A SCALABLE ARCHITECTURE FOR RSA CRYPTOGRAPHY ON LARGE FPGAS

E. Allen Michalski, Duncan A. Buell

CSE Department
The University of South Carolina
Columbia, SC 29208, U.S.A.
email: michalsk@cse.sc.edu, buell@sc.edu

ABSTRACT

The RSA algorithm is the standard for public-key cryptography today, with Montgomery multiplication the most common mechanism of implementation due to modulo operations using a bitwise shift in place of a division operation. Several Montgomery designs have been proposed for ASIC and FPGA implementation based on limited resource availability to satisfy the computational burden. FPGAs are now available that have large configurable logic resources in addition to dedicated high-speed ALU logic for operations such as multiplication. This paper describes an improvement to a limited resource Montgomery multiplier design, the MWR2MM algorithm proposed by Tenca and Koc, which is suitable for implementation on large FPGAs. The design can be scaled to utilize available FPGA multipliers, CLB logic and frequencies of operation. Implementation and design choices are discussed for an RSA core based on this design, and a comparison against the OpenSSL open source cryptographic library is given. Our results show a 1024-bit RSA core on a 100MHz Virtex2 Pro 100 FPGA platform to be 3.13x faster than an equivalent software implementation on a 2.8 GHz Intel Xeon PC workstation.

1. INTRODUCTION

RSA is a public-key algorithm that has become the default standard for public key cryptography. It is based on modular multiplications and exponentiation, which is computationally resource-intensive and time-consuming given a minimumally secure operand size of 1024 bits or greater. Peter Montgomery in [3] proposed a method of modular multiplication that replaces modular division with a simple power-of-2 bit shift, at the expense of mapping the operand into and out of the Montgomery domain. This operation is well-suited for modular exponentiation where the cost of Montgomery domain mapping is a small percentage of operations compared to the number of multiplications due to exponentiation.

Previous research exists in [4,5,6,7,8] for both microprocessor and hardware-based implementations based on Montgomery multiplication, since the algorithm is ideally suited for word-based implementation. One advantage of a dedicated hardware solution is based on exploiting the deep parallelism inherent in exponentiation of large exponents and operands, which can be sequentially limited when using a general-purpose microprocessor.

While much of previous research in [4,5,6,7,10] focused on limited-resource implementations, this paper proposes a hardware-based RSA design suitable for implementation on large FPGAs with dedicated multiplier arithmetic resources. We propose a improved high-radix Montgomery multiplier design based on a limited resource ASIC design, the Multiple Word Radix-2 Montgomery Multiplication algorithm presented by Tenca and Koc in [4]. The design is scalable based on operand size, available FPGA multipliers and configurable logic, and FPGA frequencies of operation. We discuss the design and implementation of the Montgomery multiplier and an RSA core, and FPGA area and execution time tradeoffs are evaluated based on resource availability. An implementation of a complete RSA core is discussed for two Xilinx FPGA families, and a comparison between the hardware implementations and a public-domain software implementation of RSA using OpenSSL are given. Finally, a conclusion comparing design tradeoffs to performance is provided.

2. AN INTRODUCTION TO RSA AND MONTGOMERY MULTIPLICATION

RSA [1,2] is a block cipher based on the modular exponentiation of integers, with plaintext with values less than a calculated modulus m and processed in block sizes of k such that $2^{k-1} < M < 2^k$. Encryption requires the knowledge of a publicly available key by the encrypting party, consisting of an exponent e and the modulus M. Encryption is performed on a plaintext block P producing a ciphertext block C such that:

$$C = P^e \bmod M \tag{1}$$

Decryption requires a secretly-held private key by the decrypting party, consisting of an exponent d and the modulus M. Decryption is performed on a ciphertext block C producing a plaintext block P such that:

1-4244-0 312-X/06/$20.00 ©2006 IEEE.

$$P = C^d \bmod M = P^{ed} \bmod M \quad (2)$$

Values for e, d and M are related through the choice of two large prime numbers p and q and the use of Euler's theorem and its corollary [9], such that:

$$M = pq \quad (3)$$
$$ed = 1 \bmod \varphi(M), \text{ where } \varphi(M) = (p-1)(q-1) \quad (4)$$

(3) shows the resulting modulus M is a product of two odd numbers and is itself an odd value. (4) guarantees a unique inverse relationship between the encryption exponent e and decryption exponent d.

