High-Radix Design of a Scalable Modular Multiplier*,**

Alexandre F. Tenca, Georgi Todorov, and Çetin K. Koç

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

Abstract. This paper describes an algorithm and architecture based on an extension of a scalable radix-2 architecture proposed in a previous work. The algorithm is proven to be correct and the hardware design is discussed in detail. Experimental results are shown to compare a radix-8 implementation with a radix-2 design. The scalable Montgomery multiplier is adjustable to constrained areas yet being able to work on any given precision of the operands. Similar to some systolic implementations, this design avoid the high load on signals that broadcast to several components, making the delay independent of operand's precision.

Key Words: modular multiplier, montgomery multiplier, scalable architecture, high-radix.

1 Introduction

Several applications, such as RSA algorithm, [14] Diffie-Hellman key exchange algorithm [5], Digital Signature Standard [12], and Elliptic curve cryptography [6,9] use modular multiplication and modular exponentiation. The Montgomery Multiplication (MM) algorithm [10] provides certain advantages in the implementation of modular multiplication. Multiple software and hardware designs have been developed using the algorithm.

An aspect of cryptographic applications is that very large numbers are used. The precision varies from 128 and 256 bits for elliptic curve cryptography to 1024 and 2048 bits for applications based on exponentiation [15]. Most of the hardware designs for modular multiplication are fixed-precision solutions. That is, the operands cannot exceed a fixed bit-size. Designs that can take operands with an arbitrary precision are researched in the ASIC [18] and the FPGA [2] realms.

It is recognized that designing hardware requires making the area-time tradeoff [21]. In the general case "faster means better". However, an application where this rule is not valid can always be found. Therefore, it is important that the designers have several options or choices that they can choose from.

* This research was supported by rTrust Technologies.

** The reader should note that Oregon State University has filed US and International patent applications for inventions described in this paper.

Ç.K. Koç, D. Naccache, and C. Paar (Eds.): CHES 2001, LNCS 2162, pp. 185–201, 2001.

The basic idea of the scalable Montgomery multiplier has been presented in [18]. The main features of this multiplier are (1) the ability to work on any given operand precision at the kernel level,(2) be adjustable to any chip area, a (3) use a pipelined organization that reduces the impact on signal loads as a result of high precision of the operands.

The first feature is unique in comparison to other designs. The ability to handle long-precision numbers with small precision operations has been done using conventional multipliers, and a control algorithm that uses these multipliers [7]. The general approach is to reuse a hardware core with a fixed precision, usually at most 32 or 64 bits. The current publications show conventional multipliers that do not exceed a precision of 100 bits [16,1]. The control algorithm is usually complex in this case and the increase in parallelism involve multiple datapaths and high complexity at the system level. Other solutions that use systolic array implementation are designed for a fixed precision and the implementation must be modified if a precision larger than the one originally considered is required.

The second feature comes from the flexibility of the algorithm and hardware to be adjusted in both word size and number of processing elements. The more hardware is available, the better is the performance of the multiplier. Similar adjustment is also possible on algorithms based on conventional multipliers, at the cost already presented above. Beyond any doubt, cryptographic algorithms will be embedded in almost any application involving exchanging of information. Applications, such as smart cards [11] and hand-held devices require hardware designs restricted on area and power resources.

The high load on signals broadcast to several hardware components is an important factor to slow down high-precision Montgomery multiplier (MM) designs. For this reason, the use of systolic structures have been considered by other researchers. The organization presented in this paper is not purely systolic, and has a flavor of serial-parallel implementation of the multiplication algorithm.

In this work we present an evolution of the radix-2 algorithm proposed in previous papers, which lead us to a higher radix design of the system. This paper describes the issues involved in this design and the experimental results to compare with the former radix-2 design.

2 High-Radix Word-Based Montgomery Algorithm

The notation used throughout this text is shown in Table 1.

Figure 1 shows the Multiple-word High-Radix (2^k) Montgomery Multiplication algorithm (MWR2^kMM), a generalization of the MM algorithm presented in [18]. A full-precision High-Radix Montgomery algorithm has been presented and proven to be correct in [8]. To prove correctness of the algorithm in Figure 1 we show that it is equivalent to the one presented in [8].

The parameter k changes depending on how many bits of the multiplier X are scanned during each loop, or 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 [3]. Booth encoding is

Table 1. Notation

- M - modulus for modular multiplication;
- 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;
- q_{Y_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;
- q_{M_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;
- CS - carry-save;
- 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.

applied to a bit vector to reduce the complexity of multiple generation in the hardware. For radix-8 the Booth function for each digit is given as:

$$Booth(\underline{X_i}, x_{i-1}) = -4x_{i+2} + 2x_{i+1} + x_i + x_{i-1}$$

where $\underline{X_i} = (x_{i+2}, x_{i+1}, x_i)$ is a radix-8 digit ($i = km$ where m is an integer), $x_j \in \{0, 1\}$, and x_{i-1} is the most significant bit (MSbit) of the previous digit.

