A Scalable Architecture for Montgomery Multiplication *

Alexandre F. Tenca and Çetin K. Koç

Electrical & Computer Engineering
Oregon State University, Corvallis, Oregon 97331
{tenca,koc}@ece.orst.edu

Abstract. This paper describes the methodology and design of a scalable Montgomery multiplication module. There is no limitation on the maximum number of bits manipulated by the multiplier, and the selection of the word-size is made according to the available area and/or desired performance. We describe the general view of the new architecture, analyze hardware organization for its parallel computation, and discuss design tradeoffs which are useful to identify the best hardware configuration.

1 Introduction

The Montgomery multiplication algorithm [10] is an efficient method for modular multiplication with an arbitrary modulus, particularly suitable for implementation on general-purpose computers (signal processors or microprocessors). The method is based on an ingenious representation of the residue class modulo M, and replaces division by M operation with division by a power of 2. This operation is easily accomplished on a computer since the numbers are represented in binary form. Various algorithms [11,7,1] attempt to modify the original method in order to obtain more efficient software implementations on specific processors or arithmetic coprocessors, or direct hardware implementations. In this paper we are interested in hardware implementations of the Montgomery multiplication operation.

Several algorithms and hardware implementations of the Montgomery multiplication for a limited precision of the operands were proposed [1,11,3]. In order to get improved performance, high-radix algorithms have also been proposed [11, 8]. 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. A theoretical investigation of the design tradeoffs for high-radix modular multipliers is given in [15]. An example of a design in radix-4 is shown in [13]. The increase in the radix forces the use of digit multipliers, and therefore more complex designs and longer

* Readers should note that Oregon State University has filed or will file a patent application containing this work to the US Patent and Trademark Office.

Ç.K. Koç and C. Paar (Eds.): CHES'99, LNCS 1717, pp. 94–108, 1999.

clock cycle times. For this reason, low-radix designs are usually more attractive for hardware implementation.

The Montgomery multiplication is the basic building block for the modular exponentiation operation [4,5] which is required in the Diffie-Hellman and RSA public-key cryptosystems [2,12]. Currently, most modular exponentiation chips perform the exponentiation as a series of modular multiplications, which is the most compelling reason for the research of fast and inexpensive modular multipliers for long integers. Recent implementations of the Montgomery multiplications are focused on elliptic key cryptography [9] over the finite field $GF(p)$. The introduction [6] of the Montgomery multiplication in $GF(2^k)$ opened up new possibilities, most notably in elliptic key cryptography over the finite field $GF(2^k)$ and discrete exponentiation over $GF(2^k)$ [5].

In this paper, we propose a *scalable* Montgomery multiplication architecture, which allows us to investigate different areas of the design space, and thus, analyze the design tradeoffs for the best performance in a limited chip area. We start with a short discussion of the scalability requirement which we impose in our design, and then give a presentation of the general theoretical issues related to the Montgomery multiplication. We then propose a word-based algorithm, and show the parallel evaluation of its steps in detail. Using this analysis, we derive an architecture for the modular multiplier and present the design of the module. We also perform simulations in order to provide area/time tradeoffs and give a first order evaluation of the multiplier performance for various operand precision.

2 Scalability

We consider an arithmetic unit as scalable if

> *the 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.*

For example, a multiplier designed for 768 bits [13] cannot be immediately used in a system which needs 1,024 bits. The functions performed by such designs are not consistent with the ones required in the larger precision system, and the multiplier must be redesigned. In order to make the hardware scalable, the usual solution is to use software and standard digit multipliers. The algorithms for software computation of Montgomery's multiplication are presented in [6,7]. The complexity of software-oriented algorithms is much higher than the complexity of the radix-2 hardware implementation [1], making a direct hardware implementation not attractive. In the following, we propose a hardware algorithm and design approach for the Montgomery multiplication that are attractive in terms of performance and scalability.

