Imperial College London

Department of Computing

Parametric encryption hardware design

by

Adrien Le Masle

Submitted in partial fulfilment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

September 2009

Abstract

We present new hardware designs of modular multiplication, modular exponentiation and primality test. These operations are the core of most public-key cryptosystems. The idea is to gather the main strengths of existing designs in a highly parametric and new design. 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 parallelism by replication. To manage any number of pipeline stages, we create a flexible pipeline control treating the triangular register array of the pipeline as FIFOs. All the adders and subtractors of the multiplier can also be pipelined with any depth. The data dependencies in the Montgomery algorithm prevent us from simply duplicating the processing block and performing the computation in parallel. We cope with that problem by designing a replication control logic performing the iterations by consecutive blocks through carry-save adders put in series.

We develop a model highlighting the effects of our pipelining and replication methods on the throughput of a hardware design. Applied to our Montgomery multiplier, this model allows fast parametrisation and integration of the multiplier into larger designs.

Our model is evaluated against synthesis results. It gives relevant insights about the design space for our multiplier in terms of logic area, register area and throughput. The effect of each parameter on the speed and area taken by our modules is quantified. The multiplier, exponentiator and the prime tester are compared with existing software and hardware implementations. The implementation of our multiplier on a 150 MHz XC5VLX50T FPGA is more than 16 times faster than the optimised software implementation on a 2.8 GHz Core 2 Duo E7400 CPU. The exponentiator and prime tester achieve speedup of 1.5 times over their software implementations on the same CPU. The exponentiator is up to 22.3 faster than existing hardware implementations. The prime tester implementing the Rabin-Miller strong pseudoprime test is faster than existing designs of the same algorithm with speedups from 1.1 times up to 13.6 times with only 10% area overhead.

Acknowledgements

I would like to thank my supervisor, Prof. Wayne Luk, for his advice and invaluable support during this project.

I would also like to thank the many academics who have provided helpful advice and knowledge during this year and all my friends who have made this year unforgettable.

Finally, I would like to thank my family for their love and encouragement.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Most crypto-systems rely on the ability to perform the same few basic operations. They often 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 cryptosystems, 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 pseudoprime 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, [3] recommends a 2048 bit key whereas a 3072 bit key is recommended for beyond 2031. This creates a need for scalable designs working with any bitwidth.

Many new algorithms [4] and improvements of existing algorithms [5] for modular multiplication have been presented during the last decade. This led to many hardware implementations of modular multiplication [6], [1], [7], modular exponentiation [8], [9], [10] and primality testing [11]. Most implementations target Field Programmable Gate Arrays (FPGAs) which offer rapid-prototyping platforms to compare different designs and can be reprogrammed as needed.

Any hardware design is driven by three main constraints: speed, area and power. The relative importance of these constraints on one another is not fixed and depends on the application. If the main design goal is to do encryption/decryption of data at high throughput, speed will be put forward. For embedded application, the area and power constraints are often considered first.

The major problem of the existing encryption hardware designs is that they cannot fully explore the design space given by the large families of FPGAs available in the market, especially in terms of the speed/area tradeoffs. In fact, most of them consider the influence of only one parameter (the bitwidth, the number of pipeline stages) on these tradeoffs. Figure 1.1 shows the number of FPGA slices available in the different members of the Xilinx Virtex-5 family. It goes from a small and cheap FPGA with less than 5000 slices to a large and expensive chip with more than 50 000 slices. The vast range of area in a single FPGA family highlights the huge potential for design space exploration. This introduces the need for parametric designs capable of exploring this space.

In this report, we develop a parametric hardware design of modular multiplication, modular exponentiation and primality testing. Our design is based on the most interesting techniques used in the existing designs. Each module is highly parametric. The parameters proposed allow one to

