A Scalable Hardware Architecture for Prime Number Validation

Ray C.C. Cheung, Ashley Brown, Wayne Luk
Department of Computing
Imperial College London, UK
r.cheung, ashley.brown, w.luk@ic.ac.uk

Peter Y.K. Cheung
Department of EEE
Imperial College London, UK
p.cheung@ic.ac.uk

Abstract

This paper presents a scalable architecture for prime number validation which targets reconfigurable hardware. The primality test is crucial for security systems, especially for most public-key schemes. The Rabin-Miller Strong Pseudoprime Test has been mapped into hardware, which makes use of a circuit for computing Montgomery modular exponentiation to further speed up the validation and to reduce the hardware cost. A design generator has been developed to generate a variety of scalable and non-scalable Montgomery multipliers based on user-defined parameters. The performance and resource usage of our designs, implemented in Xilinx reconfigurable devices, have been explored using very large prime numbers. Our work demonstrates the flexibility and trade-offs in using reconfigurable platform for prototyping cryptographic hardware in embedded systems. It is shown that, for instance, a 1024-bit primality test can be completed in less than a second, and a low cost XC3S2000 FPGA chip can accommodate a 32k-bit scalable primality test with 64 parallel processing elements.

1. Introduction

For centuries, the problem of validating prime numbers, the primality test, has posed a great challenge to both computer scientists and mathematicians [5]. The problem of identifying Prime and Composite numbers is known as one of the most important problems in arithmetic. Many security applications also involve large numbers: while it is easy to multiply prime numbers to get a product, the reverse process of recovering the primes is much more difficult.

Field programmable gate arrays offer a rapid-prototyping platform for the verification of embedded systems [16]. FPGAs are also used as active components in security systems, such as Firewall processors [12] and Elliptic Curve Cryptographic processors [13]. As the threat of attacks appears to be increasing, many existing systems do not seem to be secure enough. One of the reasons is due to the use of weak key generation. It has been recognised that strong prime number generation is important, and the prime validation is an intrinsic part of the generation. Here we define validation as the process of performing prime testing on a given number.

This paper presents a scalable architecture for prime number validation which targets reconfigurable hardware such as FPGAs. In particular, users are allowed to select predefined scalable or non-scalable modular operators for their designs. Our main contributions include:

- Parallel designs for Montgomery modular arithmetic operations (Section 3).
- A scalable design method for mapping the Rabin-Miller Strong Pseudoprime Test into hardware (Section 4).
- An architecture of RAM-based Radix-2 scalable Montgomery multiplier (Section 4).
- A design generator for producing hardware prime number validators based on user-specified parameters (Section 5).
- An implementation of the proposed hardware architectures in reconfigurable devices, with an evaluation of its effectiveness compared with different size and speed tradeoffs (Section 6).

The rest of the paper is organized as follows. Section 2 describes the basic principles of the Rabin-Miller algorithm and the scalable Montgomery modular multiplication algorithm, and related work. Section 3 presents a design flow for mapping scalable prime number validators into hardware. Section 4 covers a scalable architecture for modular multiplication. Section 5 presents a design generator $GenDesign$ which produces architectures with different design trade-offs, based on user-specified parameters. Section 6 evaluates experimentally the performance and area usage of the proposed approach using FPGAs, and compares it with different architectures. Finally, Section 7 summarises our current and future research.

2. Background and Related work

Primality test is essential for prime number generation. Cryptography often uses large prime numbers, for instance, RSA public key algorithm requires 512-bit, 1,024-bit for key generation [21]. Much research work has been done on primality test and generation, and most algorithms are based on factorization [20]. In [2], the primality test has been proved to be solvable in polynomial time. However, it requires a log operation which is expensive for hardware implementation; it also requires knowledge of the primality of preceding numbers, which is impractical for arbitrary prime testing.

In [10], Joye et al has proposed an efficient prime number generation scheme based on pseudo-random number generation and implemented the design on a smart-card platform. Lu et al [14] has further developed the RSA key generation for smart card using different prime number algorithms.

There is a special class of very large prime numbers which have the representation of $2^n - 1$. They are called Mersenne prime numbers. For example, the 41^{st} Mersenne prime number $2^{24,036,583} - 1$ was discovered in May 2004. A clustered workforce GIMPS [9] in the Internet is currently dedicated on locating the next Mersenne number.