For Radix-2 computation $k = 1$ and $q_{Y_j} = x_j$ are used, making the algorithm equivalent to the one presented in [18]. C_a and C_b 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, $q_{M_j}M$ is added to the partial product. This is required to avoid losing bits in the shift operation performed in Step 10. The value of q_{M_j} that satisfies this condition is determined by examining the least significant k-bits of S generated at Step 4.

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.

The MWR2^kMM is a multiple-word version of a full-precision algorithm presented in Figure 2, which is called in this work R2^kMM algorithm. To obtain the R2^kMM algorithm we transform the word-based sequence of operations into full-precision operations. It is shown in [8] that the requirement for q_M is given as:

$$q_M \ * \ M = -S \ (mod \ 2^k).$$

Step
1: $S := 0$
$x_{-1} := 0$
2: FOR $j := 0$ TO $N - 1$ STEP k
3: $q_{Y_j} = Booth(x_{j+k..j-1})$
4: $(C_a, S^{(0)}) := S^{(0)} + (q_{Y_j} * Y)^{(0)}$
5: $q_{M_j} := S^{(0)}_{k-1..0} * (2^k - M^{(0)^{-1}}_{k-1..0}) \ mod \ 2^k$
6: $(C_b, S^{(0)}) := S^{(0)} + (q_{M_j} * M)^{(0)}$
7: FOR $i := 1$ TO $NW - 1$
8: $(C_a, S^{(i)}) := C_a + S^{(i)} + (q_{Y_j} * Y)^{(i)}$
9: $(C_b, S^{(i)}) := C_b + S^{(i)} + (q_{M_j} * M)^{(i)}$
10: $S^{(i-1)} := (S^{(i)}_{k-1..0}, S^{(i-1)}_{BPW-1..k})$
END FOR;
11: $C_a := C_a \ or \ C_b$
12: $S^{(NW-1)} :=$ sign ext $(C_a, S^{(NW-1)}_{BPW-1..k})$
END FOR;
13: IF $S \geq M$ THEN $S := S - M$
END IF;

Fig. 1. Multiple-word High-Radix *(Radix-2^k)* Montgomery Multiplication (MWR2^kMM) Algorithm.

This requirement can be also rewritten as

$$S_{k-1..0} + q_M * M_{k-1..0} = 0 \ mod \ 2^k.$$

The latter equation is another representation of the requirement that the last k bits of S must be zeros. The Step 5 is equivalent to this requirement as shown below:

$$q_{M_j} = S_{k-1..0} * (2^k - M^{-1}_{k-1..0}) \bmod 2^k$$
$$q_{M_j} = S_{k-1..0} * (-M^{-1}_{k-1..0}) \bmod 2^k$$
$$S_{k-1..0} = S \bmod 2^k, M_{k-1..0} = M \bmod 2^k$$

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

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

$$q_{M_j} = S * (-M^{-1}) \bmod 2^k$$
$$q_{M_j} * M = -S \bmod 2^k$$

It is also easy to show that

$$Y = \sum_{j=0}^{\lceil \frac{N-1}{k} \rceil} (2^k)^j * q_{Y_j},$$

from the Booth encoding properties.

The last two equations show that the coefficients q_{Y_j} and q_{M_j} are determined the same way as in [8], which makes both algorithms equivalent. In [8] there are requirements for X and Y that determine the boundaries for the result S. There are no such requirements in the R2^{k}MM algorithm. The R2^{k}MM algorithm inherits the boundaries for the result from the original MM algorithm.

3 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 [18]. 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 $(q_M * M)^{(*)}$ shown in the MWR2^{k}MM 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 q_M in the most critical part of the algorithm as it is also pointed out by other authors [13,20].

The architecture of a Montgomery multiplier implementing the MWR2^{k}MM algorithm is shown in Fig. 3. There are two main functional blocks: *Kernel* and *IO*. Only the data path is shown. The Kernel's datapath is where the computation takes place according to the algorithm. A control block (not shown) supplies the signals to synchronize the system.

The *final reduction* functional block computes the final result in a suitable form for the multiplier's output, implementing step 13 of the algorithm. More details are provided later.

The Kernel's datapath 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

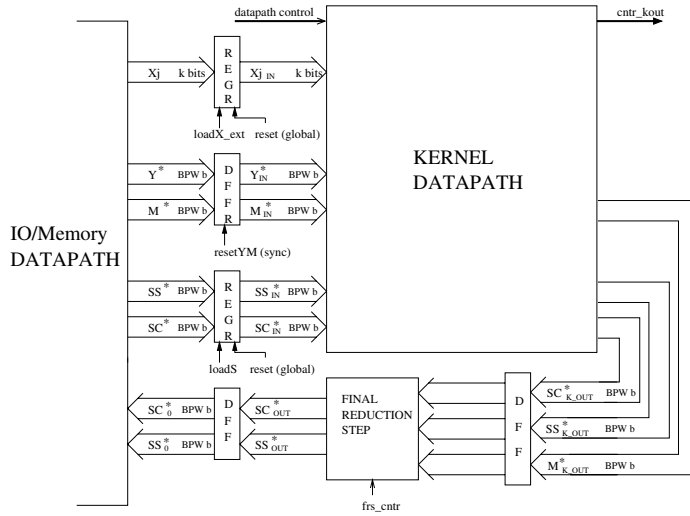

Fig. 3. System Level Diagram of Modular Multiplier.

