of replicated multiplier modules. Three types of multipliers that are considered in cryptographic systems are:

- **General-purpose**
- **Multiplication by a constant**
- **Multiplication using a redundant coding scheme**

The general-purpose multiplier is the most expensive to implement in that the operands can assume any value. In the realm of cryptography, however, a multiplication of $n \times n$ generally requires a result of only n bits wide [12] which is how the Montgomery Multiplier should be used within a cryptographic application, ignoring the unnecessary bits

and thus avoiding extra hardware for implementation. This is the first optimization that may be exploited in hardware implementations whether they be reconfigurable architectures or ASIC designs.

The second instance of a multiplier is where one of the operands is a constant which is again common in cryptographic applications. In hardware, such multipliers can be made considerably smaller and faster than general-purpose multipliers. “On average, single-operand multipliers of this type are half the size and twice as fast as their general-purpose counterparts [12].” The implementation of our Montgomery Multiplier combined with the optimizations of a constant operand can yield significant gains.

The final form of multipliers listed, is one that makes use of a redundant coding scheme. Again this provides gains as it reduces the number of partial products required to compute the result. The Montgomery multiplier under discussion exhibits this technique. This provides considerable gains when compared to general purpose multipliers.

III. Digit Multipliers And Their Complexity

In order to get improved performance, high-radix algorithms have been proposed[1]. 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.

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[13]. 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 cryptographic 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.

IV. Scalability

Many designs for Montgomery Multiplication were proposed. Some of these 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. However, software implementations are inefficient in utilizing inherent concurrency of the multiplication because of the inconvenient pipeline structure of the microprocessors being used. Furthermore, 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 long-precision results independently of the data path precision for which the module was originally designed[10]. Some of the proposed designs work 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 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.

V. Notation

The notation used in the High-radix MM algorithm is shown below:

Step
1: $S := 0$
$x_{-1} := 0$
2: FOR $j := 0$ TO N - 1 STEP k
3: $q_Y = Booth(x_{j+k-1..j-1})$
4: $S := S + (q_Y * Y)$
5: $q_{M_j} := S_{k-1..0} * (2^k - M^{-1}_{k-1..0})\ mod\ 2^k$
6: $S := signext(S + q_{M_j} * M)/2^k$
END FOR;
7: IF $S \geq M$ THEN $S := S - M$
END IF;

Fig. 1. High-Radix ($Radix - 2^k$) Montgomery Multiplication (R2^kMM) Algorithm

- **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.**

VI. High-radix word-based Montgomery Algorithm

The high radix MM algorithm is given below.

The parameter k **changes depending on how many bits of the multiplier** X **are scanned during each loop, or in 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** q_Y**, according to Booth encoding. Booth encoding is applied to a bit vector to reduce the complexity of multiple generation in the hardware[1]. 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[1]. 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[1].**

VII. 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[1]. 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 $q_Y * Y$ **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** q_Y **and** q_M **(steps 3 and 5) can be executed by table look-up.** q_Y **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 [1]. 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.

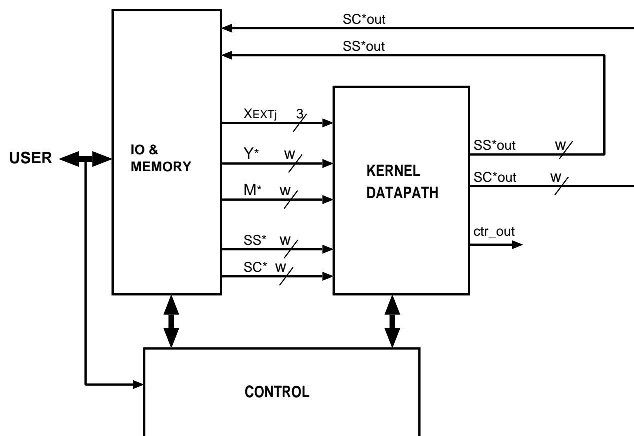

Fig. 2. System Level Diagram of Modular Multiplier.

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 X_j 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.

VIII. Efficiency of Reconfigurable Architectures

Reconfigurable architectures with respect to cryptography are ideal in that they shorten the time-to-market a product, provide a great deal of flexibility and at the same time are faster than software implementations. Flexibility is important in cryptography because implemented algorithms need to respond to flaws within the algorithm itself and/or changes in standards [12]. ASIC designs do not offer this capability and thus are unattractive for dynamic cryptographic algorithms.

Since the aim of our Montgomery Multiplier is to be used in cryptographic algorithms we are targeting FPGA implementations for our system. It is worth noting that general-purpose processors would be inefficient for specific implementations of cryptographic algorithms due to variable or unconventional bit-widths, no pipeline exploitation, constant propagation for reduced complexity, and other factors [12]. Therefore, a typical cryptographic system would comprise of a general-purpose processor in conjunction with a coprocessor as implemented in [14] for the Crypto-Booster to accelerate cryptographic operations.