2.1. Rabin-Miller Primality Test

In this paper, we have selected the Rabin-Miller probabilistic primality test algorithm (figure 1) as the core of the primality test. This algorithm is based on the properties of strong pseudoprimes and has also been adopted by Mathematica's PrimeQ command [15]. This algorithm is developed by Rabin [19] based on Miller's [18] idea. The algorithm can quickly determine the primality of a given large number with a controllably small probability of error [1]. It requires a number of small primes for repeatedly testing the input number. The probability of falsely identifying a composite as a prime decreases with every additional small prime used. This method enables tradeoff between accuracy (more small primes) and efficiency (fewer small primes) for different applications.

From the above algorithm, "`Inconclusive`" implies the number p maybe prime. The selection of number b is based on a set of small primes Z where

$$Z = 2, 3, 5, 7, 11, 13, 17, 19, ..., n \leq p - 1$$

With the use of first eight small primes, the 100% accuracy of primality test can be achieved for numbers up to 3.4×10^{10} [1]. The actual number of small primes can be customised by the design generator presented in Section 4 in this paper.

```
Input: The number under primality test: p

1. Random choose a number b in [1, p-1] ∈ Z
2. Let p = 2^(q×m)+1 where m is an odd integer
3. IF either
4.    Case 1: b^m = 1 (mod p) or
5.    Case 2: there is an integer i in [0, q-1]
6.       such that b^(m×2^i) = -1 (mod p)
7.    RETURN "Inconclusive"
8. ELSE
9.    RETURN "Composite"
```

Figure 1. The Rabin-Miller Probabilistic Primality Test Algorithm.

```
Input: X * Y (mod M), Output: S
1.  S = 0
2.  FOR i = 0 to n - 1
3.     (Ca,S^(0)) = x_i*Y^(0) + S^(0)
4.     IF S_0^(0) = 1 THEN
5.        (Cb,S^(0)) = S^(0) + S^(0)
6.        FOR j = 1 to e
7.           (Ca,S^(j)) = C_a + x_i*Y^(j) + S^(j)
8.           (Cb,S^(j)) = C_b + M^(j) + S^(j)
9.           S^(j-1) = (S_0^(j), S_(w-1..1)^(j-1))
10.       END FOR
11.    ELSE
12.       FOR j = 1 to e
13.          (Ca,S^(j)) = C_a + x_i*Y^(j) + S^(j)
14.          S^(j-1) = (S_0^(j), S_(w-1..1)^(j-1))
15.       END FOR
16.    END IF
17.    s^(e) = 0
18. END FOR
```

Figure 2. The Multiple Word Radix-2 Montgomery Multiplication Algorithm.

2.2. Scalable Montgomery Algorithm

The fundamental operation of the RSA public key cryptosystem is modular exponentiation, achieved by repeated modular multiplications. The Montgomery modular multiplication [17] has been widely used to speed up the multiplication, squaring and exponentiation. There are many different extended algorithms [11] and software implementations based on the Montgomery algorithm, and the high-radix Montgomery modular exponentiation has been implemented in reconfigurable hardware [3] and optimised for specific reconfigurable architecture on RSA cryptosystems [6].

The reusability and scalability of the Montgomery multiplier have been investigated in [23] such that the design is no longer restricted to a fixed precision. The input operands are partitioned into multiple words. Figure 2 shows the algorithmic description of the Radix-2 Montgomery multiplication. The operand Y (multiplicand) is scanned word-by-word and the operand X (multiplier) is scanned bitwise. The loop index n is the bit width of the prime number and e is the number of partitions in the operand Y.

The scalable design of Montgomery multiplication has been implemented as a coprocessor in reconfig-

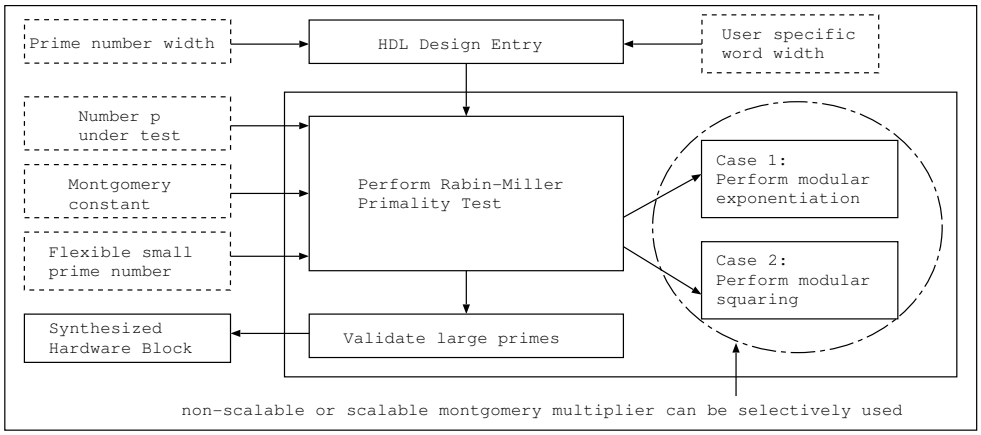

Figure 3. The design flow of the scalable prime number validator.

urable RSA System-on-Chip building block in [8]. Furthermore, a high-radix design of scalable modular multiplier has also been discussed in [22]. We observe that the scalable design is particularly useful for very large precision such as the prime test.

