Parametric Encryption Hardware Design

Adrien Le Masle[1], Wayne Luk[1], Jared Eldredge[2], and Kris Carver[2]

[1] Department of Computing, Imperial College London, UK
{al1108,wl}@ic.ac.uk
[2] BlueRISC, Inc, Amherst, MA, USA
{jared,kris}@bluerisc.com

Abstract. We present new scalable hardware designs of modular multiplication, modular exponentiation and primality test. These operations are at the core of most public-key crypto-systems. All the modules are based on an original Montgomery modular multiplier. Our multiplier is the first Montgomery multiplier design with variable pipeline stages and variable serial replications. It is 8 times faster than the best existing hardware implementation and 30 times faster than an optimised software implementation on an Intel Core 2 Duo running at 2.8 GHz. Our exponentiator is 2.4 times faster than an optimised software implementation. It reaches the performance of a more complex FPGA design using DSP blocks which is the fastest in the literature. Our prime tester is 2.2 times faster than the software implementation and is 85 times faster than hardware implementations of the same algorithm with only 60% area overhead.

1 Introduction

Most public-key cryptographic algorithms consist of two main stages: the key generation which requires the ability to generate large prime numbers and the encryption/decryption part.

Modular exponentiation is a common operation used by several public-key crypto-systems, such as the Diffie-Hellman key exchange protocol and the Rivest, Shamir and Adleman (RSA) encryption scheme. It is also, together with modular multiplication, the core of common prime tests such as the Rabin-Miller strong pseudo-prime test.

As security is becoming increasingly important, algorithms such as RSA need more and more bits for the keys used to be secured. For data that need to be protected until 2030, a 2048 bit key is recommended whereas a 3072 bit key is recommended for beyond 2031 [2]. This creates a need for scalable designs working with any bit-width.

Many new algorithms and improvements of existing algorithms for modular multiplication have been presented during the last decade [10]. This led to many hardware implementations of modular multiplication [3,4,6,8,9,11], modular exponentiation [3,7,9,11,12,13], and primality testing [5]. Most implementations target Field Programmable Gate Arrays (FPGAs) which offer rapid-prototyping platforms to compare different designs and can be reprogrammed as needed.

As FPGAs are quickly increasing in size, it is becoming more of a challenge to fully cover the design space available for a given budget. This introduces the need for parametric designs capable of exploring the entire design space, especially in terms of the speed-area trade-off.

This paper presents parametric hardware designs of modular multiplication, modular exponentiation and primality testing. Our main contributions include:

- A new parametric Montgomery multiplier design with variable pipeline stages and variable serial replications
- A modular exponentiator design based on our Montgomery multiplier
- A Rabin-Miller prime tester design using both our multiplier and our exponentiator
- An implementation of the proposed designs on Xilinx Virtex-5 FPGAs and a comparison of their performance in terms of speed and area with the main existing implementations.

Our Montgomery multiplier implementation is 8 times faster than the best existing hardware implementation [13] and 30 times faster than an optimised software implementation on an Intel Core 2 Duo running at 2.8 GHz. Our exponentiator is 2.4 times faster than an optimised software implementation and reaches the performance of a more complex FPGA design using DSP blocks [12] which is also the fastest design in the literature. Our prime tester is 2.2 times faster than the software implementation and is 85 times faster than hardware implementations of the same algorithm [5] with only 60% area overhead.

The rest of the paper is organised as follows. Section 2 explains the background relevant to our work. In section 3, we present the main challenges of our designs and how we address them. In section 4, we compare our FPGA implementations to the best existing implementations and highlight the scalability of our hardware architectures. Finally, section 5 concludes the paper.

2 Background

Most modular exponentiation algorithms require the ability to perform fast and area efficient modular multiplications. In [4], different algorithms for modular multiplication are compared in terms of the area-time product (AT). The Montgomery modular multiplication algorithm turns out to be the best with an AT complexity of $O(n^2)$.