2.1. Montgomery Multiplication

Exponentiation requires the use of modulo multiplication operations, defined as $C = XY \bmod M$ where $0 <= X, Y < M$. Peter Montgomery proposed an implementation of XY mod M without division by M [3] that implements division using a simple bit shift, with an overhead of transforming operands into and out of the Montgomery domain. While conventional algorithms are faster in the case of simple modulo multiplications, Montgomery multiplication realizes a significant speed benefit when multiple successive multiplications are required, as in exponentiation for the case of RSA.

Montgomery multiplication is defined as:

$$C' = \text{MM}(X', Y') = X'Y'R^{-1} \bmod M \quad (5)$$

where MM(X', Y') operates on numbers in the Montgomery domain and R is a calculated power of 2.

The steps involved for Montgomery multiplication are implemented as follows:

1) Pick an integer R such that $R = 2^n$ and the modulus M is in the range $2^{n-1} < M < 2^n$. If M is represented by n bits, $R = 2^n$, where R is even and M is odd. R^2 mod M, required in the next step, must be precalculated.
2) Move operands X and Y into the Montgomery domain, producing X' and Y'. The Montgomery domain representation of an operand X is defined as:

$$X' = \text{MM}(X, R^2) = XR^2R^{-1} \bmod M = XR \bmod M \quad (6)$$

3) Montgomery multiplication operations against two Montgomery domain operands X' and Y' produce a result C' also in the Montgomery domain:

$$C' = \text{MM}(X', Y') = (XR)(YR)R^{-1} \bmod M = XYR \bmod M \quad (7)$$

4) Transform the Montgomery domain result C' to a normal representation C using $C = \text{MM}(C', 1)$:

$$C = \text{MM}(C', 1) = (CR)(1)R^{-1} \bmod M = C \bmod M = XY \bmod M \quad (8)$$

Operations in the Montgomery domain are efficient due to the implementation of the operation $X'Y'R^{-1}$ mod M, which can be calculated using the relationship:

$$(t' + uM)/R \equiv t'R^{-1} \bmod M \quad (9)$$

where $t' = X'Y'$ and $(t' + uM)/R$ is referred to as Montgomery reduction.

The multiple u of modulus M is calculated using the following two relationships:

$$M' = -M^{-1} \bmod R \quad (10)$$
$$u = t'M' \bmod R \quad (11)$$

(10) is a constant which can be precalculated, thus Montgomery reduction reduces to three steps: the calculation of u based on the value of t', the calculation of uM, and the addition of t' and uM and a corresponding right shift by $R = 2^n$. The relationships between (9), (10) and (11) are such that adding uM to t' has the effect of zeroing of the least significant radix-R digit of the product $t' = X'Y'$, while division by R shifts the addition of $t' + uM$ to remove the least significant digit of the result, which is equal to zero. The simplicity of Montgomery reduction is that the operation requires division by R instead of M. Since R is a power of 2, division is reduced to a shift operation. A byproduct of the presented relationships is that the final result for (9) is less than $2M$, for which a final subtraction of M is needed if the final result is greater than M.

The above operations implement reduction in one step. If all integers are represented with o digits of radix k where $R = k^o$, reduction can be performed in increments of one digit. This is critical to the implementation of Montgomery multiplication in FPGA hardware as will be seen in the following sections.

3. MONTGOMERY MULTIPLICATION: THE MWRKMM ALGORITHM

Alexandre Tenca and Cetin Koc proposed an efficient ASIC implementation of Montgomery multiplication in [4] based on a radix-2 reduction scheme, which simplifies reduction to a bitwise calculation. A brief description follows, with the reader referred to [4] for greater detail.

Algorithm one shows the Multiple Word Radix-2 Montgomery Multiplication (MWR2MM) algorithm, which involves scanning the operands M and Y in e words of w-bits per word and the multiplier operand X in m bits, where

Algorithm 1
MWR2MM Algorithm

```
S = 0 -- Initialize all words of S.
for i = 0 to m - 1
  (C; S(0)) := xiY(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 + xiY(j) + M(j) + S(j)
      S(j-1) := (S(j)0; S(j-1)w-1::1)
    S(e-1) := (C ; S(e-1)w-1::1)
  else
    for j = 1 to e - 1
      (C; S(j)) := C + xiY(j) + S(j)
      S(j-1) := (S(j)0; S(j-1)w-1::1)
    S(e-1) := (C; S(e-1)w-1::1)
```