3. Design Flow

In this section we present the design flow for our scalable prime number validator as shown in figure 3. The input to the system includes user-specified constraints, such as the bit-width of the input prime number and the word-width which is required for the scalable multiplier, while the output from the system is a synthesizeable hardware block that can be embedded in different cryptographic designs. This flexible design flow enables upgrade of existing security systems with small overhead.

The major features of the proposed design include:

- A variable compile-time prime number validator in which user can easily update the complexity of the system by controlling the prime width.
- The variable input small primes determine the accuracy and performance of the system.
- The user-specified word-width controls the modular operation taken in the hardware.
- Users are able to select different Montgomery architectures for hardware implementation and thus enhance rapid prototyping.

3.1. Speeding Up Computation

In this section, we show how we improve the hardware design of the standard operation by non-scalable and scalable Montgomery operations. The Rabin-Miller algorithm is first implemented using standard multiplication and exponentiation with a sequential modulo-multiply. We observe that one of the bottlenecks of the Rabin-Miller algorithm is the modular exponentiation in testing the two valid checks in Section 2.

Montgomery algorithm is widely used for modular multiplication and exponentiation. It requires a constant at the start of algorithm to facilitate conversion between standard and Montgomery space. The constant c is $(2^{(2\times(width(p)+2))} \% p)$ where $width(p)$ is the bit-width of the prime number p under test. In this paper, this constant c has been precomputed and saved in hardware. Figure 4 shows the pseudo-code of the hardware implementation. Note that the statements embraced by $parallel\{...\}$ mean that they are executed concurrently in hardware.

3.2. Montgomery modular multiplication

Current modular multiplication approaches are mostly based on the Montgomery algorithm [17]. The simpler combinational logic used in this design reduces the critical path and thus accelerates the calculation. An efficient hardware implementation has been presented in [7]. Our modular multiplication component is built based on the design in [3].

The basic idea of the Montgomery algorithm is to multiply two integers modulo M, in other words $(A \times B \bmod M)$ without division by M. We first use the generated Montgomery constant c to transform the integers into m-residues and compute the multiplication with these m-residues. Finally, we transform this result back to the normal representation. Note that modular multiplication is used in modular exponentiation, since it is beneficial if we compute a series of multiplications in the transformed domain, the Montgomery space. Figure 4 shows the pseudo-code of the hardware description. For example, "$q = S + (B_0 \ ? \ A : 0)$" checks if the LSB of B is true, then "$q = S + A$" else "$q = S$". Figure 5 shows an example of code-level optimisation which has greatly reduced the overall number of cycles for more than 50% for the design using non-scalable Montgomery multipliers. In Section 4, we

```
Algorithm for computing S = A × B mod M

    parallel {
1.      S = 0
2.      j = width(A) + 2
    }
3.  FOR i = j to 0
        parallel {
4.        q = S + (B_0? A : 0)
5.        S = shiftRight(q + (q_0? M : 0))
6.        B = shiftRight(B)
        }
7.  END FOR
8.  RETURN S
```

Figure 4. Generating Montgomery modular multiplication S.

```
Pre-optimisation
1.  FOR i = j to 0
2.      q = S + (B_0? A : 0)
3.      S = shiftRight((S + (q_0? M : 0) + (B_0? A : 0))
4.      B = shiftRight(B)
5.  END FOR

Optimisation
1.  #define q (S + (B_0? A : 0)
2.  FOR i = j to 0
        parallel {
3.        S = shiftRight(q + (q_0? M : 0))
4.        B = shiftRight(B)
        }
5.  END FOR

Modular squaring (both scalable and non-scalable)
1.  Mont_mod_mult_sq(R) {
        // transform R into Montgomery space
2.      R = MontgomeryModularMulti[0](c, R)
        // perform squaring
3.      R = MontgomeryModularMulti[0](R, R)
        // transform the final result into normal space
4.      R = MontgomeryModularMulti[0](1, R)
    }
```

Figure 5. Hardware optimisation for modular multiplication and squaring.

introduce the scalable design and its optimisation using embedded memory.