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 [18].

The signal X_j is a k-bit signal. It provides the bits of X needed for Step 3 of the MWR2^{k}MM 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 system's 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. A detailed description of the signal's timing in the interface between I/O and kernel is presented in [19].

4 Kernel Datapath and Reduction

The kernel datapath is organized as a pipeline of cells (MMcell) separated by registers (Fig.4). A stage consists of a MMcell and a register. The MMcell implements one iteration of the FOR loop (steps 3 to 12) in the MWR2^{k}MM algorithm. Each stage gets as inputs one word of Y, M, SS and SC each clock cycle. Additionally, $(NS * k)$ bits of X are transferred to the kernel over $2 * NS$ clock periods, where NS corresponds to the number of stages. Depending on the computation's progress, k bits of X are loaded in a different stage every 2 clock cycles. Each stage needs these bits at different times. Thus, this signal is made common for all stages with internal control loading the signal in the right stage

at the right time. The MS bit of X_i is used to Booth encode X_{i+1}, as explained in Section 2, thus, a cell must store these two pieces of information in order to properly encode a radix-r digit of X. The datapath outputs one word of each SS and SC every clock cycle. The pipeline outputs are $SS^{(*)}_{K\_OUT}$ and $SC^{(*)}_{K\_OUT}$.

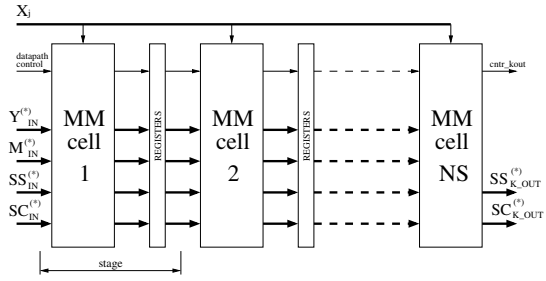

Fig. 4. Top Level Diagram of the Kernel datapath.

Each MMcell propagates the words of Y and M and the newly computed words of SS and SC to the next MMcell, which performs another computational loop of the Montgomery Multiplication algorithm and on its turn propagates the words of Y and M and the newly computed words of SS and SC, with a latency of 2 cycles.

The reduction block implements the final reduction step in the MWR2^{k}MM algorithm. The final reduction happens after the last iteration of the loop scanning the bits of X. During the intermediate iterations the *final reduction* block propagates the signals from the kernel datapath without operating on them. However, the design takes advantage of the word-serial output of the kernel datapath and implements the final reduction serially, on-the-fly, as the words of both vectors of the result are coming out of the kernel datapath. The condition $S \geq M$ will not be known before the last pair of words for S is computed in the datapath. The final reduction block implements the computation for both conditions, $S \geq M$, when $S - M$ is generated, and $S < M$, when the result is correct. In both cases the Carry-Save to non-redundant conversion is required. Both resulting vectors will be stored in the place for SS and SC (the two bit-vectors of the intermediate result) in the IO block. After the last pair of words of S is processed, a flag is set by the control circuitry indicating which condition is valid, $S \geq M$ or $S < M$. The result will be in either SS or SC. A detailed implementation of the final reduction block is presented in [19].

5 Kernel Implementation

The direct design of the kernel processing element leads to an organization shown in Figure 5(a). The figure shows the main blocks in the design: booth encoding,

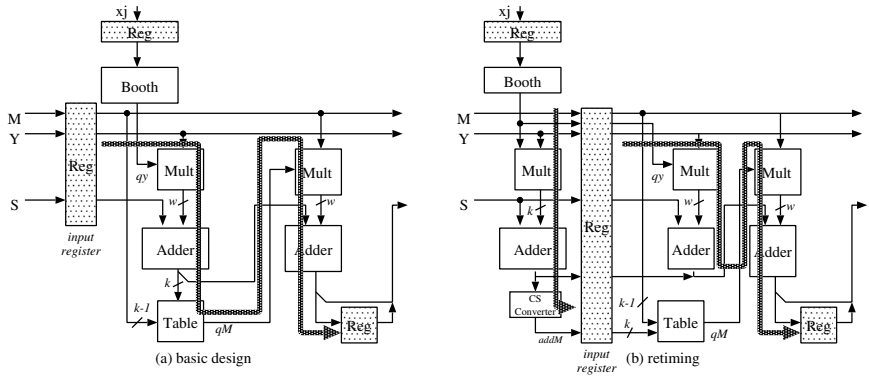

Fig. 5. Kernel cell organization: (a) first try, and (b) after re-timing.

multiple generation, adders, and registers (shaded boxes). Shifting and alignment is done by proper combination of signals.