Another important feature of a crypto-system is the ability to generate large prime numbers. Probabilistic methods for prime testing, determining whether or not a number is prime with a certain probability of error, are often used.

Modular Exponentiation. A simple but common algorithm for modular exponentiation is given in Alg. 1. To compute $X^E \bmod N$, the algorithm iterates on the bits of E from the least significant bit (LSB) to the most significant bit (MSB). At each iteration i, the variable $P_i = X^{2^i} \bmod N$ is squared modulo N to obtain $P_{i+1} = X^{2^{i+1}} \bmod N$. If $e_i = 1$, the accumulated product Z_i is multiplied by P_i modulo N, otherwise it remains the same. After n iterations, n being the bit-width of E, Z_n contains $X^E \bmod N$.

Algorithm 1. Exponentiation algorithm

```
**Input**: $X$,$E$,$N$ with $E = \sum_{i=0}^{n-1} e_i 2^i$, $e_i \in \{0,1\}$
**Output**: $Z_n = X^E \bmod N$
1 $Z_0 = 1$, $P_0 = X$
2 for $i = 0$ to $n-1$ do
3     $P_{i+1} = P_i^2 \bmod N$
4     if $e_i = 1$ then
5         $Z_{i+1} = Z_i.P_i \bmod N$
6     else
7         $Z_{i+1} = Z_i$
8 end
```

Algorithm 2. Simple Montgomery algorithm for modular multiplication

```
Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, $N = \sum_{i=0}^{n-1} n_i 2^i$, $(a_i, b_i, n_i) \in \{0,1\}^3$,
       $n_0 = 1$, $0 \leq A, B \leq N$
Output: $P = A.B.2^{-n} \bmod N$
1 $P = 0$
2 for $i = 0$ to $n-1$ do
3     $P = P + a_i.B$
4     $P = P + p_0.N$
5     $P = P \ div\ 2$
6 end
7 if $P \geq N$ then  $P = P - N$
```

Algorithm 3. Rabin-Miller strong pseudo-prime test

```
Input: $p = 2^r d + 1$ odd integer, set $P$ of $|P|$ first primes
Output: composite if $p$ is composite, prime if $p$ is probably prime
1 for $i = 0$ to $|P| - 1$ do
2     $a = P[i]$
3     if $a^d = 1 \ mod\ p\ or\ a^{2^j d} = -1 \ mod\ p\ for\ some\ 0 \leq j \leq r-1$ then
4         continue
5     else
6         return composite
7 end
8 return prime
```

Montgomery Modular Multiplication. A simple version of Montgomery modular multiplication is presented in Alg. 2. This algorithm iterates on the bits of A from the LSB to the MSB. At iteration i, $a_i.B$ is added to the accumulated product P. If P is odd, N is added to P. This does not change the result as the calculation is done modulo N. As N is odd (required by the algorithm), P becomes even and can be divided by 2 without remainder.

The drawback of the Montgomery algorithm is that it actually computes $A.B.2^{-n} \bmod N$, introducing an extra 2^{-n} factor which has to be eliminated. The common method to remove this factor is to convert the inputs in N-residue [7], to perform the modular multiplication and to convert back the result to a normal representation by Montgomery multiplying it by one.

Rabin-Miller Primality Test. In this paper, we consider a variant of the Rabin-Miller strong pseudo-prime test using a few number of primes as witnesses. This probabilistic test has a low probability of error. Algorithm 3 shows that the Rabin-Miller test relies on the ability to perform modular multiplications and exponentiations and is therefore a relevant application for our designs.

3 Design Flow

Our goal is to create parametric designs of Montgomery multiplication, Montgomery exponentiation and Rabin-Miller primality test. The benefits of such designs are that they can:

– adapt to the present and future needs in terms of key width
– explore a large design space in terms of speed and area, adapting to the accelerating advancement in FPGA development
– meet the requirements of various hardware encryption projects

Parametric Montgomery Multiplier Design. We choose a one-carry save adder (CSA) based Montgomery multiplier as the basic block of our design. A CSA is faster than a ripple-carry adder with no area overhead. Algorithm 4 from [4] is used. The diagram of a multiplier cell is given in Fig. 1. The bit-width of this cell is set as a design parameter.