3.3. Montgomery modular exponentiation

Figure 6 shows the algorithm for calculating the modular exponentiation using the Montgomery algorithm. This algorithm is not limited to the input bitwidth and is suitable for replacing the standard sequential modulo-multiplier. The values of X and 1 are first transformed into Montgomery space by using the Montgomery constant c. Since there is no data dependency between the modular squaring and multiplying operation in line 6 and line 7, both operations are put into separated hardware and execute in parallel. The final result is transformed back to the standard domain for the Rabin-Miller primality test.

The datapath of the modular exponentiation unit is depicted in figure 7. Two parallel multipliers have been deployed in this unit together with four temporary storages, P_0, P_1, Z_0 and Z_1. The inputs to this

```
Algorithm for computing S = X^E mod M

1. P[2] = 0, Z[2] = 0;
   parallel {
    // Apply two parallel multipliers
2.  P[0] = P[1] = MontgomeryModularMulti[0](c, 1)
3.  Z[0] = Z[1] = MontgomeryModularMulti[1](c, X)
4.  j = width(E) - 1
   }
5. FOR i = j to 0
     parallel {
6.    P[!i_0] = E_0?
          MontgomeryModularMulti[0](P[i_0],Z[i_0]):P[i_0]
7.    Z[!i_0] = MontgomeryModularMulti[1](Z[i_0], Z[i_0])
     }
8.   E = shiftRight(E);
9. END FOR
10.RETURN S = MontgomeryModularMulti[0](1, P[i_0])
```

Figure 6. Generating Montgomery modular exponentiation S (both scalable and non-scalable).

unit are X and E which are stored in registers or memory depending on the architecture of the multiplier. The pre-stored Montgomery constant c is used for the first step calculation in figure 6. The control path, other temporary storage and memory decoding unit are not shown in this figure.

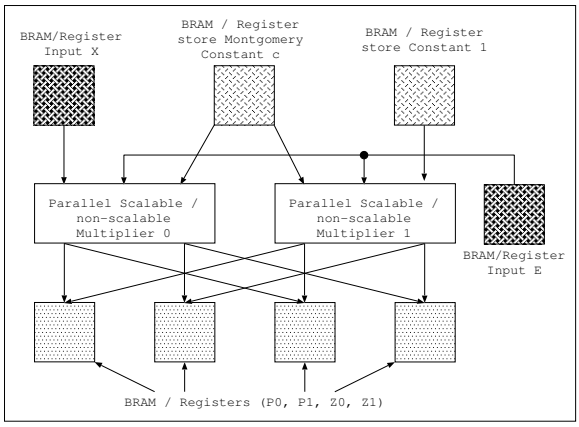

Figure 7. The architecture of the Exponentiation unit using two parallel Montgomery multipliers and multiple Block-RAMs.

4. Scalable modular multiplier

The previous section presents the general design flow of primality test using non-scalable multiplier. In this section, we present the mapping of scalable Montgomery multiplication as shown in Section 2 into technology independent hardware.

The scalable Montgomery algorithm MWR2MM is shown in figure 2 for multiplying X and Y. The general idea is to repeatedly multiply X_i, the i^{th} bit of X, with Y. The parallelism of the design can be

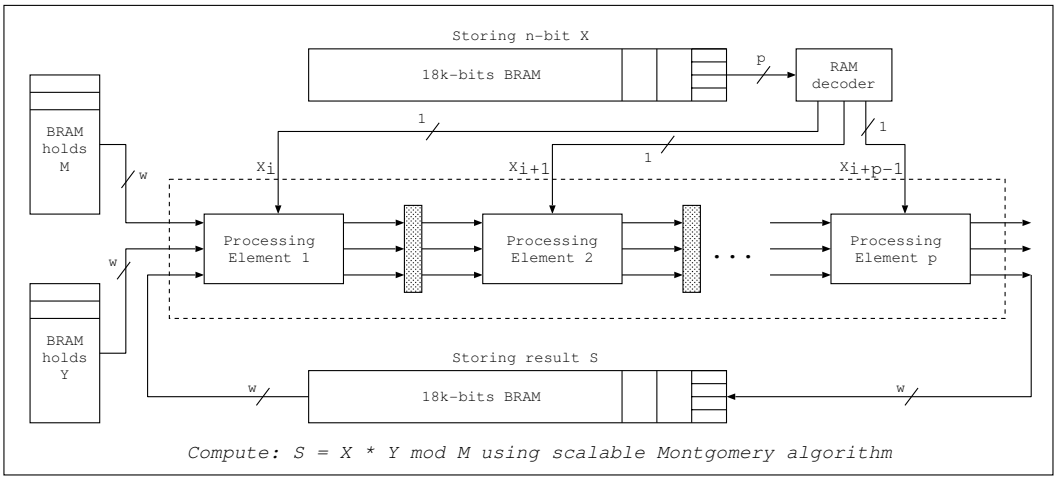

Figure 8. The architecture of the scalable Montgomery multiplier.