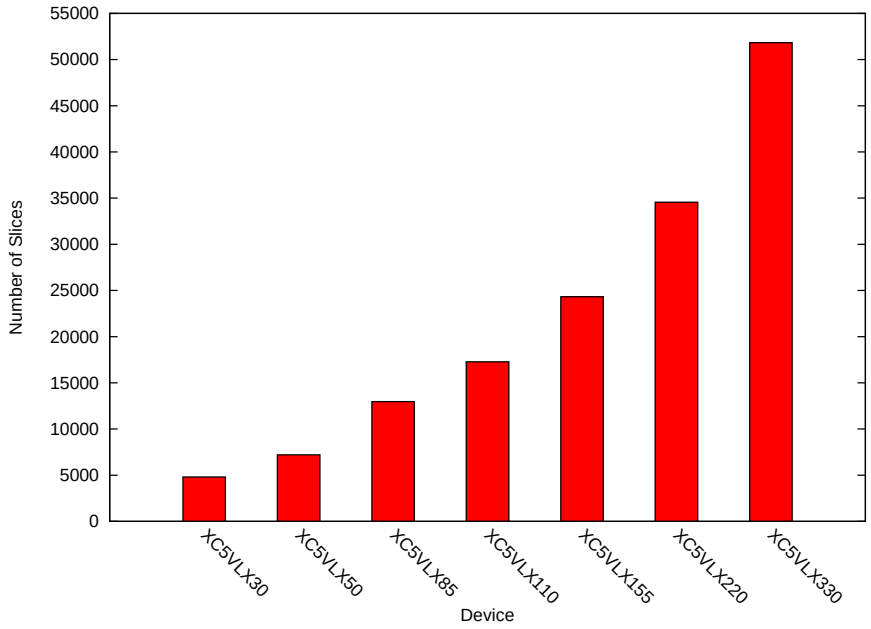


Figure 1.1: Slices available in different members of the Xilinx Virtex-5 FPGA family

independently study the influence of each technique on the global performance of the module, creating a great potential for design space exploration. We develop a model of our modular multiplier reducing its integration time in an existing or new hardware project.

1.2 Contributions

The major contributions are divided into the following parts from cryptographic designs to a model-driven performance tuning tool of these designs.

1.2.1 Parametric Montgomery multiplier

A new synthesizable hardware design of the Montgomery modular multiplication is presented (Chapter 3). This module is based on a parametric pipeline and the possibility to replicate the main processing element in each pipeline block. It supports any bitwidth, any number of pipeline stages and replications, covering a large design space. To improve the critical path of the multiplier, each of its adders and subtractors can be pipelined.

Designing such a parametric pipeline in an optimised way is challenging compared to a pipeline with a fixed depth. As a matter of fact, making the bitwidth and the number of pipeline stages totally parametric leads to the possibility to encounter pipeline blocks of different sizes and different latencies. To make this parametric pipeline work, we design a pipeline control treating the triangular register array as FIFOs of registers. It is capable of dealing with all these non-obvious configurations.

Our parametric way of replicating the main processing element in each pipeline block also presents some novel aspects. We demonstrate that the iterations performed in our Montgomery multiplier are dependent on each others. Hence replication does not simply consist in duplicating the hardware several times and perform the iterations in parallel. We put the replicated blocks in series and treat the successive dependent iterations by groups. A special control ensures that the right number of iterations is performed within each pipeline blocks and that the last result is extracted from the appropriate replicated block. These features make our design the first Montgomery multiplier with variable pipeline stages and variable parallelism by replication.

The degrees of freedom introduced by our parametric multiplier can be hard to master without appropriate tools. We address this issue by developing a model (Chapter 4) which finds the best number of pipeline stages and replications optimising the speed of the multiplier under a fixed area

constraint. This model proves to give relevant insights about the design space for our multiplier. It can be easily extended to any design based on the same pipelining and replication methods.

The influence of each parameter of the multiplier on the speed and area is studied in depth (Chapter 5). This study shows that the throughput of the multiplier can be increased as needed by increasing the number of pipeline stages. The only limitation is the area available in the FPGA. On the contrary, we show that the number of replications cannot be increased infinitely due to a critical path problem.

The implementation of our multiplier on a 150 MHz XC5VLX50T FPGA is more than 16 times faster than an optimised software version running on a 2.8 GHz Core 2 Duo E7400 CPU.

1.2.2 Parametric modular exponentiator

A modular exponentiator using our multiplier is designed (Chapter 3). It takes full advantage of the pipelining and replication features of the multiplier. We show that the optimum number of multiplier pipeline stages for use with the exponentiator is 2 due to data dependencies in the algorithm used. A potential improvement using the Montgomery multiplier with no final subtraction is integrated and evaluated.

The performance of this exponentiator in terms of speed and area is studied and compared to existing implementations (Chapter 5). The implementation of our exponentiator on a 150 MHz XC5VLX50T FPGA is 1.5 times faster than an optimised software version running on a 2.8 GHz Core 2 Duo E7400 CPU. A speedup of up to 22.3 is achieved over existing hardware implementations.

1.2.3 Rabin-Miller strong pseudoprime test hardware

Finally, a prime tester using both the modular multiplier and exponentiator is designed (Chapter 3). It is a new hardware version of the Rabin-Miller strong pseudoprime test. This algorithm is studied carefully to highlight its bottlenecks and find which parts are worth being optimised.

Our prime tester is evaluated against existing hardware and software implementations (Chapter 5). The implementation of our exponentiator on a 150 MHz XC5VLX50T FPGA is 1.6 times faster than an optimised software version running on a 2.8 GHz Core 2 Duo E7400 CPU. It is faster than existing hardware implementations of the same algorithm with speedups from 1.1 times up to 13.6 times with only 10% area overhead.

1.3 Thesis structure

This thesis is organized as follows. Chapter 2 presents the related work and background of this thesis. In Chapter 3, our parametric multiplier, exponentiator and prime tester designs are exhaustively described. Chapter 4 develops a model of our multiplier and describe our performance tuning software. In Chapter 5 our three designs together with our multiplier model are evaluated quantitatively and qualitatively.