We apply two techniques to our design: pipelining and serial replication. Pipelining improves the throughput of the design and serial replication improves

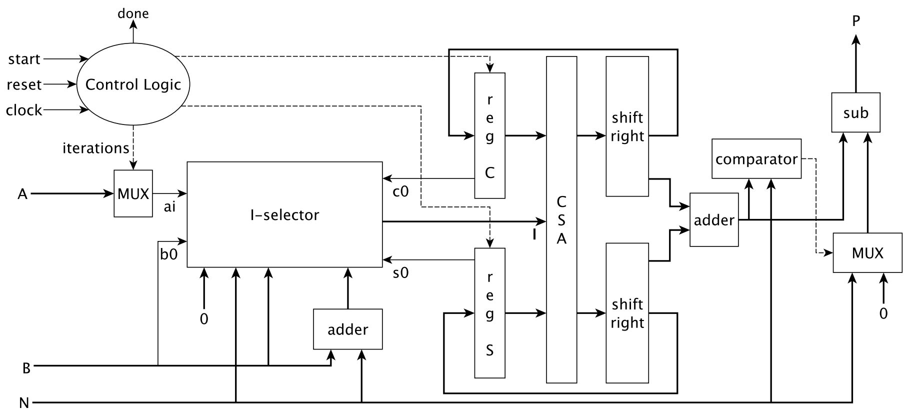

Fig. 1. Diagram of a Montgomery multiplier cell

Algorithm 4. Fast Montgomery algorithm for modular multiplication

```
   Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, $N = \sum_{i=0}^{n-1} n_i 2^i$, $(a_i, b_i, n_i) \in \{0,1\}^3$
   Output: $P = A.B.2^{-n} \bmod N$
 1 $S = 0$
 2 $C = 0$
 3 for $i = 0$ to $n-1$ do
 4     if $(s_0 = c_0)$ and $a_i = 0$ then $I = 0$
 5     if $(s_0 \neq c_0)$ and $a_i = 0$ then $I = N$
 6     if $(s_0 \oplus c_0 \oplus b_0) = 0$ and $a_i = 1$ then $I = B$
 7     if $(s_0 \oplus c_0 \oplus b_0) = 1$ and $a_i = 1$ then $I = B + N$
 8     $S, C = S + C + I$
 9     $S = S$ div $2$
10     $C = C$ div $2$
11 end
12 $P = S + C$
13 if $P \geq N$ then $P = P - N$
```

its latency. Three main challenges have to be addressed to design a parametric Montgomery multiplier using these techniques:

Challenge 1. Algorithm 4 cannot be easily parallelised due to the data dependencies between the consecutive values of S and C in the main loop. At iteration $i+1$, the values of S and C from iteration i are needed to compute the new value of I and the new values of S and C.

Challenge 2. To explore as much design space as possible, the bit-width, the number of replications and the number of pipeline stages should be parameters which can take any value.

Challenge 3. The control should adapt to the values of these parameters.

Consider Challenge 1. Figure 2 shows the basic structure of our pipelined design. Each Montgomery cell is a modified version of the basic cell presented earlier. We cope with Challenge 1 by allowing each basic cell to perform a consecutive part of the iterations. For this principle to work, the final addition and subtraction blocks are removed from this cell and the number of iterations performed by each cell becomes a parameter. The basic cell is added with the ability to load the S and C registers from the inputs. The current values of S and C are also available at the output of each cell.