The cell operates on $k+1$ bits of the multiplier X (one bit is obtained from the previous scan) and one word of each the multiplicand (Y), the modulus (M) and the partial product (S). Booth encoding is generated by a lookup table to find the coefficient q_{Y_j}. The negative multiples of Y are implemented by complementing their positive counter-pairs and adding a '1' (two's complement sign change). The coefficient q_{M_j} depends on the last k bits of the partial product S and the last $k-1$ bits of the modulus M (Step 5). Recall that M is odd. Before S is shifted to the right, the value $q_{M_j}M$ is added to S (Steps 6 and 9). The coefficient q_{M_j} is in the range $[0, 2^k - 1]$. For radix-8, the greatest value happens when $S^{(0)}_{2..0}$ = "001", and $M^{(0)}_{2..0}$ = "001" ($q_{M_j} = 7$). The lowest value happens when $S^{(0)}_{2..0}$ = "000", and $M^{(0)}_{2..0}$ = "001" ($q_{M_j} = 0$).

Multiple generation for high-radix designs is expensive because q_Y and q_M may assume values that are not powers of 2. As an example, the bit-vector $2Y$ can be produced from Y by left-shifting Y by one bit. However, the bit-vector $3Y$ is produced by adding Y and $2Y$.

The critical path in the basic design is very long and makes the design of such high-radix circuit less attractive. The high radix is going to increase the table delay and size, and the multiple generation delay and size. To increase the performance of this system, re-timing was applied, resulting in the design shown in Figure 5(b).

5.1 Improving the Performance Using Re-timing

Using re-timing, pieces of combinational logic are relocated to other other parts of a sequential system, modifying the critical path. One problem with the first direct implementation of the high-radix algorithm is the long critical path, passing through several modules, as shown in Figure 5(a). One can observe that the

determination of q_{M_j} depends on k LSBits of the partial product from the previous computational cycle, $S^{(0)}_{k-1..0}$, the k LSBits of $Y^{(0)}$, and the coefficients q_{Y_j}. If the word size for S is more than $2k$ bits the k LSBits of S for the next pipeline stage will be available well before the whole word $S^{(0)}$ is available. The idea is to advance the information on the k least-significant bits (LSBits) of the shifted $S^{(0)}$. In the previous design, these bits were propagated between two registers with no logic operation done on them. Instead of simply propagating the bits, the logic determining q_M is performed on them, as shown in Figure 5(b).

The difference between these cell designs is that a portion of the first adder was moved to before the input registers, and this portion of the adder computes only the k LSBits of the not yet shifted partial product, which is required to compute q_M. The k-bit vector *addM* in the Figure represents these bits in non-redundant form, and is applied to the Table that generates q_M in the next clock cycle, considering also $k-1$ bits of the modulus M. As a result of this hardware organization, all possible path delays will not exceed the delay of two adders and two MUXes.

The computation done on the LSBits by the leftmost is also done for all the other remaining operand words. So, while the leftmost adder works on the LS bits of a word, the topmost adder (after the input register) should be working on the other bits of the same word. There is one clock cycle difference between the two circuits, and therefore, this situation must be considered carefully.

5.2 A Radix-8 Design

Without loss of generality, the details of this design will be explained based on a radix-8 implementation. The circuit in Fig. 6 shows the diagram for a Radix-8 MMcell.

One way of implementing the coefficients q_Y and q_M is to split them into some components that will generate simple multiples and add these multiples in the adder. For $r = 8$, two values could be used. For example, $q_Y = 3$ would be split into 2 and 1, and the $3 * Y$ multiple would be generated as $2 * Y + 1 * Y$ or $4 * Y - 1 * Y$ without actually performing the addition or subtraction but using two bit-vectors, $2 * Y$ and $1 * Y$ or $4 * Y$ and $-1 * Y$ in this example. It is efficient to choose only one of the components as a negative value. This is true because negative bit-vectors, like $-Y$, are implemented by inverting the positive bit-vector, Y in this case, and introducing a carry-in with a value of '1'. Since each four-to-two adder has only one carry-in input, only one of the components can be negative.

Two multiplexers generate the multiples $(q1_{Y_j} * Y)^{(*)}$ and $(q2_{Y_j} * Y)^{(*)}$. The Booth encoding is done according to Table 2 in *DEC_XJ* functional block. As an example, $(/2 * Y)$ means that the Y is multiplied by 2 and all the bits are complemented (or negated). Also, one can notice that the values 2 and -2 are formed in two different ways. This approach simplifies the decoding logic for X_j. The outputs of *DEC_XJ* are the control signals for the multiplexers as well as the carry-in bit for the first 4-to-2 adder (during the first computational cycle only).