Algorithm 2
MWRkMM Algorithm

```
-- v-bit words, w-bit reduction, m-bit ops
-- m',u',x are w-bit; S,M,Y v-bit widths
-- m'S, u'M and xY are v+w bit products

p = m/w      -- # of reduction digits.
n = m/v + 2 -- # of operand words (one
            --   word is zero).
S = 0
for i = 0 to p
   u' := m'S(0) mod 2w

   (Cum(1); Pum(0)) := u'M(0)
   (Cmod(1); Smod(0)) := S(0) + Pum(0)
   for j = 1 to n - 1
     (Cum(j+1); Pum(j)) := u'M(j)
     (Cmod(j+1); Smod(j)) := Cmod(j) + S(j)+ Cum(j)+
Pum(j)
     S(j-1) := (Smod(j) w-1::0; Smod(j-1)v-1::w)

     -- Last x digit xp = 0.
     (Cxy(j); Pxy(j-1)) := xiY(j-1)
     (C(j); S(j-1)) :=  C(j-1)+ S(j-1)+ Cxy(j-1)+ Pxy(j-
1)
```

$M = (M^{(e-1)}, \ldots, M^{(1)}, M^{(0)})$, $Y = (Y^{(e-1)}, \ldots, Y^{(1)}, Y^{(0)})$, $X = (x_{m-1}, \ldots, x_1, x_0)$ and $e = \lceil (m+1)/w \rceil$ words which includes the extra bit required to account for the result being in the range of $[0, 2M - 1]$. Utilizing bitwise multiplication of x_i allows the reduction calculation to be determined during the summation calculation of the least significant word of S, reducing the least significant bit of S in each loop until the least significant half of the output S is reduced to zero. In [4] it was shown that the minimum cycle time for the above design is given by $Tcyc_{min} = 2m + 1$. This result is observed in the MWR2MM algorithm's word-by-bit S result output, for which each j loop has a two-iteration delay to align the S result for the next i loop iteration, and there are m i-loop iterations before the result is obtained.

This design is ideal for limited-resource environments. Work presented in [10] extends this design to a radix-8 implementation using booth-recoded multipliers, described using an MWR2kMM algorithm. While Booth recoding reduces the number of partial product accumulations and thus the delay, their use is typically limited to radix-8 or lower due to the increasing control and number of precomputed multiples of the multiplicand []. A significant advantage of the MWR2MM design is that it is systolic in nature, with adjacent processing elements responsible for carrying out word-by-bit reductions for a particular x-operand bit, and lacking broadcast elements which tend to decrease the frequency of operation. We modify this design to handle a variable-radix digit reduction, with larger reduction radices achieved through the use of dedicated multiplier capabilities within an FPGA.

3.1. The Proposed MWRkMM Algorithm

The MWRkMM algorithm given by algorithm two scans Y and M operands word-by-word and the X operand digit-by-digit. This algorithm differs from the MWR2MM and MWR2kMM algorithms by performing the reduction before adding the product xY to the reduced S result, with the S output of the ith loop reduced by the subsequent loop iteration. As a result, the first loop of i reduces a zero S sum, and one additional X-digit iteration is needed to produce a correctly reduced final result in the range of $2M - 1$, ensured when the last X-digit is zero.

The number of words required to represent the S result $2M - 1$ must be extended by one to account for the S result output requiring $m + 1 + w$ bits. The resultant S output from each loop is contained in n words, where n is one word longer than the e words required by the MWR2MM algorithm. As a result the Y and M operand number of words are extended to include one zero word. Multiplications produce a product of length $v + w$ bits, with the current product P width of v bits and the carry C of w bits. The additions produce a carry that is bounded by the equation:

$$3(2^v - 1) + \text{carry} = \text{carry}*2^v + (2^v - 1) \qquad (12)$$

assuming $v = w$. This is satisfied when carry = 2, which is represented by two bits. This result is the same as [4].

4. MWRKMM PARALLEL IMPLEMENTATION

An FPGA implementation of the MWRkMM algorithm is dependent upon available multiplier resources and their ratio to available FPGA configurable slice logic. Our implementation assumes a small number of FPGA multiplier resources relative to the available slice count, thus we analyze the MWRkMM algorithm to make efficient use of multiplier resources at the expense of additional FPGA slice logic. From algorithm two it can be seen that dependencies between j loops involve the propagation of

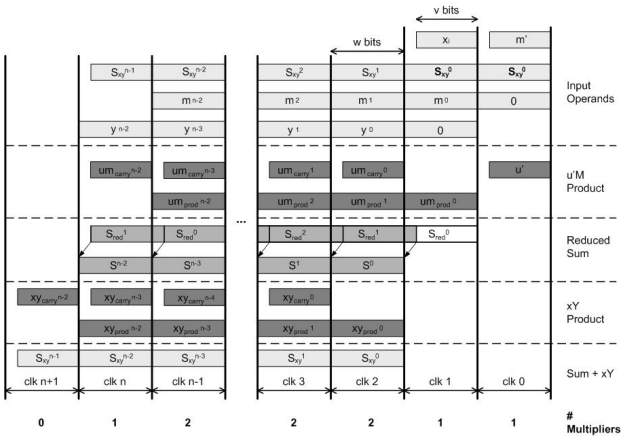

Fig. 1: MWRkMM *j*-loop Multiplier Dependency Analysis

operand words and carry information while *i* loops are dependent on previous sum calculations. While parallel computations are possible between adjacent *i* and *j* loops, we focus on parallelism that can be extracted from adjacent *i*-loop calculations as completely unrolling both loops is infeasible within one FPGA for a minimally-secure operand size of 1024 bits.