Inside each pipeline block, the CSA can be replicated as many times as needed. The data dependencies problem prevents us from simply duplicating the CSA and perform several iterations in parallel. Instead, several CSAs along with the shift logic are put in series. This is equivalent to unrolling the loop of Alg. 4 $r-1$ times. The I-selector is also replicated as the values of I differ for each CSA and at each iteration. At equal frequencies, replication decreases the latency of the design by a factor of r, the total number of CSAs. The area overhead is less than r because only a part of the basic cell is replicated. In practice, replication

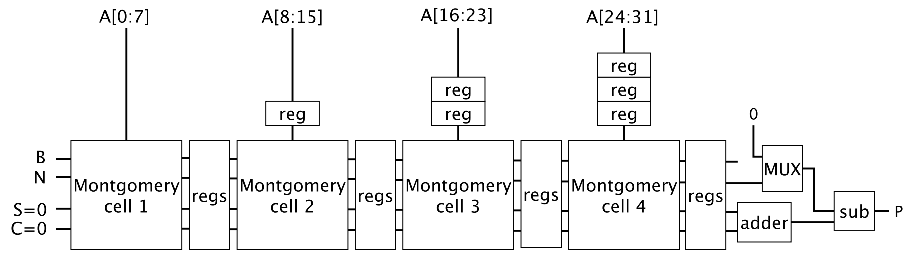

Fig. 2. Structure of a 32 bit pipelined Montgomery multiplier with 4 pipeline stages

also increases the critical path, reducing the maximum clock frequency at which the multiplier can run.

Consider Challenge 2. Allowing the bit-width (n) and the pipeline depth (p) to take any value makes it more difficult to divide the number of iterations between blocks. When n is not a multiple of p, each block cannot perform the same number of iterations. We address this challenge by adding an extra iteration to the first $n \bmod p$ pipeline blocks. This leads to $n \bmod p$ blocks computing $\lfloor n/p \rfloor + 1$ iterations, and $(n - n \bmod p)$ blocks computing $\lfloor n/p \rfloor$ iterations.

Inside a pipeline block, in order to allow the number of replications (r) to take any value, the result can be extracted from any CSA. This solution deals with the case when the number of iterations the cell has to perform is not a multiple of the number of replications.

Consider Challenge 3. A flexible pipeline control is implemented. This control deals with the updates of two types of registers: the registers between blocks and the triangular register array. The register triangular structure consists of arrays of registers controlled as FIFO queues. The inputs enter all the FIFOs at the same time when the `done` signal of the very first cell is triggered. The element at the head of the FIFO of a given cell leaves the queue when the corresponding cell has finished using it, that is when its `done` signal is triggered. In practice, extra registers store the position of the first empty slot in each FIFO, acting as pointers. When an element leaves the FIFO, the FIFO registers are updated accordingly (register i takes the value of register $i + 1$) and the corresponding pointer is decremented by 1. When an element enters the FIFO, the register indexed by the pointer is updated with the value of this element and the pointer incremented by 1.

Inside a pipeline block, the control logic manages the extraction of the result from the correct CSA, depending on the number of replications chosen and the number of iterations this particular block has to perform.

Let us consider an example summarizing this section with $n = 1024$, $p = 5$ and $r = 7$. As $\lfloor n/p \rfloor = 204$ and $n \bmod p = 4$, the first 4 pipeline blocks compute $204 + 1 = 205$ iterations and the last one computes 204 iterations. Our flexible pipeline control deals with this issue. Inside each block, we want 7 replications. In the first 4 pipeline blocks, we loop through the 7 CSAs 30 times ($\lceil 205/7 \rceil$). The result is extracted from CSA number 2 (205 mod 7) after the last iteration.

In the last pipeline block, we loop through the 7 CSAs 30 times ($\lceil 204/7 \rceil$). The result is extracted from CSA number 1 (204 mod 7) after the last iteration. This complex behaviour is managed by the control logic of each pipeline block.

Application to Modular Exponentiation. We use our modular multiplier to design a parametric hardware implementation of Alg. 1. The exponentiator uses the pipelining and replication capabilities of the multiplier description.

Two main problems have to be solved for this design to be efficient in terms of speed and area. First, the multiplier has to be optimally pipelined. We can show that the number of pipeline stages of the multiplier giving best performance for use with the exponentiator is equal to two. This is due to the fact that in Alg 1, P_{i+1} depends on P_i and Z_{i+1} depends on both P_i and Z_i. If we use more than two pipeline stages, the multiplier's pipeline cannot be kept full due to these data dependencies.