When speaking of common cryptographic applications, two main forms of input are considered [12]:

- Stream-based
- Custom-instructions

For stream-based functions a large input stream is processed producing a large data output stream, whereas custom-instructions take in a limited number of inputs producing a few outputs.

Cipher components are built on various arithmetic operations, where among the most difficult is multiplication. Multipliers in hardware consume a large amount of area and compute the results very slowly as mentioned before. The implementation of our proposed Montgomery multiplier is much more feasible than other implementations.

Taking into consideration a stream-based function, we would like to increase the throughput of our multiplier as much as possible. Given the fact that it is already a radix-4 system, the throughput is very much comparable to a radix-2 implementation as mentioned before. However, as we begin to increase the number base, the overall gains achieved begin to decrease, i.e. area/trade performance. As we mentioned before, for a radix-8 implementation the penalty paid in area was costly.

To get around this problem we propose the idea of using the flexibility of reconfigurable architectures to our advantage.

If an unacceptable amount of area is consumed in a radix-8 implementation we can use the concept of run-time reconfigurability. The idea behind Run-Time Reconfigurable (RTR) systems is the ability to configure a reprogrammable device at run-time, or while it is executing. Run-time reconfiguration lifts the resource limitations imposed by the device and gives designers even greater flexibility in in the implementation of their designs. Area is not an issue anymore and we only have to deal with the latency which can be cleverly hidden if we are careful with the implementation.

In the case of RTR designs, the execution of a system is divided over time. The reconfigurable device executes one portion of the overall design at one instance or phase and is then reconfigured

in order to run other parts of the design at a later time. In this manner one chip can fit large designs that would not be possible without the use of reconfigurability at run-time.

This is much more efficient than software based approaches as it exhibits a hardware architecture resembling that of an ASIC. However, it has the flexibility of software as opposed to the custom ASIC in that it can be updated during run-time. The main drawback to such an approach is that the design process becomes more complex as you must consider the added dimension of time.

Use of partial reconfiguration with respect to a run-time reconfigurable design is done by overlapping execution and reconfiguration of the system. Ganesan and Vemuri [15] put into practice a temporally functional design approach to solve this problem.

In terms of overlap between the execution of a portion of the system and the reconfiguration of another part of the chip, the goal is to hide the reconfiguration overhead which in turn improves the design latency.

The authors provide a simple design methodology which includes two steps:

- Partitioning the design into a sequence of temporal segments.
- Enter a pipelining phase where the execution of each temporal partition is pipelined with the reconfiguration of the following partition.

Figure 3 provides a visual illustration of these steps [15].

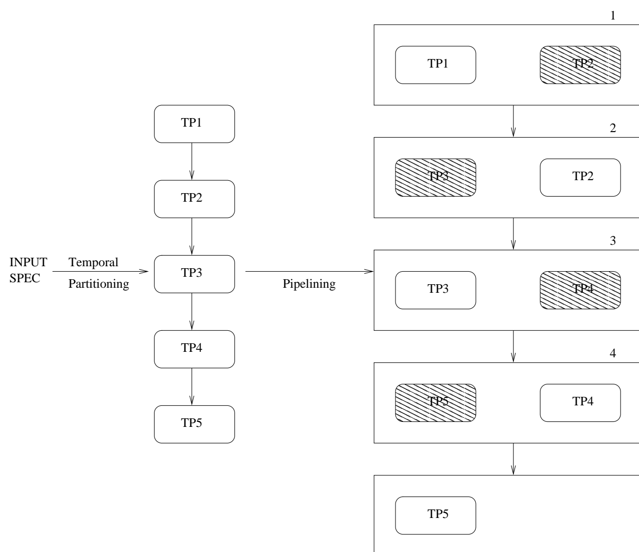

Fig. 3. Temporal partitioning and pipelining.

Given the fact that our design consists of a 2-stage pipeline, for the radix-8 design we would preconfigure the device for the first stage. Following completion of the first pipeline stage, the device would be configured for the next stage to complete the operation. Making use of a partially reconfigurable device would allow the rest of the cryptographic system to remain undisturbed while reconfiguration is taking place. As we mentioned earlier the drawback of complexity in time, another drawback is the use of an external memory to store the intermediate results between reconfigurations. For stream based inputs that would be required and the size of the memory is directly proportional to the length of the input stream.