3 Montgomery Multiplication

Given two integers X and Y, the application of the radix-2 Montgomery multiplication (MM) algorithm with required parameters for n bits of precision will result in the number

$$Z = \mathrm{MM}(X, Y) = XYr^{-1} \bmod M \ , \tag{1}$$

where $r = 2^n$ and M is an integer in the range $2^{n-1} < M < 2^n$ such that $\gcd(r, M) = 1$. Since $r = 2^n$, it is sufficient that the modulus M is an odd integer. For cryptographic applications, M is usually a prime number or the product of two primes, and thus, the relative primality condition is always satisfied. The Montgomery algorithm transforms an integer in the range $[0, M-1]$ to another integer in the same range, which is called the image or the M-residue of the integer. The image or the M-residue of a is defined as $\overline{a} = ar \bmod M$. It is easy to show that the Montgomery multiplication over the images $\overline{a}$ and $\overline{b}$ computes the image $\overline{c} = \mathrm{MM}(a, b)$ which corresponds to the integer $c = ab \bmod M$ [7]. The transformation between the image and the integer set is accomplished using the MM as follows.

- From the integer value to the M-residue: $\overline{a} = \mathrm{MM}(a, r^2) = ar^2r^{-1} \bmod M = ar \bmod M$.
- From the M-residue to the integer value: $a = \mathrm{MM}(\overline{a}, 1) = arr^{-1} \bmod M = a \bmod M$.

Provided that $r \pmod M$ and $r^2 \pmod M$ are precomputed and saved, we need only a single MM to perform either of these transformations. The tradeoff is the lower complexity of the MM algorithm when compared to the conventional modular multiplication which requires a division operation. Another important aspect of the advantage of the MM over the conventional multiplication is exposed in modular exponentiation, when multiple MMs are computed over the M-residues before the result is translated back to the original integer set.

The radix-2 Montgomery multiplication algorithm for m-bit operands $X = (x_{m-1}, ..., x_1, x_0)$, Y, and M is given as:

The Radix-2 Algorithm
$S_0 = 0$
for $i = 0$ to $m - 1$
    if $(S_i + x_iY)$ is even
        then $S_{i+1} := (S_i + x_iY)/2$
        else $S_{i+1} := (S_i + x_iY + M)/2$
if $S_m \geq M$ then $S_m := S_m - M$ — the final correction step

This algorithm is adequate for hardware implementation because it is composed of simple operations: word-by-bit multiplication, bit-shift (division by 2), and addition. The test of the *even* condition is also very simple to implement, consisting on checking the least significant bit of the partial sum S_i to decide if the

addition of M is required. However, the operations are performed on full precision of the operands, and in this sense, they will have an intrinsic limitation on the operands' precision. Once a hardware is defined for m bits, it cannot work with more bits.

4 A Multiple-Word Radix-2 Montgomery Multiplication 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. 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, $e = \lceil (m+1)/w \rceil$ words are required. The extra bit used in the calculation of e is required since it is known that S (internal varible 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.

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:

$$\begin{aligned} M &= (M^{(e-1)}, ..., M^{(1)}, M^{(0)}) \ , \\ Y &= (Y^{(e-1)}, ..., Y^{(1)}, Y^{(0)}) \ , \\ X &= (x_{m-1}, ..., x_1, x_0) \ , \end{aligned}$$

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 a 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.

The MWR2MM Algorithm

$S = 0$ — 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
    $(C, S^{(0)}) := (C, S^{(0)}) + M^{(0)}$
    for $j = 1$ to $e - 1$
      $(C, S^{(j)}) := C + x_i Y^{(j)} + M^{(j)} + S^{(j)}$
      $S^{(j-1)} := (S_0^{(j)}, S_{w-1..1}^{(j-1)})$

$$
\begin{array}{l}
\quad S^{(e-1)} := (C, S^{(e-1)}_{w-1..1}) \\
\text{else} \\
\quad \text{for } j = 1 \text{ to } e-1 \\
\qquad (C, S^{(j)}) := C + x_i Y^{(j)} + S^{(j)} \\
\qquad S^{(j-1)} := (S^{(j)}_0, S^{(j-1)}_{w-1..1}) \\
\quad S^{(e-1)} := (C, S^{(e-1)}_{w-1..1})
\end{array}
$$

The MWR2MM algorithm computes a partial sum S for each bit of X, scanning the words of Y and M. 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 the three vectors S, M, and $x_i Y$. To have containment in the addition of 3 w-bit words and a maximum carry value C_{max} (generated by previous word addition), the following equation must hold:

$$3(2^w - 1) + C_{max} \leq C_{max} 2^w + 2^w - 1$$

which results in $C_{max} \geq 2$. Thus, choosing $C_{max} = 2$ is enough to satisfy the containment condition. The carry variable C is represented by two bits.