Second, integrating our multiplier in a bigger design can reduce its running frequency due to critical path problems. The latency of the adders and subtractors used in the multiplier would become a bottleneck for large bit-widths. To reduce the critical path, all the ripple-carry adders and the subtractors can be pipelined with any depth.

The design of our exponentiator is represented in Fig. 3. It has three main parameters: the number of multiplier's pipeline stages, the number of replications for the multiplier's pipeline cells, and the pipeline depth of the adders/subtractors. At each iteration, the current values of P and Z are stored in a RAM. The control logic manages the inputs to give to the multiplier and the data to write back. It also controls the multiplier.

Application to Primality Testing. We design a parametric Rabin-Miller prime tester based on both our multiplier and our exponentiator. The challenge is to use these two modules optimally, while keeping the design simple.

To save area, one single multiplier is shared by the exponentiator (to perform the multiplications needed to compute $a^d \bmod p$) and the prime tester (to perform the consecutive modular multiplications needed to compute the $a^{2^j d} \bmod p$). To improve the speed of the prime test, the exponentiator takes full advantage of the pipelining and replication features of the multiplier. However, it is not worth using the pipeline of the multiplier for the calculation of the $a^{2^j d} \bmod p$. It can be shown that the mean value of r in Alg. 3 is equal to two and that on average the multiplier is only used once directly by the prime tester at each iteration. Pipelining the computation of the $a^{2^j d} \bmod p$ would therefore make the design more complex with almost no performance benefit.

A diagram containing the important blocks, signals and connections of the prime tester is presented in Fig. 4. The values of the first prime numbers are stored in a ROM. At each iteration, the control logic selects the prime to use for the test and the inputs to give to the multiplier, to the comparator and to other intermediate registers. The control logic contains the state machine of the prime tester which controls the multiplier and the exponentiator.