Chapter 2

Background

This chapter covers background material to the project. Section 2.1 describes the principles of RSA encryption and key generation. Section 2.2 and 2.3 present the existing algorithms for modular multiplication and exponentiation and their most interesting hardware implementations. Finally, section 2.4 shows different methods used for primality testing, focusing on the Rabin-Miller strong pseudoprime test and section 2.5 sums up the chapter.

2.1 RSA

RSA (Rivest-Shamir-Adleman) is a public key encryption algorithm whose strength relies on the difficulty of solving the following number-theoretic problems:

- Given $N = p.q$ with p and q two large primes, try to factorise back N.
- Find P such that $P^E = C \mod N$ given integers N, E, and C such that $N = p.q$ where p and q are two large primes, $2 \leqslant E \leqslant N$ is coprime to $(p-1)(q-1)$ and $0 \leqslant C < N$

For integers of more than 1024 bits, solving these problems is computationally infeasible as of today.

The RSA algorithm consists of three parts: key generation, encryption and decryption.

2.1.1 RSA key generation

To compute the public key used for encryption and the private key used for decryption the following steps are performed [12]:

1. Pick two large prime numbers p and q and keep them secret
2. Calculate $N = p.q$
3. Calculate the Euler's totient function[1] $\phi(N) = (p-1)(q-1)$ and keep it secret
4. Calculate E and D such that $E.D = 1 \mod \phi(N)$

The public key is (E, N) and the private key (D, N).

2.1.2 Encryption/Decryption

The ciphertext C corresponding to the encryption of the plaintext P with the public key (E, N) is:

$$C = P^E \mod N \tag{2.1}$$

[1]Number of numbers less than N relatively prime to N.

To decrypt the ciphertext C, one has to own the private key (D, N). C can then be decrypted as follows:

$$P = C^D \bmod N \tag{2.2}$$

As a matter of fact, Euler proved that:

$$P = P^{k.\phi(p.q)+1} \bmod p.q \text{ for } k \in \mathbb{N} \tag{2.3}$$

As E and D are chosen so that $E.D = k.\phi(p.q) + 1$ (step 4 of the RSA key generation), we have:

$$\begin{aligned} C^D \bmod N &= (P^E)^D \bmod N \\ &= P^{E.D} \bmod N \\ &= P^{k.\phi(p.q)+1} \bmod N = P \end{aligned}$$

2.1.3 Key size

As said in section 2.1, as of today a key size of 1024 bits is enough for most application. More precisely in [3], the use of a 1024 bit key is stated to be secure at least until 2010. 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.

2.2 Modular exponentiation

Equations 2.1 and 2.2 show that the basic operation at the core of RSA encryption and decryption is the modular exponentiation. A simple but commonly used algorithm for modular exponentiation is given in algorithm 1.

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 1: Exponentiation algorithm

To compute $X^E \bmod N$ the algorithm iterates on the bits of E from the LSB to the 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. This step relies on the following formula:

$$X^E \bmod N = X^{\sum_{i=0}^{n-1} e_i 2^i} \bmod N \tag{2.4}$$

$$= \prod_{i=0}^{n-1} X^{e_i 2^i} \bmod N \tag{2.5}$$

After n iterations, n being the bitwidth of E, Z_n contains $X^E \bmod N$. In practice, if we do the test of e_i first, only two variables P and Z are needed.

2.3 Modular multiplication

Lines 3 and 5 of algorithm 1 show that the basic operation performed in modular exponentiation is modular multiplication.

2.3.1 Simple modular multiplication algorithm

A simple algorithm computing $A.B \bmod N$ consists of two steps:

1. Compute $R = A.B$
2. Reduce $P = R \bmod N$

The common methods used for this algorithm are the Ofman's and Booth's methods for multiplication (step 1) and the Barrett's method for modular reduction (step 2). These algorithms and their hardware implementations are detailed in [4].

2.3.2 Interleaved modular multiplication

Another way of performing a modular multiplication is to interleave the multiplication and reduction steps. The interleaved modular multiplication algorithm is given in algorithm 2.

Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, $N = \sum_{i=0}^{n-1} n_i 2^i$, $a_i,\ b_i,\ n_i \in \{0,1\}$, $0 \leqslant A, B \leqslant N$
Output: $P = A.B \bmod N$

1 $P = 0$
2 **for** $i = n-1$ **to** 0 **do**
3 $\quad P = 2.P$
4 $\quad I = a_i.B$
5 $\quad P = P + I$
6 $\quad$ **if** $P \geqslant N$ **then** $P = P - N$
7 $\quad$ **if** $P \geqslant N$ **then** $P = P - N$
8 **end**

Algorithm 2: Interleaved Modular Multiplication (adapted from [1])

This algorithm iterates on the bits of the operand A from the MSB to the LSB. For $i = k$ ($k < n-1$), at the beginning of the iteration:

$$P = (a_{n-1}.2^{n-(k+2)} + ... + a_{k+2}.2 + a_{k+1}).B \bmod N$$

P is multiplied by 2 and $a_k.B$ added to the result. Hence after line 5:

$$P = 2((a_{n-1}.2^{n-(k+2)} + ... + a_{k+2}.2 + a_{k+1}).B \bmod N) + a_k.B$$

Given that $0 \leqslant B \leqslant N$, a maximum of two subtractions is needed to get P back between 0 and N, giving after line 7:

$$P = (a_{n-1}.2^{n-(k+1)} + ... + a_{k+2}.2^2 + a_{k+1}.2 + a_k).B \bmod N$$

Therefore after n iterations, $i = 0$ and:

$$\begin{aligned} P &= (a_{n-1}.2^{n-1} + a_{n-2}.2^{n-2} + ... + a_0).B \bmod N \\ &= A.B \bmod N \end{aligned}$$

A hardware implementation of this algorithm is given in [1].

2.3.3 Montgomery modular multiplication

Algorithm

The Montgomery algorithm is another widely used method for modular multiplication. A simple version is presented in algorithm 3.

Unlike the interleaved modular multiplication, this algorithm iterates on the bits of A from the LSB to the MSB. At iterations i, $a_i.B$ is added to the accumulated product P. If P is odd, N is

Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, $N = \sum_{i=0}^{n-1} n_i 2^i$, $a_i,\ b_i,\ n_i \in \{0,1\}$, $n_0 = 0$, $0 \leqslant A, B \leqslant N$
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 ⩾ N then P = P − N
```

Algorithm 3: Simple Montgomery Algorithm for modular multiplication

added to P. This does not change the result as the calculation is done modulo N. As N is odd[2], P becomes even and can be divided by 2 without remainder.

For $i = 0$, after execution of line 4, P has a maximum bitwidth of $n+1$ and $P \leqslant 2N$. The shift right makes sure that P is n-bit width and $P \leqslant N$ after execution of line 5.

For $i = 1$, after execution of line 4, P has a maximum bitwidth of $n+2$ and $P \leqslant 3N$. The shift right makes sure that P is $(n+1)$-bit width and $P \leqslant 2N$ after execution of line 5.

For $i \geqslant 2$, after each iteration, the bitwidth of P remains $n+1$ bits and $P \leqslant 2N$. Hence only one subtraction is needed at the end of the algorithm if $P \geqslant N$ (line 7).

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 [8] as follows:

$$A_r = A.2^n \bmod N$$
$$B_r = B.2^n \bmod N$$

It can be done by Montgomery multiplying the inputs by the constant $N_r = 2^{2n} \bmod N$.

The result of the Montgomery multiplication of A_r by B_r becomes:

$$P_r = A.B.2^n \bmod N$$

P_r is converted back to a normal representation by Montgomery multiplying it by one.

Use of the Montgomery multiplier for modular exponentiation

When using the Montgomery multiplier for modular exponentiation (see algorithm 1) P_0 and Z_0 have to be converted to N-residue before running the loop and Z_n has to be converted back at the end of the algorithm. This only leads to three extra modular multiplications compared to the $2n$ performed in the main loop. Hence the cost of these conversions can be neglected for large bitwidths.

No final subtraction

Even if only one final comparison and subtraction are needed in the Montgomery algorithm, the extra hardware required is quite area-consuming.

To solve this problem, [7] uses a (n+3)-bit Montgomery multiplier to compute the modular multiplications of the exponentiation. With such an design, it is shown that the result of the exponentiation is strictly less than the modulus N with a probability $1 - 2^{-(n+2)}$. With a common bitwidth n of 1024:

$$2^{-(n+2)} \approx 10^{-309}$$

Hence this probability can be considered equal to 1.

[2]When using this algorithm for encryption/decryption N is odd as the product of two odd numbers.

Hardware implementations

Many hardware implementations of the Montgomery algorithm have been achieved for FPGA. We describe here some major contributions.

[8] presents the ARSA core, a scalable RSA architecture taking advantage of the high-speed adder/subtractor logic of Altera FPGAs in arithmetic mode. The Montgomery multiplier of the ARSA core uses two simple adders along with two multiplexers for the input selection. The quicker version of the ARSA core runs at 200 MHz and takes an area of 900 Altera Logic Elements (LE). It is capable of computing a 1024-bit modular exponentiation in about 80 ms. This design is very small but slow. Our exponentiator design is close to the one presented in this paper. However, in order to make it much faster, we do not perform an n-bit multiplication using a $\frac{n}{k}$ bit block requiring several passes through the same block, as is done in [8].

In [6] and [1], a comparison between a Montgomery multiplier implementation with two carry-save adders (CSA) and one with a single CSA is done. A carry-save adder takes three numbers x, y and z and add them together to obtain two numbers: the sum s and the carry c. A n-bit CSA consists of n full-adders and performs the addition $x+y+z = c+s$ in $O(1)$ time. For comparison, a standard ripple adder has a latency in $O(n)$ due to the need for carry propagation. The redundant representation (s, c) used by the CSA leads to no need for carry propagation making the CSA faster than a conventional adder. The one-CSA implementation turns out to take half the area of the two-CSA implementation with better speed performance. [1] also compares these two implementations with an interleaved modular multiplication design. The one-CSA implementation takes the same area than the interleaved modular multiplication version but is a bit slower for bitwidths greater than 512 bits. We need to be careful with these results which were obtained in 2002 on a Virtex II FPGA which is now out of date. An implementation of the one-CSA multiplier on a current FPGA should be fast and not too area-consuming. The main weakness of this design is that it is not parametric. Our design adds pipeline and replication capabilities to this implementation to explore as much design space as possible.

Another interesting design is presented by Blum in [7]. The full multiplier uses a systolic array of processing elements, each of them being a Montgomery multiplier cell with no final subtraction. This implementation is much more area consuming than a one-block implementation but is also faster. According to [8], this implementation takes more than ten times the area of an ARSA core for a bitwidth of 1024, and is twice faster. With our design, one can choose between a version with or without final subtraction. We do not use a systolic array but a pipeline instead.

In [13], a parametric Montgomery multiplier design is presented. The parameters are the number of processing elements, the radix and the number of words used. Our multiplier explores new possibilities by applying variable pipelining and variable replication to the Montgomery multiplication algorithm.

Finally, [11] presents a scalable pipelined Montgomery multiplier whose number of processing elements can be chosen according to the area availability and the speed requirements. Every pipeline element of this design undergoes two stall operations. Moreover, the way the pipeline is organised leads to the need of a quite complex RAM decoder. Our pipeline organisation is simpler and no extra stall cycle is introduced by adding a new pipeline block.

2.3.4 Performance comparison

A comparison of the previous modular multiplication algorithms in term of the product hardware area by time (AT) is done in [6]. The simple method for modular multiplication has an AT complexity of $O(n^2 \log n^2)$ where n is the bitwidth of the inputs A and B and of the modulus N. The interleaved modular multiplication method is $\log n$ better with an AT in $O(n^2 \log n)$. The Montgomery algorithm has the best asymptotic AT complexity in $O(n^2)$. These results make the Montgomery multiplier used in almost all designs relying on modular multiplication.

Power consumption is also an important feature of embedded designs [14] and has to be taken into account together with speed and area when talking about performance.

2.4 Primality tests

The first step of RSA key generation presented in section 2.1.1 relies on the ability of the encryption system to generate large prime numbers. Testing random numbers for primality is a common method.

2.4.1 Naive methods

A simple primality test for an integer n consists in checking whether any integer m from 2 to $\sqrt{n}$ divides n. The efficiency of this test can be improved with several methods like skipping all the even numbers except 2. As 2 divides all the even numbers, if an even number divides n then 2 divides n indeed. However, for large primes this method is impractical. For instance the version skipping the odd numbers requires about $\frac{\sqrt{n}}{2}$ divisions and is therefore very slow.

2.4.2 Probabilistic methods

Probabilistic methods determine whether or not a number is prime with a certain probability of error. More precisely, all numbers declared composite[3] by these tests are not prime whereas a number declared prime can be composite with a small probability.

The Rabin-Miller strong pseudoprime test

A common probabilistic test is the Miller-Rabin strong pseudoprime test given in algorithm 4. This test relies on the fact that if we can find an integer a such that:

$$a^d \neq 1 \bmod p$$
and
$$a^{2^j d} \neq -1 \bmod p \text{ for all } 0 \leqslant j \leqslant r-1$$

then an odd integer p, written as $p = 2^r d + 1$, is composite.

Monier and Rabin have shown that if k tests are performed on a composite number, then the probability that this number passes each test is less or equal to:

$$P = \frac{1}{4^k} \tag{2.6}$$

Input: $p = 2^r d + 1$ odd integer, k parameter determining the accuracy of the test
Output: *composite* if p is composite, *prime* if p is probably prime

```
for i = 0 to k − 1 do
    Choose a random integer a with 1 ⩽ a ⩽ p − 1
    if a^d = 1 mod p or a^(2^j d) = −1 mod p for some 0 ⩽ j ⩽ r − 1 then
        continue
    else
        return composite
end
return prime
```

Algorithm 4: Rabin-Miller strong pseudoprime test (from [2])

Instead of using random numbers, the first k prime numbers can be used. This is shown in algorithm 5.

To reduce the probability of errors, prime testers often combine several probability tests. For instane, the `PrimeQ` Mathematica function[2] uses a Lucas test combined with the Rabin-Miller

[3] A composite number is a number that can be written as a product of more than one prime factor, that is a non-prime number.

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 ⩽ j ⩽ r − 1 then
4         continue
5     else
6         return composite
7 end
8 return prime
```

Algorithm 5: Rabin-Miller strong pseudoprime test deterministic variant

test. More information on the Lucas Test can be found in [15].