5 Parallel Computation of the MWR2MM

In this section we analyze the data dependencies on the proposed algorithm (MWR2MM) giving more information on the its potential parallelism and investigating parallel organizations suitable for its implementation.

The dependency between operations within the loop for j restricts their parallel execution due to dependency on the carry – C. However, parallelism is possible among instructions in different i loops. The dependency graph for the MWR2MM algorithm is shown in Figure 1. Each circle in the graph represents an atomic computation and is labeled according to the type of action performed. Task A corresponds to three steps: (1) test the least significant bit of S to determine if M should be added to S during this and next steps, (2) addition of words from S, $x_i Y$, and M (depending on the test performed), and (3) one-bit right shift of a S word. Task B corresponds to steps (2) and (3). We observe from this graph that the degree of parallelism and pipelining can be very high.

Each column in the graph may be computed by a separate processing element (PE), and the data generated from one PE may be passed to another PE in a pipelined fashion. An example of the computation executed for 5-bit operands is shown in Figure 2 for the word size of $w = 1$ bit. Since the j^{th} word of each input operand is used to compute word $j-1$ of S, the last B task in each column must receive $M^{(e)} = Y^{(e)} = 0$ as inputs. This condition is enough to guarantee that $S^{(e-1)}$ will be generated based only on the internal PE information. Note also that there is a delay of 2 clock cycles between processing a column for x_i

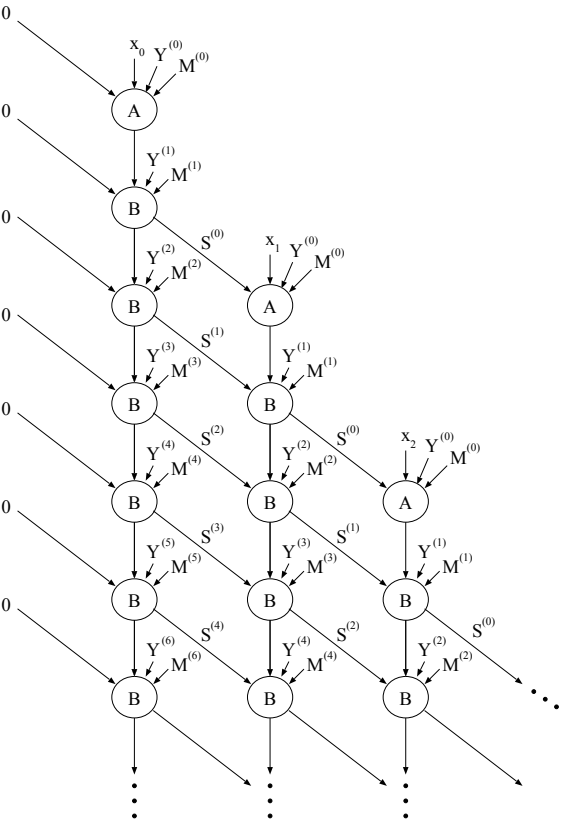

Fig. 1. The dependency graph for the MWR2MM Algorithm.

and a column for x_{i+1}. The total execution time for the computation shown in Figure 2 is 14 clock cycles.

Tasks A and B are performed on the same hardware module. The local control circuit of the module must be able to read the least significant bit of $S^{(0)}$ at the beginning of the operation, and keep this value for the entire operand scanning. Recall that the *even* condition of $S_0^{(0)}$ determines if the processing unit should add M to the partial sum during the pipeline cycle. The *pipeline cycle* is the sequence of steps that a PE needs to execute to process all words of the input operands.