Fig. 1 shows the number of multipliers required to maximize the speed of operation of each *j*-loop iteration, assuming each iteration is equivalent to one clock tick. Two multiplications are required for iterations other than the first or last two: one for the *u'M* product and one for the *xY* product. During the first and last two iterations the number of multiplications is one (the calculation of the *u'* reduction multiplier) or zero (allowing the final *xY* product carry to add to the *S* sum), which gives opportunity to overlap these calculations in a pipelined implementation of the loop. We take advantage of this in the implementation of the multiplier circuit.

Parallel computation of the algorithm is possible between adjacent *i* loops, with each *i*-loop implementing a reduction by the product *u'M* along with a *xY* operand product and summation. An unrolled dependency graph of the MWR2MM algorithm is shown in Fig. 2, where each circle represents an *i-j* loop atomic execution. Each graph row represents successive *i*-loop iterations and dependencies while each column represents successive *j*-loop iterations and dependencies. [4] proposed allocating a separate processing element (PE) to each column's calculations. Each row can thus be interpreted as representing the state of adjacent PEs column calculations in relation to each other.

The three states that each PE performs are determined by an analysis of data calculations and overlap that are possible from Fig. 1. State A corresponds to the calculation of the *u'* constant of the current *x*-digit column allocated to the PE in combination with the completion of the clock-*n* calculation from a previous *x*-digit calculation allocated to the same PE. State B combines the clock-1 calculation of

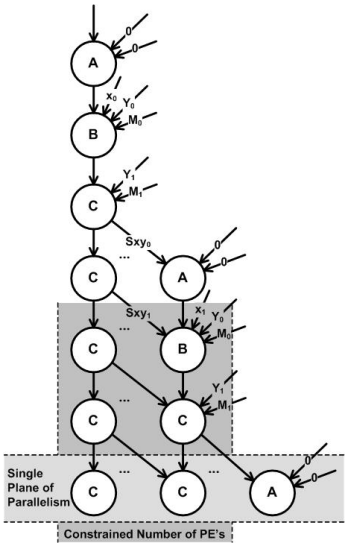

Fig. 2: MWRkMM Dependency Graph

the current *x*-digit with the clock *n*+1 calculation of the previous *x*-digit. State C represents the addition of $S + u'M$ reduction with the *xY* product of the current *x* digit. From the graph it is observed that there is a three-clock delay in the processing of data between adjacent column digits x_i and x_{i+1}. With this analysis we are now ready to implement this design.

5. A SCALABLE ARCHITECTURE

The implementation of the MWRkMM algorithm is based on the use of a systolic array of PEs in combination with operand registers and additional shift registers. The maximum number of PEs that can be instantiated is equivalent to the largest degree of parallelism attained with this design, which is given by:

$$p_{max} = (\lfloor m/v \rfloor + 2)/3 \qquad (13)$$

$(\lfloor m/v \rfloor + 2)$ represents the number of words required for a *m*-bit operand and 3 is the latency of the PE, with greater than p_{max} PEs stalling the pipeline and not producing any increase of speed. When $p <= p_{max}$ units are available, the computation can be performed with fewer units by adding delay registers between the last and first PEs, creating a shift register of size $(\lfloor m/v \rfloor + 2) - 3p$ that pipelines the results of the last processing element into the first when the first PE is complete with its previous column's calculation. Fig. 3 shows a hardware representation of a MWRkMM implementation with two PEs.