explored by applying multiple processing elements (PEs) to calculate the X_i values in parallel, since there is no data dependency between these calculations. The datapath of the multiplier using p PEs is shown in figure 8. We can see that the RAM decoder produces the p-bits, $X_i, X_{i+1}, \ldots X_{i+p-1}$ and feeds these bits into p PEs. These signals are valid for j cycles and the memory decoder then extracts the next p-bits from the memory storing X.

For the scalable version, the modular multiplier is replaced by the one described in figure 2. Note that the number of clock cycles and the performance of the scalable MWR2MM algorithm depends on the number of bits of the number under test and the user-specified partition size which is the word-length. The words extracted from Y, M are serially put into the first PE and are then pipelined through the other PEs. Figure 10 shows the data dependency involved in the multiplication. In this simplified figure, only three PEs are shown. Note that two intermediate pipeline registers are installed between every two PEs. We can see that at time t_0 and t_1 in figure 10, only the first PE, PE_1, operates while other PEs are stalled.

In the cycle when $j = 0$, the PE element determines the addition of M for the next j cycles within this PE element. We can refer to line 4-10, and line 12-15 in figure 2 for more details. The exact datapath of a single PE is described in figure 9. We use a multiplexor to select and store $S_0^{(0)}$, the bit-0 of the lowest word in S, in a register only when $j = 0$. The input S_j is latched for computing the S_{j-1} in the next cycle as shown in line 9 and 14 of figure 2.

We retarget the scalable Montgomery multiplier in FPGA and explore the available resources such as embedded memory. For instance, if we use a shift register for n-bit data, the hardware usage is linearly proportional to the input bit-width. In order to save area, our design stores the n-bit data into Block-RAM (BRAM) which holds up to 18k-bit data. Together

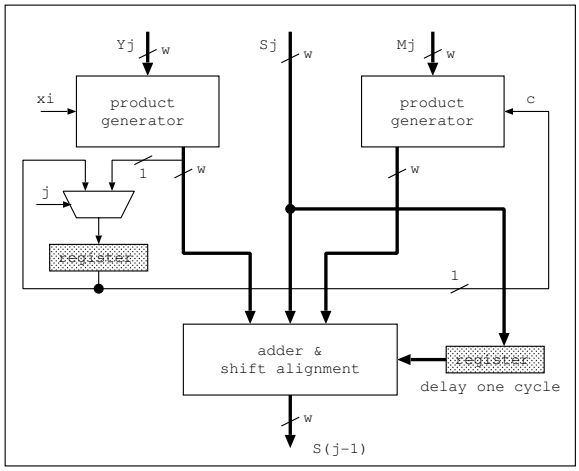

Figure 9. The architecture of a single Processing Element (PE).

with all the temporary storage, each design takes 20 BRAMs which is half of the available BRAMs in XC2V1000 FPGA chip. In figure 8, each data bit in BRAM-X is decoded for a PE while in each cycle, the j^{th} data in BRAM-Y and BRAM-M are stored in the first PE and propagated through the pipeline registers. The outputs of $PE_1, \ldots PE_{p-1}$ are stored in the intermediate registers, so that in each cycle only the last PE, PE_p, is responsible to update the content in BRAM-S.

5. Hardware Design Generator

This section presents the $GenDesign$ tool which we develop for transforming user-constraints into Hardware Description Language (HDL) format, as well as implementing hardware designs. $GenDesign$ is coded in C. Developers can easily embed the generated validator into their security designs. This genera-