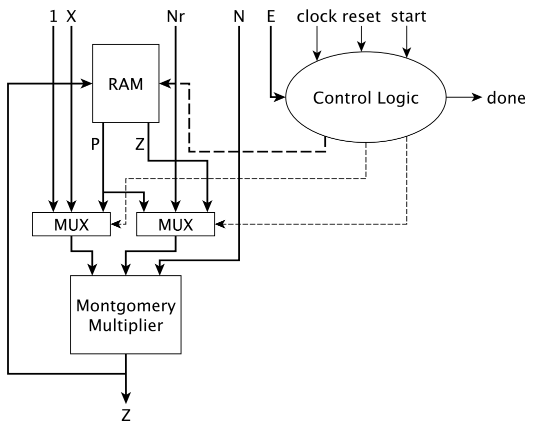

Fig. 3. Montgomery exponentiator

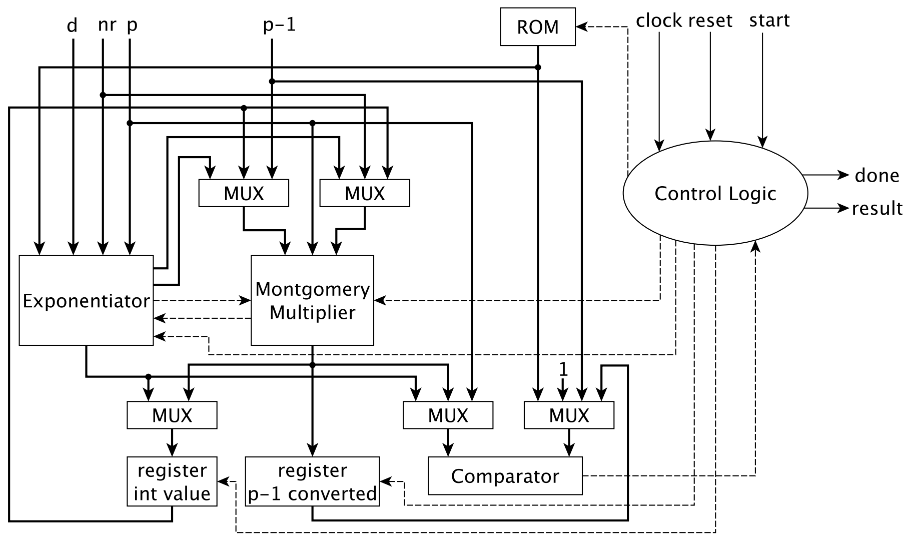

Fig. 4. Rabin-Miller prime tester

4 Results

Our three designs are implemented in Verilog. We synthesize our designs with Xilinx ISE 11.1 for Xilinx Virtex-5 FPGAs with "speed" as the optimisation mode and "normal" as the optimisation level. The results for the area and the maximum clock frequency are those given by the synthesis operation. When comparing the reported performance with other implementations, one should take into account that they do not all target the same FPGA. These results only give an idea of how our implementations perform compared to the best implementations in the literature. However, the scalability results are made accurate by implementing our designs on the same FPGA for most values of r and p.

Multiplier. Table 1 compares the execution time of our multiplier with other implementations for $n = 1024$ bits. For the software version, we report the mean and the standard deviation (σ) of the execution time for one million multiplications of random numbers. The execution time of our hardware multiplier only depends on p, r and the clock frequency. Our multiplier without any pipelining and replication is faster than most existing implementations with a runtime of 4.28 μs. It is also faster than the software implementation of modular multiplication using the very optimised GMP library on an Intel Core 2 Duo E7400 running at 2.8 GHz. For $p = 8$ and $r = 8$, our multiplier is 8 times faster than the best existing hardware implementation and 30 times faster than the optimised software implementation. We can still get better performance by increasing r and p if we target an FPGA with enough available area. Our design scales as the device scales and can therefore adapt to future FPGA families.

The results of Tab. 1 also show how our multiplier scales with p and r. Keeping r constant, when p doubles (from 1 to 2), the execution time is halved with 85% area overhead. This area overhead is less than 100% as the area of the adder and subtractor required at the output of the pipeline is not negligible, especially for small values of p and r. Keeping p constant, by increasing r from 1 to 4 a speedup of 2.5 times is achieved with less than 80% area overhead. Increasing the number of replications is less area-consuming than increasing the number of pipeline stages as a smaller part of the basic cell is duplicated. However, the

Table 1. Performance comparison of 1024 bit multipliers

Design	Device	Clock (MHz)	Area (LUT-FF pairs)	Ex. Time (μs)
Our design ($p = 8$, $r = 8$)	XC5VLX330T-2	99.13	206 982	0.18
Our design ($p = 2$, $r = 8$)	XC5VLX110T-3	110.98	59 337	0.59
Our design ($p = 2$, $r = 4$)	XC5VLX110T-3	149.55	42 429	0.87
Our design ($p = 1$, $r = 8$)	XC5VLX110T-3	102.63	31 925	1.27
Tang [13]	XC2V3000-6	90.11	N/A	1.49
Our design ($p = 1$, $r = 4$)	XC5VLX110T-3	150.23	20 593	1.72
Our design ($p = 2$, $r = 1$)	XC5VLX110T-3	226.31	23 436	2.28
Our design ($p = 1$, $r = 1$)	XC5VLX110T-3	239.74	13 671	4.28
GMP 4.2.4 [1] `mpz_mul/mpz_mod`	Intel Core 2 Duo E7400	2800	N/A	mean: 5.45 σ: 0.53
Oksuzoglu [11] (1020 bit)	XC3S500E-4FG320C	119	6 906	7.62
McIvor [8]	XC2V3000	75.23	23 234	13.45
Daly [6]	XCV1000	55	10 116	18.67
Amanor [9]	XVC2000E-6	49	8 064	21.00

Table 2. Time-Area products normalised to our design for 1024 bit multiplication

Design	Area (LUT-FF pairs)	Clock Cycles	Time × Area
McIvor [8]	23 234	1025	6.08
Daly [6]	10 116	1027	2.65
Amanor [9]	8 064	1027	2.11
Oksuzoglu [11] (1020 bit)	6 906	907	1.60
Our design ($p = 2$, $r = 8$)	59 337	66	1.00

increase in speed is less substantial due to the negative effect of replication on the maximum frequency.