5.1. The Processing Element Multiplier

The processing element (PE) consists of both a multiplier component and an addition unit. The multiplier is shown in

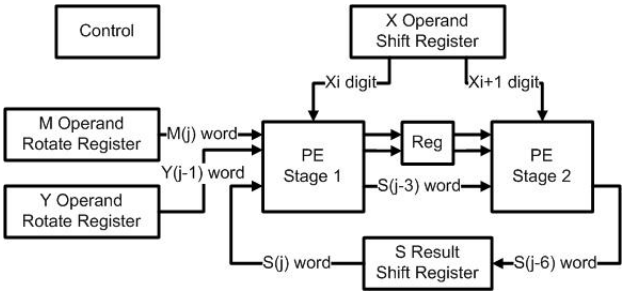

Fig. 3: MWRkMM Hardware Representation

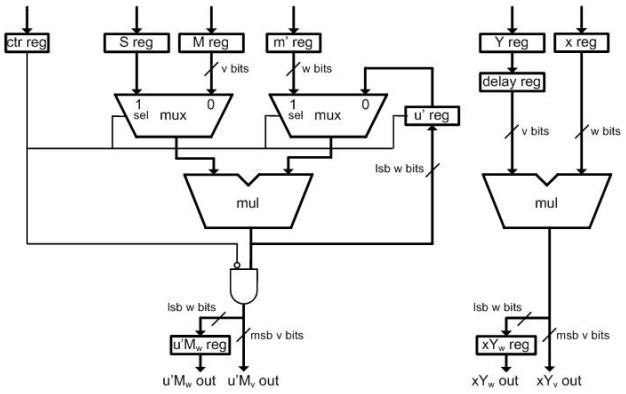

Fig. 4: PE Multiplier Unit

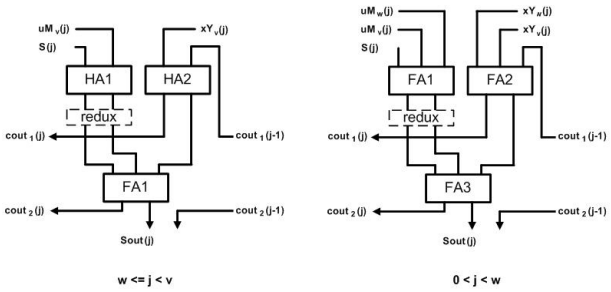

Fig. 5: PE Addition CSA Adders

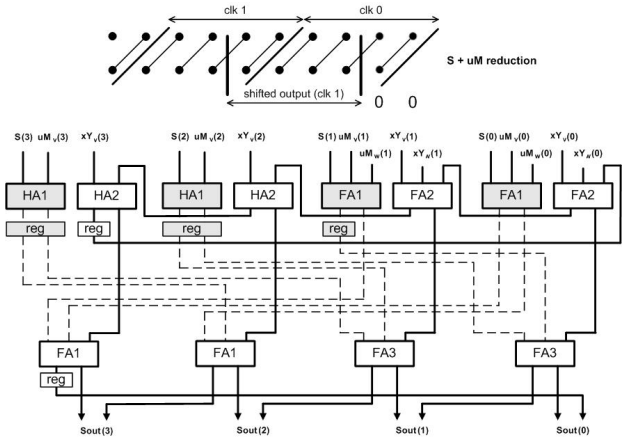

Fig. 6: PE Addition CSA Data Path

Fig. 4, and consists of two multipliers that are responsible for computing *u'*, the *u'M* product and the *xY* product. As two multipliers are responsible for three products, additional muxs are required on the *u'M* product multiplier to handle the computation of *u'*. In addition, the overlap of calculations in state A requires additional gating logic at the output of the multiplier circuit. The overlap of calculations in state B is possible without any additional logic due to the zero in the *Y* operand at clock *n*+1. Both multipliers take in operands of *u* and *v* bit widths and produce a product of *u*+*v* bits, with the upper *v* bits registered as carry. Note that the Y operand is delayed one clock to handle the iteration difference between the two product calculations as shown in Fig. 2.

5.2. The Processing Element Addition Unit

The addition unit of a PE implements the logic for the addition and reduction of $S_{i-1,j}$ and the $u'M_j$ product and carry, and the addition of the x_iY_j product and carry to the previous result. We implement a combination of carry-save carry-propagate architectures to produce the final result and reduce critical path timing for the addition of five inputs. The carry-save architecture reduces to three inputs when the current bit position *i* is greater than the carry word width *w*. Alignment requires a portion of the output to be registered and combined with the previous clock's shifted result. The CSA column adders for both cases are shown in Fig. 5. An example of the addition CSA architecture for $v = 4$ bits and $w = 2$ bits is shown in Fig. 6, with the $S + u'M$ reduction indicated by the dashed interconnections. The final $S+xY$ reduced-form sum is produced by a built-in FPGA fast carry-propagate adder to calculate the result, and takes as input the CSA result to produce a CPA sum with carry.