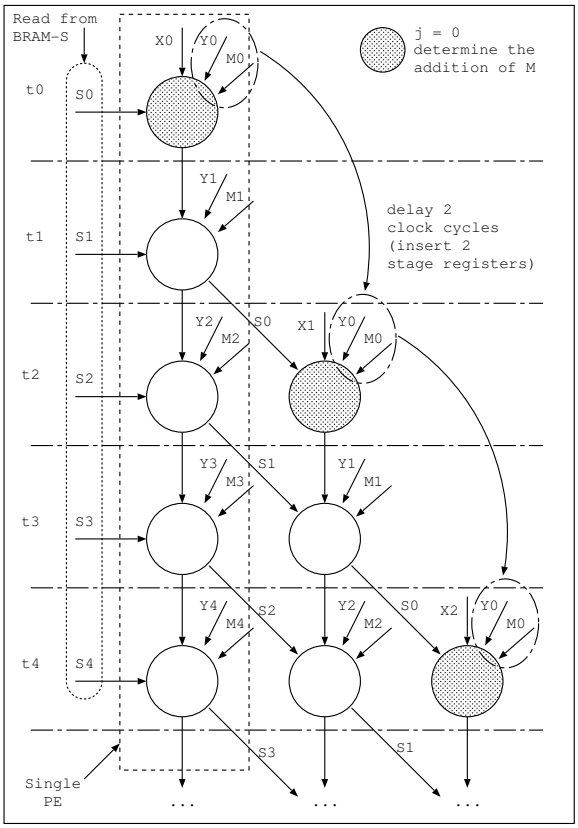

Figure 10. The dependency graph of the computation of scalable Montgomery multiplier using three processing elements.

tor enhances designer productivity and exploits available resources from the rapid-prototyping platform.

The objective of this design generator is to accelerate the overall design flow from specification to final implementation. Using this tool, designers can specify architectural-specific description of their designs. The predefined libraries consist of scalabe and non-scalable Montgomery operators based on the Rabin-Miller algorithm. The existing libraries are developed in Handel-C [4], and with slightly modification, the design generator is able to support other HDL output format for synthesis.

$GenDesign$ enables customisable designs such that users can have full control to the output HDL file. Two major operations in the design generator are the architectural selection routine and RAM decoding routine. In our design, the width of most internal signals are dependent on the prime number bit width. Second, user-specified word width controls the iteration of the inner while loop in the scalable Montgomery multiplier (the value e in figure 2). Third, besides the predefined small prime numbers, users are able to add or remove small prime numbers of the Rabin-Miller algorithm in the final HDL output. Consider step 14 in figure 2 as an example:

$$S^{(j-1)} = (S_0^{(j)}, S_{w-1..1}^{(j-1)})$$

the predefined libraries provide the exact implementation of a single PE for this particular bit concatenate operation. The bits in the j^{th} and $(j-1)^{th}$ word from S are concatenated using an internal register. $GenDesign$ is then used to expand the design into p parallel PEs.

6. Results

Our designs can be implemented on any FPGA-based prototyping platform. For small size problems, we select the RC200E development board as the testing platform which contains the Xilinx XC2V1000-4FG456C FPGA chip. For large size problems, the RC2000 development board which has a Virtex II XC2V6000-4FF1152 FPGA has been selected for implementation. We also place and route our designs on a low cost Spartan XC3S2000-4-FG900 chip. The generated designs are synthesized by using Celoxica DK2 and PDK tools [4]. The FPGA chip is then configured as a prime number validator.

Results are divided into two parts: Performance and Area. Table 1 shows the performance and area comparisons between non-scalable Montgomery multiplication and scalable Montgomery multiplication. From the collected results, the non-scalable designs produce the fastest execution time with a linearly increase of area penalty, while the scalable designs achieve relatively small and stable increase in both resource usage and critical path delay as the design scale grows. In the same table, we observe that by adopting multiple PE elements, the total number of cycles taken for the primality test has been much reduced while there is a slight effect to the delay path. As shown in [22], speed up of 2 to 3 times can be achieved when we use two or three processing elements in the scalable designs when compared to the design with only one PE. Our experimental results confirm this speed improvement.

We also study the tradeoff of speed and resource usage of the scalable Montgomery multiplier using Virtex chip in three different dimensions: degree of processing element parallelism, prime-size and word-size. In each case, one of these three parameters has been fixed and the results are shown in Table 2, 3 and 4. In Table 2, we observe that if we double the number of concurrent processing elements, the area is also doubled. As our scalable architecture extensively uses BRAM and minimises other components in the datapath, the effect of area increase the prime-size is slight. In Table 3, there are 8 PEs, the comparison of different prime-size for two word sizes is shown. With the use of memory decoding components, the complexity of addressing has been much reduced, resulting in a relatively slow increase in area usage. Table 4 shows the effect of changing the word-size with