The maximum degree of parallelism that can be attained with this organization is found as

$$p_{max} = \left\lceil \frac{e+1}{2} \right\rceil \ . \tag{2}$$

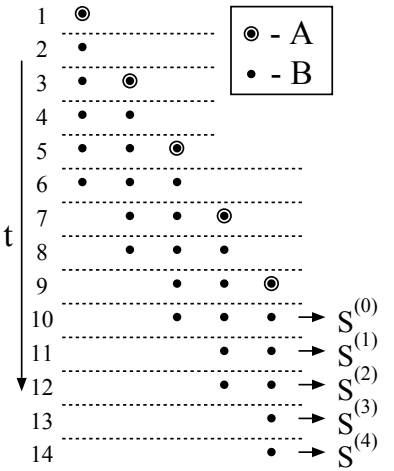

Fig. 2. An example of computation for 5-bit operands, where $w = 1$ bit.

It is easy to see from Figure 2 that $p_{max} = 3$. When less than p_{max} units are available, the total execution time will increase, but it is still possible to perform the full precision computation with the smaller circuit. Figure 3 shows what happens when only 2 processing modules are used for the same computation shown in Figure 2. In this case, the computation during the last pipeline cycle wastes one of the stages, because m is not a multiple of 2.

The total computation time T (in clock cycles) when $n \leq p_{max}$ modules (stages) are used in the pipeline is

$$T = \left\lceil \frac{m+1}{n} \right\rceil (e+1) - 1 + 2(n-1) \tag{3}$$

where the first term corresponds to the number of pipeline cycles ($\lceil (m+1)/n \rceil$) times the number of clock cycles required by a pipeline stage to compute one full-precision operand, and the last term corresponds to the latency of the pipeline architecture. With n units, the average utilization of each unit is found as

$$U = \frac{\text{Total number of time slots per bit of } X \times m}{\text{Total number of time slots } \times n} = \frac{m(e+1)}{Tn} . \tag{4}$$

Figure 4 shows the hardware utilization U, total computation time T, and speedup of 2 or 3 units over one unit, for a small range of the precision, and word size $w = 8$ bits. We can see that the overhead of the pipelined organization becomes insignificant for precision $m \geq 3w$. We can attain a speedup very close to the optimum for even small number of operand bits.

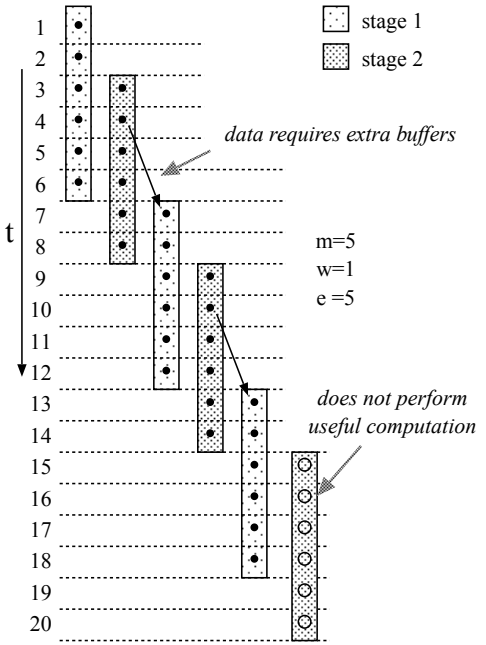

Fig. 3. An example of computation for 5-bit operands with two pipeline stages.