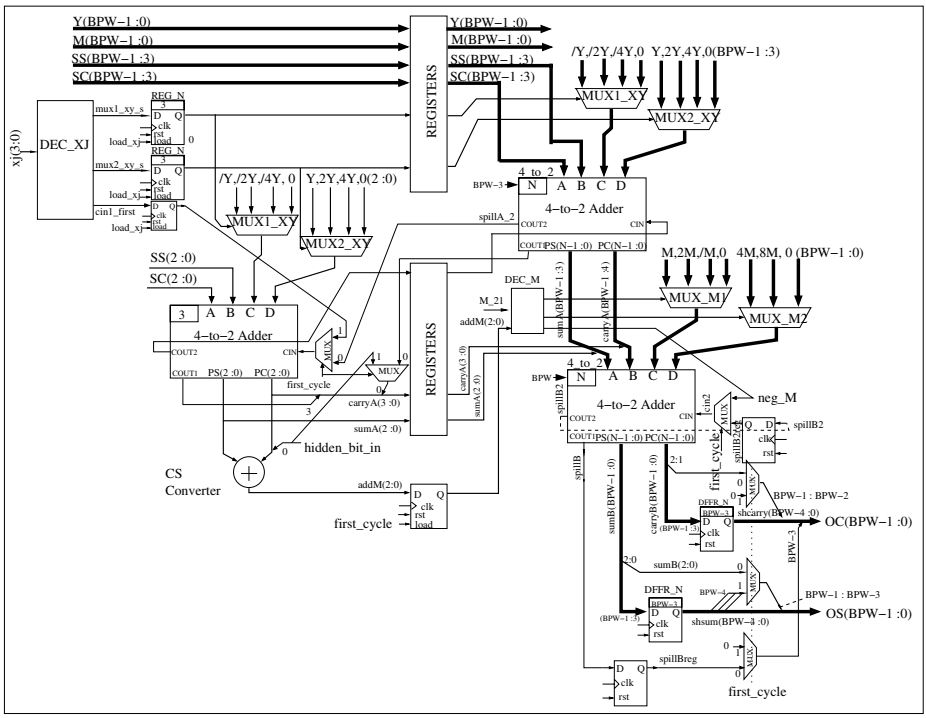

Fig. 6. Radix-8 MM Cell.

Because the coefficients q_Y and q_M are split into two each, the adders need to have an extra input. The two four-to-two adders have a total of two carry-out bits propagating between sequential words of the partial product S. One carry-out is inserted at the LSB position of vector $carryA$. The other carry-out is introduced back to the same adder as a carry-in bit for the next word of S.

The coefficient q_{M_j} depends on the 3 LSBits of the partial product S and the three LSBits of the modulus M. The product is represented by 2 vectors. There is one additional input bit, *hidden-bit*, which affects q_{M_j}. The hidden-bit is generated by carry propagation in the least significant bits of the least significant word computation, which are zeroed in the process. Knowing that the LSB of M is always '1' and the LSB of $carryA$ is always '0', q_{M_j} will depend only on eight bits: $sumA_{2..0}$, $carryA_{2..1}$, *hidden-bit* and $M^{(0)}_{2..1}$.

In Step 10 of the MWR8MM algorithm the partial product is right-shifted by three bits. Because carry-save representation (CS) is used for S, the LS words of the two bit-vectors $(sumB^{(0)}, carryB^{(0)})$ after Step 6 in the algorithm can be, for example: $sumB^{(0)} = \times..\times 110$ and $carryB^{(0)} = \times..\times 010$, where $\times$ represents any value of the bit in this position. The last three bits of S are equivalent to zeros when converted to a non-redundant form. However, data will be lost if these bits are shifted out without taking into account the carry propagation $(110 + 010 = \underline{1}000)$. The carry bit generated in this case is the *hidden-bit*.

Table 2. Booth encoding for q_Y, the backslash means bit-complement.

$X_j(3:0)$	q_{Y_j}	$q1_{Y_j}$	cin	$q2_{Y_j}$	$X_j(3:0)$	q_{Y_j}	$q1_{Y_j}$	cin	$q2_{Y_j}$
0000	0	0	0	0	1000	-4	/4	1	0
0001	1	0	0	1	1001	-3	/4	1	1
0010	1	0	0	1	1010	-3	/4	1	1
0011	2	0	0	2	1011	-2	/4	1	2
0100	2	/2	1	4	1100	-2	/2	1	0
0101	3	/1	1	4	1101	-1	/1	1	0
0110	3	/1	1	4	1110	-1	/1	1	0
0111	4	0	0	4	1111	0	0	0	0

Instead of using a carry propagate adder to obtain the hidden-bit, in radix 8 the following observation is made: the last bit of $carryB^{(0)}$ is always '0', therefore, to detect a hidden-bit it is enough to test if there is a 1 value in the second or third bits of either $carryB^{(0)}$ or $sumB^{(0)}$. The circuit for the hidden-bit detection is reduced to $sumB_2^{(0)} + sumB_1^{(0)}$. These two bits of $sumB^{(0)}$ are stored into flip-flops, thus, the hidden-bit logic does not stand in the critical path for the whole cell. Since the hidden bit is found after the operation on the LS word is done, it is transferred from one cell to another, as part of the LS word. It can be inserted in the free LSBit position in $carryA^{(0)}$ and also participates in determining q_M.