Table 2 compares implementations in terms of the Time × Area product. For $p = 2$ and $r = 8$ our multiplier ranks first. It is interesting to see that $(p, r) = (2, 8)$ is a design point favouring speed over area. Our design benefits mainly applications with high speed requirements and large area available.

Exponentiator. Table 3 compares the execution time of our exponentiator with other implementations for $n = 1024$ bits. The pipeline depth of all ripple-carry adders and all subtractors is fixed to 8 in order to reduce the critical path. For the software version, we also report the mean and standard deviation for one million exponentiations of random numbers. For $p = 2$ and $r = 8$, our implementation running at 97.9 MHz is 2.4 times faster than the optimised software implementation using the GMP library on an Intel Core 2 Duo E7400 running at 2.8 GHz. Our exponentiator can also reach the speed of the best Montgomery modular exponentiator in the literature which uses DSP operations.

Prime tester. Table 4 compares the execution time of our prime tester with other implementations for $n = 1024$ bits. The pipeline depth of all ripple-carry

Table 3. Performance comparison of 1024 bit exponentiators

Design	Device	Clock (MHz)	Area	Ex. Time (ms)
Suzuki [12]	XC4VFX12-10SF363	200/400	7 874 LUT-FF + 17 DSP48	1.71
Our design ($p = 2$, $r = 8$)	XC5VLX110T-3	97.9	65 200 LUT-FF	1.74
Tang [13]	XC2V3000-6	90.11	14 334 slices + 62 multipliers	2.33
Our design ($p = 2$, $r = 2$)	XC5VLX110T-3	145.66	28 008 LUT-FF	3.88
GMP 4.2.4 [1] `mpz_powm`	Intel Core 2 Duo E7400	2800	N/A	mean: 4.23 σ: 0.12
Oksuzoglu [11] (1020 bit)	XC3S500E-4FG320C	119	6 906 LUT-FF + 20 multipliers	7.95
Our design ($p = 1$, $r = 2$)	XC5VLX110T-3	135.9	17 414 LUT-FF	8.14
Blum [3]	XC4000	45.7	13 266 LUT-FF	11.95

Table 4. Performance comparison of 1024 bit prime testers

Design	Device	Clock (MHz)	Area (LUT-FF)	Ex. Time mean/σ (ms)
Ours ($p = 2$, $r = 8$)	XC5VLX110T-3	86.7	64 817	2.01/0.741
Ours ($p = 2$, $r = 4$)	XC5VLX110T-3	87.1	41 970	3.54/1.31
GMP 4.2.4 [1] `mpz_millerrabin`	Intel Core 2 Duo E7400	2800	N/A	4.33/1.82
Ours ($p = 1$, $r = 4$)	XC5VLX110T-3	87.1	31 538	6.81/2.51
Ours ($p = 2$, $r = 2$)	XC5VLX110T-3	87.0	30 892	6.64/2.45
Ours ($p = 1$, $r = 1$)	XC5VLX110T-3	87.1	22 346	25.3/9.32
Cheung [5] (non-scalable design)	XC3S2000	6.1	40 262	171.45/_
Cheung [5] (scalable design 32 PE)	XC3S2000	25.6	18 566	2235.08/_
Cheung [5] (scalable design 8 PE)	XC3S2000	26.5	5 684	6338.95/_

adders and all subtractors is also set to 8. Unlike our other designs, the execution time of the prime tester depends on the number under test. We choose 10 000 random numbers and use them for all the experiments. We report the mean and the standard deviation. For $p = 1$ and $r = 1$, our prime tester is 6.8 times faster than Cheung's non scalable design and takes 1.8 times less area. It is 88 times faster than the fastest scalable design from [5] with only 20% area overhead. For $p = 2$ and $r = 8$, our design running at 86.7 MHz is 2.2 times faster than the GMP implementation on an Intel Core 2 Duo running at 2.8 GHz. It is 85 times faster than Cheung's non scalable design and only takes 1.6 times more area.