6 Design of the Scalable Architecture

A pipeline with 2 computational units is shown in Figure 5. One aspect of this organization is the register file design. Since the data is received word-serially by the kernel, the registers must work as rotators (circular shift registers) in some cases and shifters in other cases. The registers which store Y and M work as rotators. The processing elements itself must relay the received digits to the next unit in the pipeline. All paths are w bits wide, except for the x_i inputs (only 1 bit). The values of x_i come from a p-shift register, where p equals to the number of processing elements in the pipeline. The register for S must be a shift register, since its contents is not reused. The length (L) of the shift register for S values depends on the number of words (e) and the number of stages (n) in the pipeline. This length is determined as:

$$L = \begin{cases} e + 2 - 2n & \text{if } (e+2) > 2n \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Observe that these registers will not consume more than what is normally used in a conventional radix-2 design of the MM. These registers can be easily implemented by connecting one memory element to another in a chain (or loop), which will not impact the clock cycle of the whole system. Since we also need

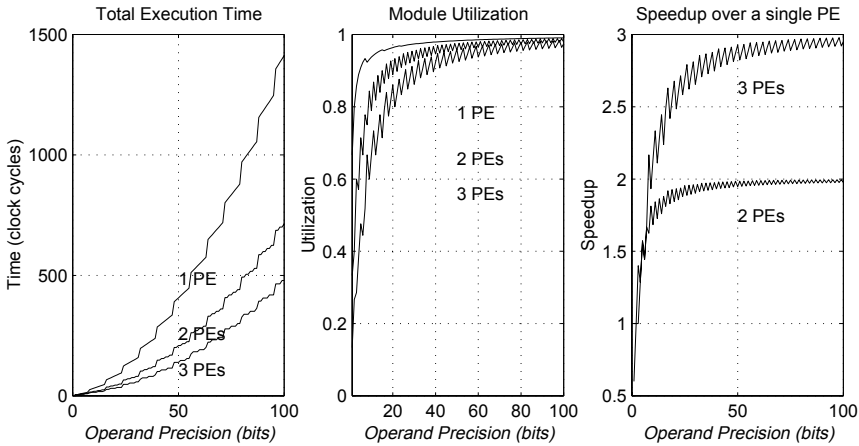

Fig. 4. The performance figures for multiple units with $w = 8$ bits.

loading capability for the rotator, multiplexers (MUXes) should be used between certain memory elements in the chain. The delay imposed by these MUXes will not create a critical path in the final circuit. To avoid having too many MUXes, we may load M and Y serially, during the last pipeline cycle of the algorithm. In this case, MUXes are required between two memory elements of the rotator only (not between all of the memory elements).

The global control block was not included in this figure for simplicity. Its function can be inferred from the dependency graph and the algorithm already presented. The shaded box represent flip flops.

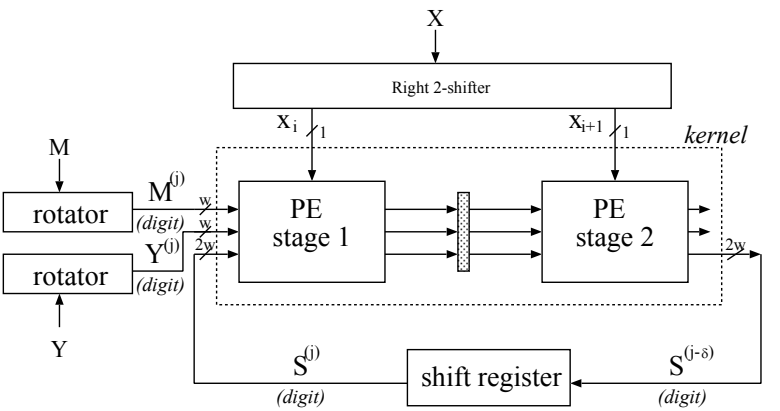

Fig. 5. Pipelined organization with 2 units.

6.1 Processing Element

The block diagram of the processing element is shown in Figure 6. The data path receives the inputs from the previous stage in the pipeline, and computes the next $S^{(j)}$ digits serially. The inputs are delayed one extra clock cycle before they are sent to the next stage.

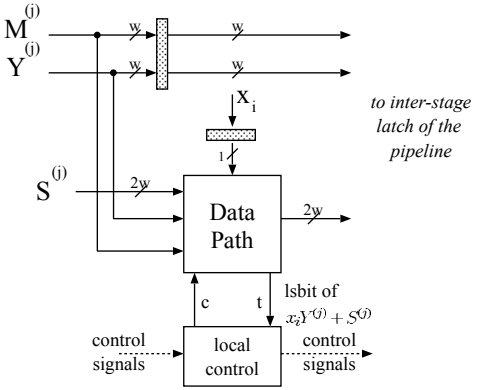

Fig. 6. The block diagram of the processing unit.