A hardware implementation of the deterministic variant of the Rabin-Miller test is done in [11] using a pipelined Montgomery multiplier. It can test a 1024-bit number in less than 2s and takes about 9000 slices of a Virtex II FPGA. It uses the pipelined multiplier presented in the same paper.

2.5 Summary

This chapter has presented the background material and the related work of this report. The important steps of the RSA key generation and encryption processes have been put forward. The most relevant algorithms used for modular multiplication, modular exponentiation and primality testing have been described together with their existing hardware designs and implementations. Most designs in this thesis are inspired by these hardware implementations, retaining the best and more interesting parts of each of them.

In the next chapter, we present our new parametric hardware implementation of the modular multiplication, modular exponentiation and primality test.

Chapter 3

Hardware design and implementation

This chapter presents our hardware designs of the Montgomery multiplication, the modular exponentiator and the Rabin-Miller prime tester. Section 3.1 describes our design goals and objectives. Section 3.2 summarizes our overall design choices. In section 3.2, our Montgomery multiplier design is described in depth. In sections 3.4 and 3.5, our multiplier is integrated into two modules used by most public key cryptosystems: a modular exponentiator and a new design of the Rabin-Miller prime tester. Section 3.6 puts forward the tools used for implementation and testing of our design. Finally, section 3.7 sums up the chapter.

3.1 Requirements

Our goal is to create hardware modules for the major steps of RSA key generation and encryption/decryption. Our main targets are FPGAs.

3.1.1 Key generation

For the key generation part of the RSA, we focus on the first step: the generation of large prime numbers (see section 2.1.1). This part is in fact the most time-consuming of all four. Step 2 and 3 are simple multiplications and step 4 can be performed using the extended euclidean algorithm[1]. The main input of our prime tester is a number to test. The module has to support every power of two bitwidths up to at least 2048 bits leading to a maximum bitwidth of 4096 bits for the key. That should be enough to keep data confidential for the next two decades as presented in section 2.1.3. The output of the module is a flag indicating whether the number is prime or not. Moreover the module has to be highly parametric in order to adapt to many speed and area requirements and many FPGA families.

3.1.2 Encryption/Decryption

Our design also has to support RSA encryption/decryption. The inputs are the plaintext P (respectively ciphertext C), the public key E (respectively private key D) and the modulus N. This module has to support inputs of up to at least 4096 bits. Its output is the ciphertext C (respectively plaintext P). We also want this module to cover as much design space as possible.

3.1.3 Summary of the design requirements

To sum up, our final implementation should:

1. be able to generate prime numbers given a random number source
2. be able to perform RSA encryption/decryption

[1] A shift version of the extended euclidean algorithm is presented in [16] for instance

3. be highly parametric covering a large design space from a slow but area-efficient to a very fast but area-consuming solution
4. be able to be tuned to the area/speed requirements of many projects
5. have easily modifiable parameters that can be set without having to make major changes in the code
6. be protected against major attacks

3.2 Overall design choices

3.2.1 Modular Multiplier

Section 2.1 shows that the core of most algorithms used in the RSA key generation and encryption/decryption parts is the modular multiplication. Therefore the choice of this module clearly impacts the overall performance of our system. The pre-analysis of section 2.3.4 clearly demonstrates that the primary choice for modular multiplication is the Montgomery multiplier. It is the basic block of our modular multiplication design.

To make the multiplier faster and more parametric than existing designs, our basic cell is pipelined and replicated. The number of pipeline stages and replications are the two most important parameters of our Montgomery multiplier. Both methods are explained in depth throughout the chapter.

3.2.2 Prime tester

The prime tester is a hardware implementation of the deterministic variant of the Rabin-Miller primality test. This algorithm is relatively simple as it only uses modular multiplication and exponentiation. This creates a great potential for design reuse. As a matter of fact, our design already uses modular multiplication and exponentiation for encryption. Hence only a few extra area for logic should be needed to implement this primality test. What is more, this method has a low probability of error for a few number of tests as shown in section 2.4.

The use of our multiplier makes our prime tester capable of challenging existing implementations while remaining simple. Its main parameters are:

- the number of pipeline stages of the multiplier
- the number of replications of the processing element of the multiplier
- the number of primes used to test the input

3.3 Montgomery multiplier design

This section presents our Montgomery multiplier design. Our multiplier supports any bitwidth and any number or pipeline stages. We design a simple and flexible pipeline control. Inside a pipeline block, the main processing element can be replicated any number of times to increase the speed of the multiplier.

3.3.1 Basic block

Algorithm

The basic block of our design is a one-CSA based Montgomery multiplier. As shown in section 2.3.3, the carry save adder is faster than a ripple carry adder with no area overhead. Algorithm 6 from [6] is used.