Our multiplier, our exponentiator and prime tester descriptions cover a huge design space. The exponentiator and the prime tester scale with p and r the same way as the multiplier except from one point: their speed cannot be increased by using more than two multiplier's pipeline stages as shown before. However, once this threshold is reached, increasing the number of replications remains relevant in order to increase the speed of these two designs.

5 Conclusion and Future Work

This paper presents a new parametric Montgomery multiplier design with variable pipeline stages and variable serial replications. It is 8 times faster than the best existing hardware implementation and 30 times faster than an optimised software implementation on an Intel Core 2 Duo running at 2.8 GHz. We design a modular exponentiation module based on our multiplier. It is 2.4 times faster than the optimised software implementation and reaches the performance of a more complex FPGA implementation using DSP blocks. A Rabin-Miller prime tester gathering the strengths of our modular multiplication and exponentiation modules is presented. It is 2.2 times faster than the software implementation and is 85 times faster than hardware implementations of the same algorithm with only 60% area overhead. Our three designs are scalable and their performance are only limited by the device used. As FPGAs are growing steadily, the parametric nature of our modules enables them to fully explore the design space available in any current and future project.

Current and future work includes extending our replication and pipelining methods to the exponentiator and the prime tester, making our parametric multiplier capable of reaching the very low-area end of the design space and developing tools that automate our replication and pipelining approaches.

Acknowledgments. The support of UK EPSRC and BlueRISC is gratefully acknowledged.

References

1. Gmp library manual, `http://gmplib.org/manual/`
2. RSA Labs article on RSA security, `http://www.rsa.com/rsalabs/node.asp?id=2004`
3. Blum, T., Paar, C.: High-radix Montgomery modular exponentiation on reconfigurable hardware. IEEE Trans. Comput. 50(7), 759–764 (2001)
4. Bunimov, V., Schimmler, M., Tolg, B.: A complexity-effective version of Montgomery's algorithm. In: Workshop on Complexity Effective Designs (2002)
5. Cheung, R., Brown, A., Luk, W., Cheung, P.: A scalable hardware architecture for prime number validation. In: IEEE Int. Conf. on Field-Programmable Technology, pp. 177–184 (2004)
6. Daly, A., Marnane, W.: Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic. In: ACM Symp. on FPGAs, pp. 40–49 (2002)
7. Fry, J., Langhammer, M.: RSA & Public key cryptography in FPGAs. CDC (2003)
8. Mclvor, C., McLoone, M., McCanny, J.: Fast Montgomery modular multiplication and RSA cryptographic processor architectures. In: 37th Asilomar Conf. on Signals, Systems and Computers, vol. 1, pp. 379–384 (2003)
9. Narh Amanor, D., Paar, C., Pelzl, J., Bunimov, V., Schimmler, M.: Efficient hardware architectures for modular multiplication on FPGAs. In: Int. Conf. on Field Programmable Logic and Applications, pp. 539–542 (2005)
10. Nedjah, N., de Macedo Mourelle, L.: A review of modular multiplication methods and respective hardware implementation. Informatica 30(1), 111–129 (2006)
11. Oksuzoglu, E., Savas, E.: Parametric, secure and compact implementation of RSA on FPGA. In: Int. Conf. on Reconfigurable Computing and FPGAs, pp. 391–396 (2008)
12. Suzuki, D.: How to maximize the potential of FPGA resources for modular exponentiation. In: Workshop on Crypt. Hardware and Emb. Sys., pp. 272–288 (2007)
13. Tang, S., Tsui, K., Leong, P.: Modular exponentiation using parallel multipliers. In: IEEE Int. Conf. on Field-Programmable Technology (FPT), pp. 52–59 (2003)