To reduce storage and arithmetic hardware complexity, we consider that M, X, and Y are available in non-redundant form. The internal sum S is received and generated in the redundant Carry-Save form. In this case, $2w$ bits per word are transferred between units in each clock cycle. The data path also makes available the information on the least significant bit (t) of the computation $S^{(j)} + x_i Y^{(j)}$ which is the first computation step performed by the data path in each pipeline cycle. Only the value t obtained when the least significant digits of Y and S come into the unit should be used to control the addition of M (control signal c). The local control is responsible for storing the t value during the pipeline cycle, and also relay some control signals to the downstrem modules.

The design of the data path follows the idea presented in [14] modified for least-significant-digit-first type of computation. The basic organization of the data path consists of two layers of carry-save adders (CSA). Assuming a full-precision structure as in Figure 7(a), we propose the retiming process shown for the case $w = 1$ to generate the serial circuit design presented in Figure 7(b). For $w > 1$, larger groups of adders are considered, based on the same approach. Notice that the cycle time may increase for larger w as a result of the broadcast problem only; it will not depend on the arithmetic operation itself. The high-fanout signals in the design are x_i and c, and both change value only once for each pipeline cycle. Observe that the bit-right-shift that must be performed by the data path is already included in the CSA structure shown in the Figure.

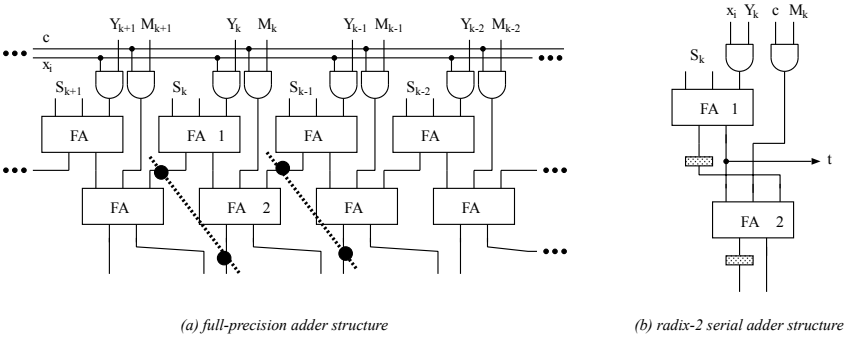

Fig. 7. The serial computation of the MM operations.

The data path design for the case $w = 3$ is shown in Figure 8. It has a more complicated shift and alignment section to generate the next S word. When computing the bits of word j (step j), the circuit generates $w - 1$ bits of $S^{(j)}$, and the most significant bit of $S^{(j-1)}$. The bits of $S^{(j-1)}$ computed at step $j-1$ must be delayed and concatenated with the most significant bit generated at step j (alignment).

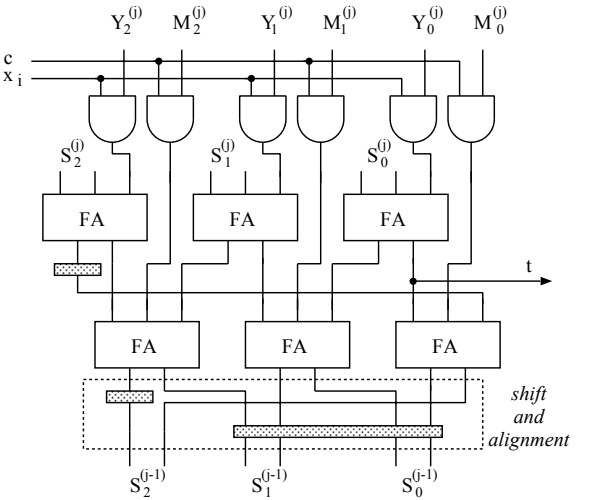

Fig. 8. PE's data path for $w = 3$ bits.