Lines 4 to 7 of this algorithm are the equivalent of lines 3 and 4 of algorithm 3. The four following cases are considered:

- If $a_i = 0$ and S and C are both odd or both even, then the intermediate result is even and S and C are left unchanged.

- If $a_i = 0$ and only one of S and C is even, the other being odd, N is added to make the intermediate result even.

- If $a_i = 1$, B has to be added to S and C. This intermediate result will be even if exactly one of S, C and N is even or the three of them are. In that case, N does not need to be added to this intermediate result.

- If $a_i = 1$ and if exactly one of S, C and N is odd or the three of them are, then the intermediate result if odd and N has to be added.

Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, $N = \sum_{i=0}^{n-1} n_i 2^i$, $a_i,\ b_i,\ n_i \in \{0,1\}$
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** $P \geqslant N$ **then** $P = P - N$

Algorithm 6: Fast Montgomery Algorithm for modular multiplication

Only one final basic addition is needed in line 12 to convert P back to a normal representation. If the Montgomery multiplier is used alone, the final subtraction of line 13 also has to be performed.

Design

The diagram of a multiplier cell is given in figure 3.1. A counter gives the value of the current iteration. Lines 4 to 7 of algorithm 6 are implemented as a 4-to-1 n-bit multiplexer with special control lines selecting the value of I corresponding to the current iteration. The input $B + N$ is precomputed with an n-bit adder. Two registers are used to store the intermediate values of C and S. A multiplexer is used to select the value of a_i in each iteration. Line 8 is implemented with the carry-save adder. Another adder is used to perform the final addition of line 12. The comparator compares the result of the addition with the modulus N. If this result is greater than N, N is selected as the second input of the subtractor, otherwise this input is set to zero. The *start* and *reset* signals allow the control of the multiplier. *done* signals the availability of the result. For a complete description of how to use the multiplier cell and the corresponding timing diagrams, see the specification of this module in appendix A.

Finite State Machine

The finite state machine (FSM) of the Montgomery multiplier basic cell is given in figure 3.2.

The FSM consists of four states, the current state being updated at each positive edge of the clock:

- IDLE: the multiplier is waiting for work.