If all eight bits are used for a lookup table for q_M, the table will have 256 entries. The number of entries can be reduced by assimilating the carries for $sumA_{2..0}$, $carryA_{2..1}$, and *hidden-bit* by a three-bit adder. The resulting three-bit vector is named $addM$:

$$addM_{2..0} = (sumA_{2..0} + (carryA_{2..1}, 0) + (00, hiddenbit)) \ mod\ 8.$$

which reduces the table for q_M to only 32 entries. It is represented by the *DEC_M* functional block according to Table 3. The decoder outputs are the control signals for the multiplexers implementing $(q1_{M_j} * M)^{(*)}$ and $(q2_{M_j} * M)^{(*)}$. The decoder also has an output which is asserted '1' whenever $q1_{M_j}$ is negative. This signal becomes a carry-in for the second four-to-two adder.

The multiples of Y and M, like $2Y, 4Y, 2M, 4M, 8M$, require that these operands be left-shifted. Caused by the word-serial scanning of this algorithm, this shifting requires some of the MSBits from the previous words of Y and M to be kept when the new words arrive. If it is the first word (*first_cycle*='1') then a number of zeros is shifted in to produce the needed multiple. Otherwise, the MSBits of the previous word are shifted in as the LSBits of the current word.

As described at the end of the previous section, the leftmost adder is operating on the LSbits of words j of S and $q_Y Y$ while the topmost adder is operating of the MSbits of word $j-1$. This arrangement requires that the carry-out propagation among words of the partial sum A ($carryA$ and $sumA$) be considered carefully. The carry-out of the topmost adder, net *spillA2*, is introduced immediately as carry-in for the leftmost adder. The carry-out of the leftmost adder is delayed one clock cycle before it is introduced as carry-in to the topmost adder.

Table 3. Decoding for q_M.

	q_{M_j}				$q1_{M_j}$				$cin2$				$q2_{M_j}$			
$addM_{2..0}$	$M^{(0)}_{2..1}$				$M^{(0)}_{2..1}$				$M^{(0)}_{2..1}$				$M^{(0)}_{2..1}$			
	00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
001	7	5	3	1	/1	1	/1	1	1	0	1	0	8	4	4	0
010	6	2	6	2	2	2	2	2	0	0	0	0	4	0	4	0
011	5	7	1	3	1	/1	1	/1	0	1	0	1	4	8	0	4
100	4	4	4	4	0	0	0	0	0	0	0	0	4	4	4	4
101	3	1	7	5	/1	1	/1	1	1	0	1	0	4	0	8	4
110	2	6	2	6	2	2	2	2	0	0	0	0	0	4	0	4
111	1	3	5	7	1	/1	1	/1	0	1	0	1	0	4	4	8

6 Experimental Results and Analysis

This section describes the experimental data obtained with the radix-8 Kernel designs and compares them with the radix-2 design. Although both radix-8 designs were implemented, only the results for the re-timed radix-8 design is presented in detail. The complete data is presented in [19].

6.1 Synthesis and Simulation Environment

The Mentor Graphics' package of applications was used to generate this data. The target technology was set to AMI05_slow ($0.5\mu m$) provided in the ASIC Design Kit (ADK) from the same company. A data-book for this technology is available at [4]. Before the designs were synthesized, they were simulated in ModelSim for functional correctness. The designs were described in VHDL, synthesized with Leonardo as flattened designs (no hierarchy), and laid-out using ICStation. This last tool provides RC parameter extraction. RC-extraction allows the determination of time delay values for each wire in the design, bringing further simulations closer to the real-silicon simulations. Using the information from ICStation and Leonardo, the designs were back annotated and verified with Velocity. The values presented in this section were obtained from several experiments.

The kernel area depends on the number of stages in the pipeline (NS) and the word size (BPW). The area for the radix-8 kernel was obtained as:

$$A_{kernel_{r8}} = 92 * BPW * NS + 269 * NS - 9.42 * BPW - 35.5.$$

The total computational time for the kernel is a product of the number of clock cycles (T_{CLKs}) and the clock period (t_p). The clock period is derived from the synthesis results, and will depend on the number of stages, the word size, and other parameters. The number of clock periods to complete a computation is obtained from the algorithm.

Table 4 shows the critical path delay (t_p) as a function of the number of stages for the re-timed radix-8 kernel as well as the number of bits per word in

the operands. These two parameters also determine the design area. The bold-faced figures in the Table show tested configurations. The rest of the figures are produced by linear interpolation. An increase in area leads to an increase in the critical path delay. This is due to increased wire lengths (parasitic resistance and capacitance) and fan-outs for the gates. A setup time plus clock-to-Q propagation time of 1.2ns for flip-flops is given for AMI05-slow technology. The hold time requirement is insignificantly small. The setup and hold time requirements will scale with the technology giving the same proportional effect on the clock period.