7 Area/Time Tradeoffs

After describing the general building block for the implementation of our scalable MM architecture, we discuss the area/time tradeoffs that arise for different values of operand precision m, word size w, and the pipeline organization. The area A is given as a design constraint. In this analysis, we do not consider the wiring area. For a first order approximation we consider that the propagation delay of the processing element is independent of w (this hypothesis is reasonable when w is small). This assumption implies that the clock cycle is the same for all cases and the comparison of speed among different designs can be made based on clock cycles. The area used by registers for the intermediate sum, operands and modulus is the same for all designs.

It is clear that the proposed scheme has the worst execution time for the case $w = m$, since some extra cycles were introduced by the computational unit in order to allow word-serial computation, when compared to other full-precision designs. Thus, we will consider the case when the available chip area is not sufficient to implement a full-precision conventional design. The performance evaluation resumes to the question:

What is the best organization for the scalable architecture for a given area?

We used VHDL on the Mentor graphics tools to synthesize the circuit with the $1.2\mu m$ CMOS technology. The cell area for a given word size w is obtained as

$$A_{cell}(w) = 47.2w \ ,$$

where the value 47.2 is the area cost provided by the tool (a 2-input NAND gate corresponds to 0.94). When using the pipelined organization, the area of each inter-stage latch is important, and was measured as $A_{latch}(w) = 8.32w$. The area of a pipeline with n units is given as

$$A_{pipe}(n, w) = (n-1)A_{latch}(w) + nA_{cell}(w) = 55.52nw - 8.32w \ . \qquad (6)$$

The maximum word size that can be used in the particular design (w_{max}) is a function of the available area A and the number of pipeline stages n. It is found as

$$\begin{aligned} A_{pipe}(n, w) &\leq A \\ 55.52nw - 8.32w &\leq A \\ w &\leq \frac{A}{55.52n - 8.32} \\ w_{max}(A, n) &= \left\lfloor \frac{A}{55.52n - 8.32} \right\rfloor \ . \end{aligned} \qquad (7)$$

Based on w_{max}, we obtain the total execution time (in clock cycles) for operands with precision m from Equation 3, as follows:

$$T(m, A, n) = \left\lceil \frac{m+1}{n} \right\rceil \left(\left\lceil \frac{m+1}{w_{max}(A, n)} \right\rceil + 1 \right) - 1 + 2(n-1) \ . \qquad (8)$$

For a given area A, we are able to try different organizations and select the faster one. The graph given in Figure 9 shows the computation time for various pipeline configurations for $A = 20,000$. The number of stages that provides the best performance varies with the precision required in the computation. For the cases shown, 5 stages would provide good performance. We don't want to have too many stages for two reasons: (1) high utilization of the processing elements will be possible only for very high precision and (2) the execution time may have undesirable oscillations (as shown in the rightmost part of the curve for $m = 1024$). The behavior mentioned in (2) is the result of **(i)** word size w is not a good divisor for m, producing one word (most significant) with few significant bits, and **(ii)** there is not a good match between the number of words e and n, causing a sub-utilization of stages in the pipeline.

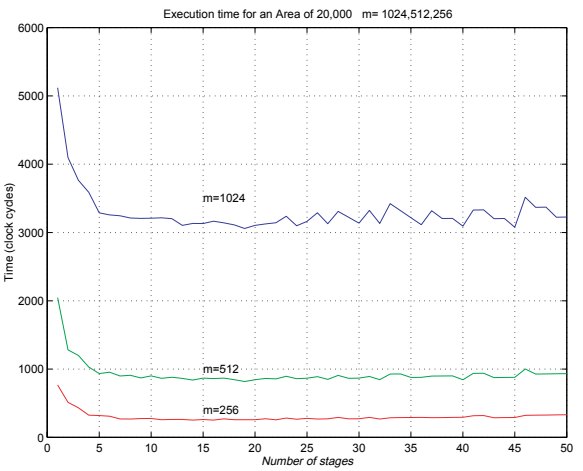

Fig. 9. The execution time of the MM hardware for various precision and configurations.