5.3. Critical Path

The critical path of the original MWR2MM algorithm was reduced through the use of CSA logic between PE stages. One PE implements the multiplier logic, the carry-save addition path, and the reduction of the carry-save *S* result to produce a reduced sum for the next PE's *S* register input. The critical path of this design is thus dependent on the availability of the multiplier output bits, the shifting logic in the reduction CSA tree that shifts most-significant-bit to least-significant-bit positions and the final CPA adder for the *S* reduced result. For both multiplier and sum calculations, bit-output timing from earliest-to-latest delay is observed from the least-significant to most-significant output bits. We evaluate the placement of the *S* reduction adder at the output of the CSA tree and propagating the reduced sum between PEs, which increases the critical path but make the design more favorable to subpipelining the PE at the multiplier output to increase timing.

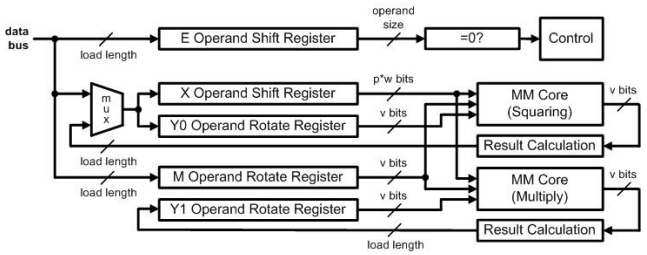

Fig. 7: RSA Core

5.4. Final Result Calculation

The Montgomery multiplication design requires an implementation for a final adder that latches the result from the PE indicated by the remainder of $(\lfloor m/w \rfloor + 1)/p$, which represents the number of radix-2^w digit reductions using an m-bit X operand, divided by the number of PEs instantiated in the design. The last PE is thus indicated by a remainder of zero and the remaining PEs are numbered sequentially from left to right.

The final adder handles subtraction of M from the result when the result is greater or equal to M. The design makes use of fast carry-propagate logic as in the PE addition to calculate the result, with stages of the addition registered to reduce the critical path. The subtraction of a word m_j is handled during the subsequent clock after a word of the result is latched after it is available from the output PE. The latency of the output calculation is equal to the number of operand words plus two clocks, equal to the latency of the subtraction calculation before a word is latched. Selection of the correct result can be determined based on the presence of a carry bit after the final word of M is added in the $R - M$ calculation, signaling a positive result and hence selecting this result for output. Overflow considerations are not needed since the final result when latched from the output PE is within the range $[0, 2M - 1]$, which will not indicate an overflow after subtraction of M.

5.5. RSA Exponentiation

Several choices exist for algorithms that implement modular exponentiation. One common choice shown in algorithm six is right-to-left exponentiation [9], which is chosen for its lack of dependency between loop operations and simplicity of a parallel implementation. The right-to-left exponentiation algorithm is shown in algorithm four. Two Montgomery multipliers are required to implement maximum parallel operations within right-to-left exponentiation, where both A and S can be computed in parallel. An additional exponent shift register and comparator are used to implement exponent loop bounds. Our RSA core utilizes this method for exponentiation, and is shown in Fig. 7.

Algorithm 4
Right-to-Left Exponentiation

INPUT: An element $g \in G$, an integer $e \geq 1$
OUTPUT: g^e

1. $A \leftarrow 1$, $S \leftarrow g$.
2. While $e \neq 0$ do:
   2.1 If e is odd then $A \leftarrow A * S$
   2.2 $e \leftarrow e/2$
   2.3 If $e \neq 0$ then $S \leftarrow S * S$
3. Return A.

Five data values are loaded into this core using a bus-loaded acknowledged data transfer, given a user-supplied bus width of z bits. The data loaded are (1) the data to be encrypted/decrypted, (2) the public or private key, (3) the modulus M, (4) the constant R^2 mod M, and (5) m'. The first four operands are m bits in width, however m' is only required to be w bits wide as the calculation of u' requires only the w least significant bits of the result. After exponentiation is complete the result is read using the same transfer protocol.

6. PERFORMANCE VS. DESIGN CONSTRAINTS

The Montgomery multiplication cycle time for the MWRkMM algorithm is given by:

$$\text{Tcyc} = ((\lfloor m/w \rfloor + 1)/p)(\lfloor m/v \rfloor + 2) + 3(\lfloor m/w \rfloor + 1)/p \text{ if } m\%n > 0 \quad (14)$$

$$\text{Tcyc} = ((\lfloor m/w \rfloor + 1)/p)(\lfloor m/v \rfloor + 2) + (\lfloor m/v \rfloor + 2) \text{ if } m\%n = 0 \quad (15)$$