Table 4. Critical path delay for radix-8 Kernel (nsec).

	Bits Per Word						Bits Per Word				
NS	8	16	32	64	128	NS	8	16	32	64	128
1	**10.7**	**10.3**	**13.1**	**18.9**	**20.2**	10	11.2	15.2			
2	10.8	**12.1**	14.4	**20.5**	**30.4**	11	11.2	15.3			
3	10.9	12.5	15.7	23.0		12	11.2	**15.4**			
4	11.0	**12.9**	**17.0**	**25.4**		13	**11.3**	15.4			
5	**11.1**	**12.7**	17.6			14	11.3	15.4			
6	11.1	13.5	18.2			15	11.3	**15.5**			
7	**11.2**	**14.3**	18.7			20	**11.4**				
8	11.2	**14.9**	**19.2**			26	**13.0**				
9	11.2	15.1									

Two cases should be considered: (1) when $NW \leq 2 * NS$, and (2) when $NW > 2 * NS$. The variable $NW = \lceil \frac{N+1}{BPW} \rceil$ represents the number of words in the N-bit operands with chosen word size of BPW bits [18]. Because of the extra register in the pipeline a word propagates through the pipeline for $(2 * NS + 1)$ clock cycles For Radix-8, since 3 bits of X are used in each stage, $\lceil \frac{N}{3*NS} \rceil$ pipeline cycles are required. Equation 1 represents the total number of clock cycles needed for the re-timed Radix-8 Montgomery multiplication design as:

$$T_{CLKs} = \begin{cases} \lceil \frac{N}{3*NS} \rceil * (2 * NS + 1) + NW + 1 \,, if\ NW \leq 2 * NS \\ \lceil \frac{N}{3*NS} \rceil * (NW + 1) + 2 * NS \quad \,, if\ NW > 2 * NS \end{cases} \tag{1}$$

It can be shown that when $NW < 2 * NS$ adding more stages to the pipeline has somewhat unpredictable effect on the total number of clock cycles. It happens because in this case the number of words NW has a small effect on the computational time, while the fraction $\lceil \frac{N}{3*NS} \rceil$ has minimums and maximums as the number of stages NS changes. Thus, it may be the case that a design with more stages will be slower than a design with less stages.

Figure 7 shows the total actual computational time ($T_{CLKs} \times t_p$) for $N = 256$ and $N = 1024$, using designs with different number of stages (NS) and word size (BPW). The first observable minimum computational time happens when the boundary $NW \leq 2 * NS$ and $NW > 2 * NS$ is crossed. With further increase in the number of pipeline stages the computational time goes through a series of

minimal and maximal values. The boundary $NW > 2NS$ is crossed at a different number of stages for a different precision of the operands (a different number of words). Operands with precision 256 bits will require a smaller number of stages in the pipeline than operands with 1024 bits precision, in order to execute the operation in minimal time. The goal of choosing a design point is to have computational time for 256-bit precision close to its absolute minimal value and at the same time to have as small computational time for 1024-bit precision as possible.

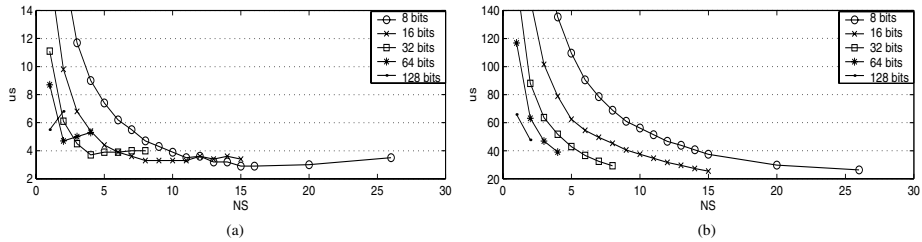

Fig. 7. Total time for 256-bit (a) and 1024-bit (b) operands for some values of NS and BPW.

It can be seen from the data obtained in the experiments that the fastest designs are achieved with a word size of 8 bits. For this word size and 256-bit precision, the first optimal design point is for $NS = 15$. The area is 14964 NOR gates. Each additional stage adds about 1005 to the gate count as can be obtained from the area equation. Other optimal points for this design, represented as NS/area pairs, are: 16/15969, 18/17979, 22/21999, 24/24009 and 26/26019.

For 1024-bits of precision, the time decreases asymptotically, with a faster decrease for a smaller number of stages.

Table 5. Some design points for radix-8 kernel, $BPW = 8$, $N = 256$ and $N = 1024$.