For a fixed area, the word size becomes a function of the number of stages only. The word size decreases as the number of stages in the pipeline increases. The word size for some values of n is given on Table 1.

Table 1. The number of pipeline stages versus the word size, for a fixed chip area.

n (stages)	1	2	3	4	5	6	7	8	9	10
w (bits)	423	194	126	93	74	61	52	45	40	36

From the synthesis tools we also obtained a minimum clock cycle time of 11 ns (clock frequency of 90MHz). For the case $m = 1024$ bits, $n = 10$ stages, and $w = 36$ bits, the total execution time is $3107 * 11 = 34,177$ nanoseconds. The correction step was not included in these estimates, but it would require another pipeline cycle to be performed.

8 Conclusions

We presented a new architecture for implementing the Montgomery multiplication. The fundamental difference of our design from other designs described in the literature is that it is scalable to any operand size, and it can be adjusted to any available chip area. The proposed architecture is highly flexible, and provides the investigation of several design tradeoffs involved in the computation of the Montgomery multiplication. Our analysis shows that a pipeline of several units is more adequate than a single unit working with a large word length. This is an interesting result since using more units we can reduce the word size and consequently the data paths in the final circuit, reducing the required bandwidth. The proposed data path for the multiplier was synthesized to a circuit that is able to work with clock frequencies up to 90MHz (for the CMOS technology considered in this work). The total time to compute the Montgomery multiplication for a given precision of the operands will depend on the available area and the chosen pipeline configuration. The upper limit on the precision of the operands is dictated by the memory available to store the operands and internal results.

Acknowledgements

This research is supported in part by Secured Information Technology, Inc. The authors would like to thank Erkay Savaş (Oregon State University) for his comments on the algorithm definition.

References

1. A. Bernal and A. Guyot. Design of a modular multiplier based on Montgomery's algorithm. In *13th Conference on Design of Circuits and Integrated Systems*, pages 680–685, Madrid, Spain, November 17–20 1998.
2. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.
3. S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, June 1993.
4. T. Hamano, N. Takagi, S. Yajima, and F. P Preparata. $O(n)$-Depth circuit algorithm for modular exponentiation. In S. Knowles and W. H. McAllister, editors, *Proceedings, 12th Symposium on Computer Arithmetic*, pages 188–192, Bath, England, July 19–21 1995. Los Alamitos, CA: IEEE Computer Society Press.

5. Ç. K. Koç and T. Acar. Fast software exponentiation in GF(2^k). In T. Lang, J.-M. Muller, and N. Takagi, editors, *Proceedings, 13th Symposium on Computer Arithmetic*, pages 225–231, Asilomar, California, July 6–9, 1997. Los Alamitos, CA: IEEE Computer Society Press.
6. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.
7. Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
8. P. Kornerup. High-radix modular multiplication for cryptosystems. In E. Swartzlander, Jr., M. J. Irwin, and G. Jullien, editors, *Proceedings, 11th Symposium on Computer Arithmetic*, pages 277–283, Windsor, Ontario, June 29 – July 2 1993. Los Alamitos, CA: IEEE Computer Society Press.
9. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Boston, MA: Kluwer Academic Publishers, 1993.
10. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
11. H. Orup. Simplifying quotient determination in high-radix modular multiplication. In S. Knowles and W. H. McAllister, editors, *Proceedings, 12th Symposium on Computer Arithmetic*, pages 193–199, Bath, England, July 19–21 1995. Los Alamitos, CA: IEEE Computer Society Press.
12. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
13. A. Royo, J. Moran, and J. C. Lopez. Design and implementation of a coprocessor for cryptography applications. In *European Design and Test Conference*, pages 213–217, Paris, France, March 17-20 1997.
14. A. F. Tenca. *Variable Long-Precision Arithmetic (VLPA) for Reconfigurable Coprocessor Architectures*. PhD thesis, Department of Computer Science, University of California at Los Angeles, March 1998.
15. C. D. Walter. Space/Time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Transactions on Computers*, 46(2):139–141, February 1997.