As we have mentioned with regard to the radix-8 MM design, the area was not feasible and at the same time the complexity of the implementation yielded a long critical path. Evidently, a RTR approach to the radix-8 MM would solve area constraints, however we are not able to avoid the critical path delay. This would have to be done by introducing more pipeline stages. An assessment of how the throughput is affected and compared against radix-4 and 2 designs after shortening the critical path needs to be done with respect to a radix-8 design. Otherwise we come back to the conclusion that the radix-4 design is the best performance-wise.

IX. Conclusion

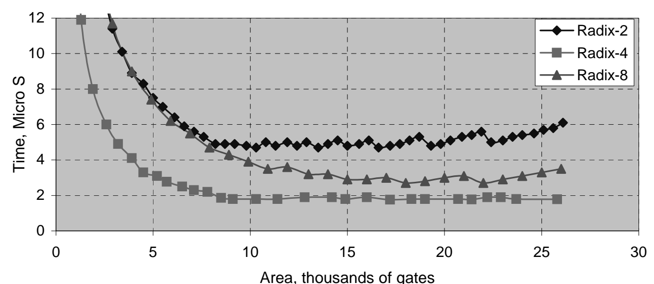

Fig. 4. Area×time comparison between radix-2, radix-8, and radix-4 for 256-bit operands.

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. It can be adjusted to any available chip area. We also considered the techniques involved in mapping the architecture to reconfigurable hardwares and their efficiency. Run-time reconfigurability was proposed to reduce the amount of area consumed for a radix-8 implementation of the Montgomery Multiplier. However, in terms of throughput, a radix-4 design still provides the best results.

References

[1] A. F. Tenca, G. Todorov, and Ç. K. Koç, "High-radix design of a scalable modular multiplier," in *Crypto-*

graphic Hardware and Embedded Systems — CHES 2001, Ç. K. Koç and C. Paar, Eds. 2001, Lecture Notes in Computer Science, No. 1717, pp. 189–206, Springer, Berlin, Germany.

[2] L. Adleman, R. L. Rivest, and A. Shamir, "A method for obtaining digital signature and public-key cryptosystems," *Comm. of the ACM*, vol. 21, no. 2, pp. 120–126, February 1978.

[3] M. E. Hellman and W. Diffie, "New directions on cryptography," *IEEE transactions on Information Theory*, vol. 22, pp. 644–654, November 1976.

[4] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, January 1987.

[5] National Institute for Standards and Technology, "Digital signature standard (dss)," Tech. Rep. 168-2, FIPS PUB, January 2000.

[6] D. M'Raihi and D. Naccache, "Cryptographic smart cards," *IEEE Micro*, vol. 16, no. 3, pp. 14–23, June 1996.

[7] G. Todorov, "ASIC design, implementation and analysis of a scalable high-radix Montgomery multiplier," Master thesis, Oregon State University, USA, December 2000.

[8] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, April 1985.

[9] E. F. Brickel, "A fast modular multiplication algorithm with application to two key cryptography," in *Advances in Cryptography - CRYPTO '82*. 1983, pp. 51–60, Plenum, New York.

[10] A. F. Tenca and C. K. Koc, "A word-based algorithm and scalable architecture for montgomery multiplication," in *Cryptographic Hardware and Embedded Systems — CHES 1999*, Ç. K. Koç and C. Paar, Eds. 1999, Lecture Notes in Computer Science, No. 1717, pp. 94–108, Springer, Berlin, Germany.

[11] C. D. Walter, "Space/time trade-offs for higher radix modular multiplication using repeated addition," *IEEE Transactions on computing*, vol. 46, no. 2, pp. 139–141, February 1997.

[12] R. R. Taylor and S. C. Goldstein, "A high-performance flexible architecture for cryptography," in *Cryptographic Hardware and Embedded Systems - CHES 1999*, Ç. K. Koç and C. Paar, Eds. 1999, Lecture Notes in Computer Science No. 1717, pp. 231–245, Springer, Berlin, Germany.

[13] C. D. Walter, "Montogomery's multiplication technique : how to make it smaller and faster," in *Cryptographic Hardware and Embedded Systems — CHES 1999*, Ç. K. Koç and C. Paar, Eds. 1999, Lecture Notes in Computer Science, No. 1717, pp. 80–93, Springer, Berlin, Germany.

[14] E. Mosanya, C. Teuscher, H. F. Restrepo, P. Galley, and E. Sanchez, "CryptoBooster: A reconfigurable and modular cryptograpic coprocessor," in *Cryptographic Hardware and Embedded Systems - CHES 1999*, Ç. K. Koç and C. Paar, Eds. 1999, Lecture Notes in Computer Science No. 1717, pp. 246–256, Springer, Berlin, Germany.

[15] Satish Ganesan and Ranga Vemuri, "An integrated temporal partitioning and partial reconfiguration technique for design latency improvement," *Design, Automation and Test in Europe*, pp. 320–325, 2000.