where % is the remainder operation. In these equations $(\lfloor m/w \rfloor + 1)/p$ represents the number of loops necessary to perform Montgomery multiplication given the number of PEs p, and the term $(\lfloor m/v \rfloor + 2)$ represents the number of words in an operand. The remaining terms provide the offset number of clock cycles before the result is available, which is dependent on the output PE. Fig. 8 shows the execution times of a number of combinations of v and w for a 1024-bit operand size. Note that the choice of the number of addition bits v has little impact on the time of execution compared to the multiplier reduction radix bit width w. For a given w the execution time is estimated by:

$$m/w*(PE\ latency) \quad (16)$$

where the latency of this design is 3. Given that the PE latency of the original MWR2MM algorithm is 2, the MWRkMM algorithm is approximately $2/3*w = 0.66w$ times as fast as the original. Hence our focus is to find the

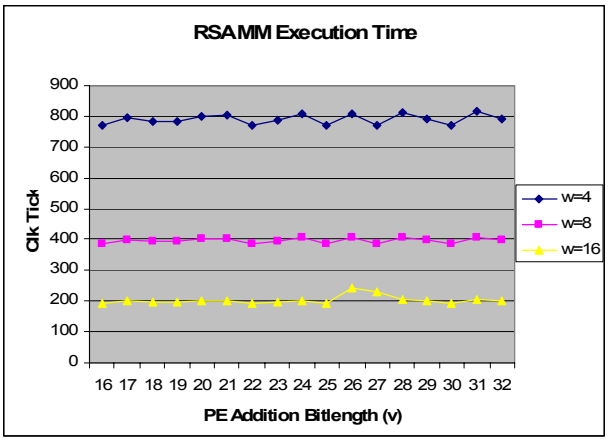

Fig. 8: MM Execution Time for 1024-bit Operands

maximum size of w that satisfies the MWRkMM and FPGA design constrains.

6.1. Design Constraints

While the original MWR2MM results targeted an ASIC fabric, designing for use within an FPGA environment necessitates different constraints. The following is a discussion of FPGA and MWRkMM constraints and how they interact:

1) The choice of the PE addition word size v impacts the maximum number of PEs that can be instantiated, which is given in the relationship presented in (13). This choice also impacts the critical path within the FPGA.
2) The choice of the reduction radix bit width w must satisfy the constraint $w <= v$, which insures that the reduction only takes two clock cycles in states B and C of the PE operation. w must also be an integer divisor of the operand length to ensure proper Montgomery reduction.
3) The combination of v and w impacts the critical path timing through the multiplier. Both v and w are bounded by the maximum multiplier input operand size supported by the FPGA. In addition, the multiplier timing results for Xilinx embedded multipliers is dependent on the bit width of the product versus that of the inputs. Input sizes of 4x6, 5x5 and 6x4 all produce the same 10-bit output propagation time.
4) The number of multipliers must be less than what is available within the FPGA. There are two multipliers per PE and two Montgomery multiplication cores for a parallel RSA implementation, thus the constraint to satisfy is given by $4p <=$ # of FPGA multipliers.

For this design we focus on FPGAs where the maximum number of PEs are able to be instantiated. The following sections provide timing and simulation results and compares this against an open source software RSA implementation.

7. A 1024-BIT RSA CORE: AREA AND TIMING RESULTS

Two versions of the RSA core were implemented: one version which had a PE latency of three clocks and one version which adds a pipeline stage after the PE multiplier output to improve critical path timing. The core was compiled with the Xilinx 7.1.04i integrated development environment using Xilinx synthesis technology and place and route tools. Synthesis was optimized for speed with an optimization effort of high, while PAR options included optimizations for speed, an effort level of high and an initial placer table entry of 23.

Results were gathered for a 1024-bit implementation using a PE addition width v of 17 bits and a reduction bit width w of 16 bits. These operating points are optimal given design and FPGA constraints: 16 bits is an even divisor of the 1024 operand length and the maximum unsigned input length to the 18x18 FPGA multiplier is 17 bits, noting that all the design constraints are met using these inputs while producing optimal performance points. Results are shown in Tables 1 and 2 for two Xilinx parts at their lowest speed grades. All designs included a 10 ns clock timing constraint, with both of the subpipelined versions able to meet timing on both chips chosen for synthesis.