NS	15	16	18	22	24	26
Area, gates	14964	15969	17979	21999	24009	26019
$\frac{t_{NS=15}}{t}$ for 256-bit	1	1	1.05	1.04	0.92	0.83
$\frac{t_{NS=15}}{t}$ for 1024-bit	1	1.04	1.21	1.37	1.39	1.42

Table 5 compares several design points for the radix-8 kernel with $BPW = 8$. The Table presents the design area and the ratio of the computational time related to the point $NS = 15$. It can be seen that the design point with $NS = 22$ is very suitable since the computational time for 256-bit precision is very close to its minimal value. At the same time the computational time for 1024-bit precision is improved by 37% as compared to the point with $NS = 15$. With

further increase of the number of stages the computational time for 256-bit precision worsens while the computational time for 1024-bit precision does not improve significantly (only 2% per stage).

A comparison of performance between the radix-2 design ([18]) and the radix-8 designs discussed in this paper is shown in Figure 8. The data shows the time to compute the modular multiplication for 256-bit operands as a function of the design area. For small areas, the radix-2 design ($v1$) performs as well as the radix-8 design with re-timing ($v3$). The basic design ($v2$) is worse than the radix-2 one. For areas of 10,000 gates or more, the radix-8 design with re-timing is better than the other two, which shows that the high-radix design has a better overall performance.

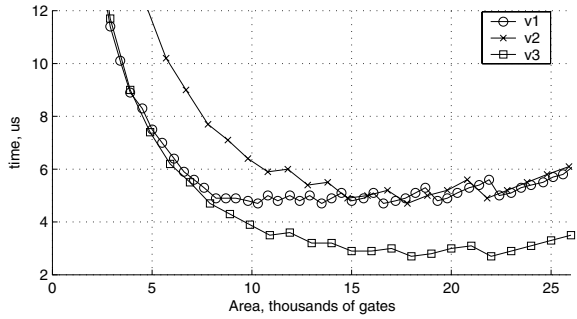

Fig. 8. Area×time comparison between radix-2 (v1), radix-8 basic (v2), and radix-8 with re-timing (v3) for 256-bit operands.

7 Conclusion

This paper presented the algorithm modifications and hardware implementation details of a high-radix implementation of the scalable modular multiplier presented in [18]. A radix-8 design was used to exemplify the design process, and to obtain experimental results that show the viability of using this approach. Experimental data shows that the radix-8 scalable multiplier is able to perform as well as the radix 2 design for small areas, and better than the radix-2 design for larger areas. The re-timing technique applied to the high-radix design was critical to obtain a competitive solution.

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. T. Blum and C. Paar. Montgomery modular exponentiation on reconfigurable hardware. In I. Koren and P. Kornerup, editors, *Proceedings, 14th Symposium on Computer Arithmetic*, pages 70–77, Bath, England, April 14–16 1999. IEEE Computer Society Press, Los Alamitos, CA.
3. A. D. Booth. A signed binary multiplication technique. *Q. J. Mech. Appl. Math.*, 4(2):236–240, 1951. (Also reprinted in [17], pp. 100–104).
4. Mentor Graphics Corporation. ASIC Design Kit. `http://www.mentor.com/partners/hep/AsicDesignKit/ASICindex.html`, 2001.
5. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.
6. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.
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. IEEE Computer Society Press, Los Alamitos, CA.
9. A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullen, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields.* Kluwer Academic Publishers, Boston, MA, 1993.
10. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
11. D. Naccache and D. M'Raïhi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, June 1996.
12. National Institute for Standards and Technology. Digital signature standard (DSS). *Federal Register*, 56:169, August 1991.
13. 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. IEEE Computer Society Press, Los Alamitos, CA.
14. 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.
15. E. Savaş, A. F. Tenca, and Ç. K. Koç. A scalable and unified multiplier architecture for finite fields $gf(p)$ and $gf(2^m)$. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000*, Lecture Notes in Computer Science No. 1965, pages 281–296. Springer, Berlin, Germany, 2000.
16. E. M. Schwarz, R. M. Averil III, and L. J. Sigal. A radix-8 CMOS S/390 multiplier. In T. Lang, J.-M. Muller, and N. Takagi, editors, *Proceedings, 13th Symposium on Computer Arithmetic*, pages 2–9, Bath, England, July 6–9 1997. IEEE Computer Society Press, Los Alamitos, CA.
17. E. E. Swartzlander, editor. *Computer Arithmetic*, volume I. IEEE Computer Society Press, Los Alamitos, CA, 1990.
18. A. F. Tenca and Ç. K. Koç. A scalable architecture for Montgomery multiplication. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science No. 1717, pages 94–108. Springer, Berlin, Germany, 1999.
19. G. Todorov. Asic design, implementation and analysis of a scalable high-radix montgomery multiplier. Master's thesis, Department of Electrical and Computer Engineering, Oregon State University, December 2000.

20. W. C. Tsai, C. B. Shung, and S. J. Wang. Two systolic architectures for Montgomery multiplication. *IEEE Transactions on VLSI Systems*, 8(1):103–107, February 2000.
21. 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.