	Virtex II - XC2V1000 / XC2V6000				*Spartan - XC3S2000*			
Prime size	128-bit	256-bit	512-bit	1,024-bit	128-bit	256-bit	512-bit	1,024-bit
	Non-scalable design							
Area (slices)	2,630	5,140	10,153	20,131	2,630	5,140	10,153	20,131
Clock cycle (ns)	36.75	49.51	80.72	122.34	43.67	62.35	95.01	162.24
Performance (ms)	0.64	3.34	21.49	129.28	0.76	4.21	25.29	171.45
	Scalable design (8 PE, word-size:w = 8-bit)							
Area (slices)	2,651	2,726	2,768	2,872	2,622	2,700	2,736	2,842
Clock cycle (ns)	31.66	30.78	31.07	32.62	40.38	34.72	39.68	37.75
Performance (ms)	20.69	109.37	734.39	5478.14	26.39	123.36	937.82	6338.95
	Scalable design (32 PE, word-size:w = 8-bit)							
Area (slices)	9,064	9,144	9,192	9,333	9,019	9,092	9,146	9,283
Clock cycle (ns)	30.17	29.81	29.47	31.03	40.87	40.61	38.94	39.03
Performance (ms)	12.70	56.07	286.18	1776.80	17.20	76.38	378.11	2235.08

Table 1. Primality test comparison.

respect to the overall performance. Finally, we draw our summary in figure 11.

	8 PE	16 PE	32 PE	64 PE
Prime size	p = 1,024-bit			
Area (slices)	1,165	2,193	4,278	8,636
Clock cycle (ns)	20.43	27.24	28.36	34.11
Performance (ms)	3.35	2.48	1.55	1.25
Prime size	p = 4,096-bit			
Area (slices)	1,198	2,253	4,404	8,864
Clock cycle (ns)	19.65	23.85	23.03	26.73
Performance (ms)	47.68	29.82	15.23	9.83

Table 2. Processing Element comparison with word size (w = 8).

Prime-size (bit)	1,024	2,048	4,096	8,192
Word size	w = 8-bit			
Area (slices)	1,165	1,197	1,198	1,230
Clock cycle (ns)	20.43	18.91	19.65	20.28
Performance (ms)	3.35	11.78	47.68	194.11
Word size	w = 32-bit			
Area (slices)	3,581	3,628	3,634	3,673
Clock cycle (ns)	32.43	38.77	31.31	27.20
Performance (ms)	1.73	7.01	20.54	67.72

Table 3. Prime-size comparison using Scalable Montgomery Multiplier with 8 Processing Elements (8 PEs).

Word-size (bit)	w=4	w=8	w=16	w=32
Prime size	128-bit			
Area (slices)	762	1,110	1,939	3,499
Clock cycle (ns)	20.98	18.31	24.02	34.19
Performance (ms)	0.14	0.08	0.07	0.08
Prime size	1,024-bit			
Area (slices)	828	1,165	1,985	3,581
Clock cycle (ns)	22.08	20.43	24.23	32.43
Performance (ms)	6.89	3.35	2.19	1.73

Table 4. Word-size comparison for 128-bit and 1,024-bit Scalable Montgomery Multiplication using 8 Processing Elements (8 PEs).

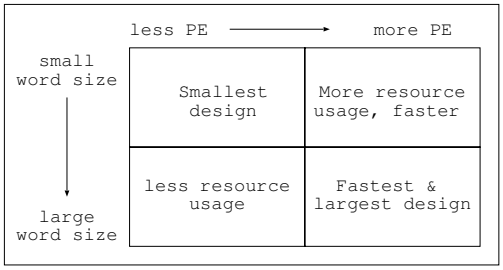

Figure 11. The analysis of the scalable Montgomery multiplier.

The design of 16k-bit primality test can be fitted into twenty 18k-bit BRAMs, which means a single XC2V1000 or a low cost Spartan XC3S2000 chip can accommodate a 32k-bit test. Since memory is the major limitation of the scalable multiplier, we can use off-chip memory or cascading multiple FPGAs for testing the Mersenne numbers. Another simple method to test 2,048 bits prime number X is to partition the input number into 2 parts: A, B and each part contains 1,024 bits. We assume that the 1,024 bit hardware validation has been implemented. We can use simple algebraic and divide-and-conquer methods to test this number. For instance, $X \times 2^n = A \times 2^{n/2} + B$ and $(X \times 2^n)^2 = (A \times 2^{n/2} + B)^2 = A \times A \times 2^n + 2AB \times 2^{n/2} + B^2$. Since modular arithmetic is commutative, associative and distributive, again we could use this method to perform primality test on very long integers.

The major overhead of the scalable multiplier is the two stall operations happened to every PE element. It means that if we apply 32 PE elements in a 128-bit test, $(2 \times 32 \times (128/32))$ stall cycles have been wasted. However, this effect is smaller when the input numbers are larger. Our flexible design scheme can fit different problem objectives from small Elliptic Curve Cryptography, medium RSA to large Mersenne number test.