Table 1. 1024-bit RSA PAR Results

Test Case	Slices	Slice FFs	LUTs	Mults	Period (ns)	Freq (MHz)
v2p100	12,791 (29%)	17,992 (20%)	19,213 (21%)	80 (18%)	11.972	83.5
v4fx100	14,910 (35%)	17,831 (21%)	19,692 (23%)	80 (50%)	13.409	74.6

Table 2. 1024-bit Subpipelined RSA PAR Results

Test Case	Slices	Slice FFs	LUTs	Mults	Period (ns)	Freq (MHz)
v2p100	12,162 (27%)	19,772 (22%)	16,359 (18%)	60 (13%)	9.622	103.9
v4fx100	14,329 (33%)	119,784 (23%)	17,003 (20%)	60 (37%)	9.914	100.9

To compare synthesized clock periods to execution time, a simulation was performed on a 1024-bit decryption key, which is representative of the maximum time the core will take to execute as public key exponents are chosen as a small prime number. The time of one Montgomery multiplication indicates a complete Montgomery reduction

operation including overhead to load results back into appropriate registers and reset the control for the next Montgomery reduction. For a 1024-bit decryption operation a total of 1026 reductions are performed in the worst case, which includes the 1024 bits of the operand and 2 bits representing the time of transition into and out of the Montgomery domain.

Utilizing the Virtex2 Pro part, one Montgomery multiplication in the subpipelined design takes 3,730 ns at 100 MHz, with a total exponentiation time of 3.829 ms. The original design at 83 MHz performs one Montgomery multiplication at 3,732 ns, with a total exponentiation time of 3.82 ms. The similar times for decryption indicate that the speedup due to subpipelining is offset by the decreased number of multipliers to implement maximum parallelism. The maximum number of PEs instantiated for the original design at 20 versus the maximum 15 possible for the subpipelined version.

8. OPENSSL LIBRARY COMPARISON

A comparison against an OpenSSL 0.9.7d library was performed given the same data input, with the algorithm for decryption making use of the same constant definitions as the RSA core. An average of ten runs was taken on a dual 2.8 GHz Xeon workstation, with each run taking an average of 11.996 ms, which is a 3.13x speedup for both cores. These results are promising, noting that it is possible to gain additional speed through functional parallelism by implementing multiple cores on one Virtex2 Pro 100 FPGA. Software results are shown for both cores in Table 3.

Table 3. 1024-bit RSA Core HW/SW Comparison

Test Case	Clock Period	Execution Time	SW Execution Time	Speedup
Pipelined	10 ns	3.829 ms	11.996 ms	3.13x
Nonpipelined	12 ns	3.831 ms	11.996 ms	3.13x

9. CONCLUSION

In this paper we have proposed an improved version of the MWR2MM algorithm, the MWRkMM algorithm, for use on large FPGAs. Our design is dependent on the availability of dedicated multiplier resources in addition to a large reconfigurable user logic. We explore design constraints against performance within an FPGA environment, and demonstrate an efficient implementation for a 1024-bit RSA core consisting of two MWRkMM units. A comparison against an open-source software OpenSSL implementation shows a speedup of 3.13x compared to a 2.8 GHz Xeon workstation. Our results are based on an appropriate ratio of user logic to multiplier resources, with both FPGAs tested allowing for additional functional parallelism through multiple core implementation.

10. REFERENCES

[1] W. Diffie, M. Hellman, “New Directions in Cryptography,” IEEE Transactions on Information Theory, November, 1976.

[2] R. Rivest, A. Shamir, L. Aldeman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” Communications of the ACM, February, 1978.

[3] P. Montgomery, “Modular multiplication without trial division,” Mathematics of Computation, No. 44, pp. 519–521, 1985.

[4] A. Tenca, C. Koc, “A Scalable Architecture for Montgomery Multiplication,” CHES 99, Lecture Notes in Computer Science, No. 1717, pp. 94-108.

[5] A. Tenca, C. Koc, “A Scalable Architecture for Montgomery Multiplication Based on Montgomery’s Algorithm”, IEEE Transactions on Computers, Vol. 52, pp. 1215-1221, Sept., 2003.

[6] A. Tenca, G. Todorov, C. Koc, “High-radix design of a scalable modular multiplier,” Cryptographic Hardware and Embedded Systems - CHES 2001, C. K. Koc, D. Naccache, and C. Paar, editors, Third International Workshop, Paris, France, pages 185-201, Springer Verlag, LNCS No. 2162, May 14-16, 2001.

[7] C. D. Walter, “Montgomery's Multiplication Technique: How to Make It Smaller and Faster”, Cryptographic Hardware and Embedded Systems - CHES '99, LNCS No. 1717, pp. 80-93. Springer-Verlag, August 1999.

[8] G. Hachez, J. Quisquater, “Montgomery exponentiation with no final subtraction: Improved results”, Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2000, Worchester, USA, July 2000, LNCS 1965, pp. 293-301.

[9] B. Parhami, “Computer Arithmetic Algorithms and Hardware Designs”, Oxford University Presss, 2000.