7. Conclusions

This paper presents a scalable architecture for prime number validation which targets reconfigurable hardware. Two significant algorithms have been discussed: Rabin-Miller and Montgomery algorithms. The Rabin-Miller Strong Pseudoprime Test has been successfully mapped into hardware. Several modular multipliers have been developed as libraries in the design generator, $GenDesign$. In particular, the Montgomery modular multiplication is the core of other modular operations used in the primality test. The scalability and parallelism of this design have been explored for very large prime numbers. We systematically implement the design for different bit lengths on reconfigurable devices using both scalable and non-scalable multipliers, and study the trade-off amongst different designs. Our architectures and tools appear to be useful for exploring efficient designs for use in embedded systems.

On-going and future developments include: (1) The extension of this scalable architecture across multiple reconfigurable hardware device enables the division of the original large problem and expands the available memory space for validating Mersenne numbers. (2) Run-Time Reconfiguration (RTR) techniques can be also applied for the validation. The implemented design can run-time select different algorithms such as Fermat test or the sieve of Eratosthenes method [20].

8. Acknowledgment

The support of Celoxica, Xilinx, the Croucher Foundation and UK EPSRC (Grant number GR/R 31409, GR/R 55931, GR/N 66599) is gratefully acknowledged.

9. References

[1] F. Arnault, *Rabin-Miller Primality Test: Composite Numbers Which Pass It*, Math. Comput. 64, 355-361, 1995.

[2] M. Agrawal et al, *Primes in P*, ITT Kanpur, August, 2002.

[3] T. Blum and C. Paar, *High Radix Montgomery Modular Exponentiation on Reconfigurable Hardware*, IEEE Transactions on Computers, vol. 50, no. 7, pp.759-764, July, 2001.

[4] Celoxica: *http://www.celoxica.com/*

[5] R. Crandall and C. Promerance, *Prime Numbers - A Computational Perspective*, Springer, 2001.

[6] A. Daly and W. Marnane, *Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable logic*, International Symposium on Field-Programmable Gate Arrays, pp. 40-49, 2002.

[7] S. E. Eldridge and C. D. Walter, *Hardware Implementation of Montgomery's Modular Multiplication Algorithm*, IEEE Transactions on Computers, vol. 42, no. 7, pp. 693-699, 1993.

[8] V. Fischer and M. Drutarovsky, *Scalable RSA Processor in Reconfigurable Hardware - a SoC Building Block*, XVI Conference on Design of Circuits and Integrated Systems (DCIS), pp. 327-332, November, 2001.

[9] *The Greatest Internet Mersenne Prime Search*: "http://www.mersenne.org/"

[10] M. Joye, P. Paillier and S. Vaudenay, *Efficient Generation of Prime Numbers*, Cryptographic Hardware and Embedded Systems, pp. 340-354, 2000.

[11] C.K. Koc, T. Acar and B. S. Kaliski, *Analyzing and Comparing Montgomery Multiplication Algorithms*, IEEE Micro, vol. 16, no. 3, pp. 26-33, 1996.

[12] T. K. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu and N. Dulay, *Compiling policy descriptions into reconfigurable firewall processors*, IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 39-48, 2003.

[13] K. Leung et al, *FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor*, IEEE Symp. on FCCM, pp. 68-76, 2000.

[14] C. Lu et al, *Implementation of fast RSA key generation on smart cards*, ACM symposium on Applied computing, pp. 214-220, 2002.

[15] "http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html"

[16] M. Meerwein et al., *Embedded Systems Verification with FPGA-Enhanced In-Circuit Emulator*, International Symposium on System Synthesis, 2000.

[17] P. Montgomery, *Modular Multiplication without Trial Division*, Math. of Computation, vol. 44, pp. 519-521, 1985.

[18] G. L. Miller, *Riemann's Hypothesis and Tests for Primality*, Journal of Computer Systems Science, Vol. 13, No. 3, pp. 300-317, Dec, 1976.

[19] M. O. Rabin, *Probabilistic Algorithm for Primality Testing*, Journal of Number Theory, Vol. 12, pp. 128-138, 1980.

[20] H. Riesel, *Prime Numbers and Computer Methods for Factorization*, Progress in Maths., vol. 126, 1994.

[21] B. Schneier, *Applied Cryptography*, John Wiley, 1996.

[22] A. F. Tenca, G. Todorov and C. K. Koc, *High-Radix Design of a Scalable Modular Multiplier*, Cryptographic Hardware and Embedded Systems, pp. 189-205, May, 2001.

[23] A. F. Tenca and C. K. Koc, *A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm*, IEEE Transactions on Computers, Vol 52, No. 9, pp. 1215-1221, September, 2003.