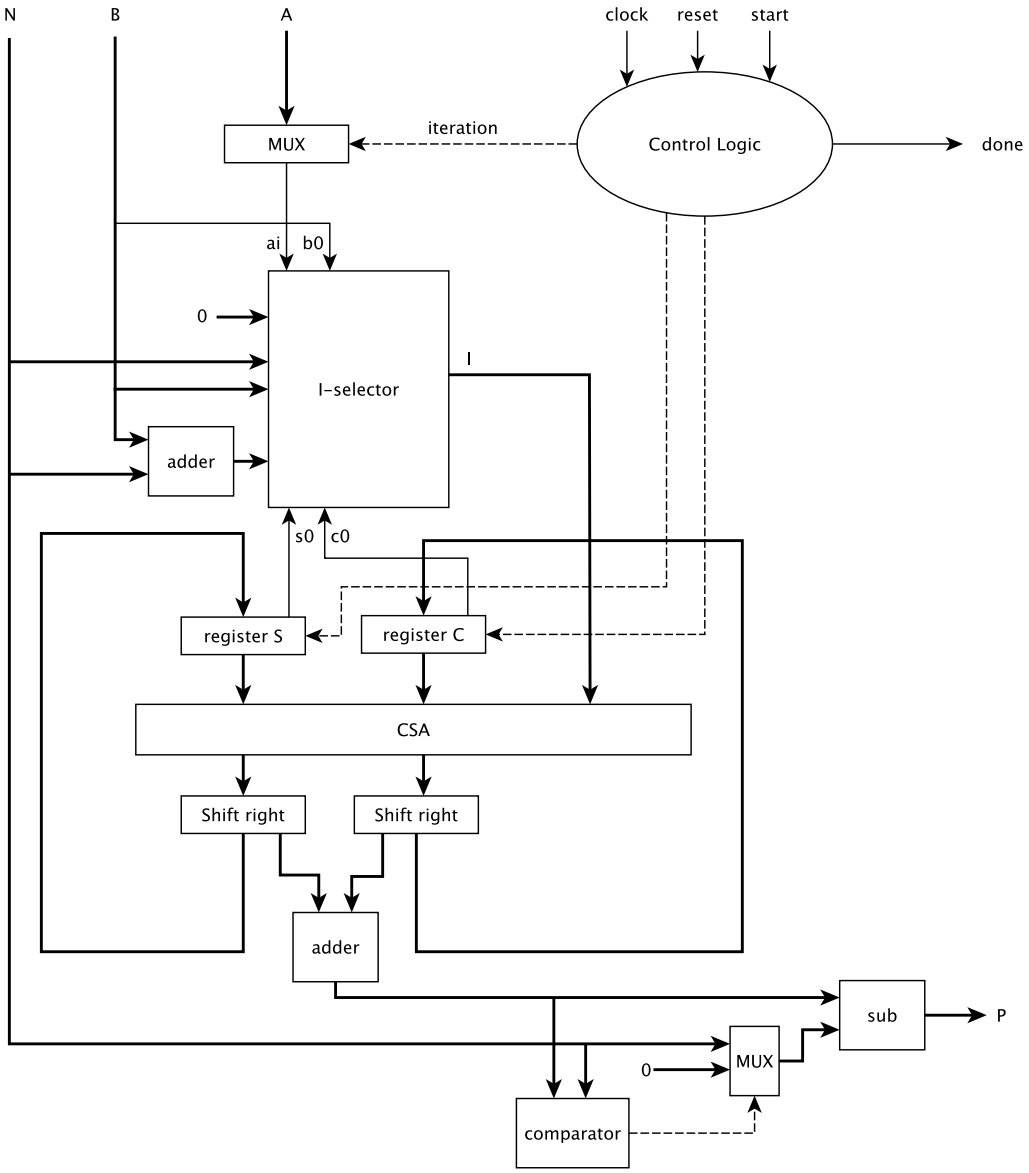


Figure 3.1: Diagram of a Montgomery Multiplier cell

- LOADING: the registers containing the values of S and C are initialised and the iteration counter is reset.

- RUNNING: the modular multiplication is being performed. At each iteration the new values of S and C are computed, shifted right and stored in the corresponding registers.

- FINISHED: the modular multiplication is finished and the `done` signal asserted.

The inner `mult_done` signal indicates that all the iterations have been performed. Note that if `reset` is deasserted, the LOADING state always lasts one clock cycle (no transition condition between LOADING and RUNNING). Also note that in the FINISHED state, the `start` signal has to be deasserted to get back to the IDLE state, enabling a new modular multiplication to be performed.

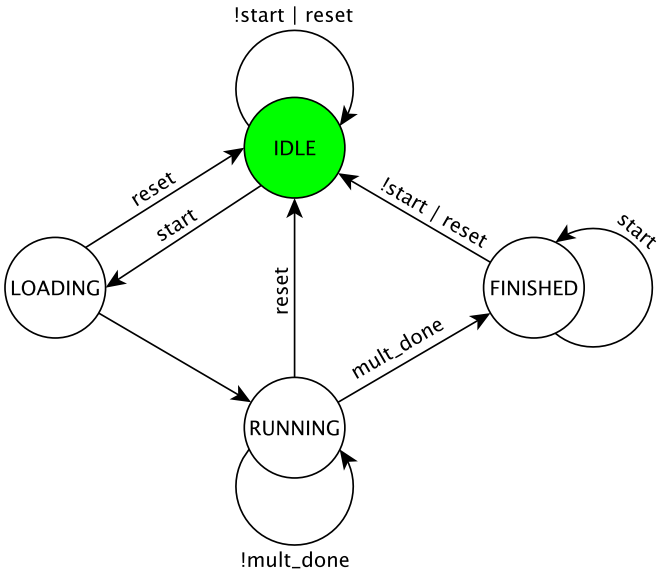


Figure 3.2: Finite state machine of the Montgomery Multiplier cell

3.3.2 Pipelining the design

Idea

After I obtained a working Montgomery multiplier block, I realised that the algorithm used could be easily pipelined. Pipelining increases the throughput[2] of a design by dividing a block into sub-blocks, each of them performing a part of the operations done by the initial block. Registers are inserted between sub-blocks to save their outputs for treatment by the following block as soon as the latter is ready. Pipelining increases the area taken by a design as extra registers are needed. The sum of the area taken by each sub-block is also often bigger than the area taken by the block alone due to routing area overhead.

Looking closely at algorithm 6, we see that each iteration depends on the result of the previous one. Hence this algorithm cannot be easily parallelizable. However we can divide the number of iterations between different pipeline blocks. For instance for a bitwidth of 32 bits, we can choose to implement 4 pipeline stages. In that case, the first sub-block could perform the first 8 iterations (0 to 7) using the first 8 bits of A, the second sub-block iterations 8 to 15 using bits 8 to 15 of A, etc. After having finished its iterations, sub-block 1 passes its final values of C and S to sub-block 2 which begins its computation. Then sub-block 2 passes its values to sub-block 3, and so on. After the last block has completed its iterations, the final addition and subtraction can be performed and the `done` signal set. The only changes that we have to add to the basic cell are:

- the ability to load the S and C registers from inputs

- a parametric number of iterations

We also need to remove the adder and the subtractor from each cell. This hardware is now placed after the last pipeline stage. The basic structure of the pipeline omitting the control and pipeline logic is presented in figure 3.3.

The design presented in [11] has each of its processing elements performing only one iteration at a time. This implies a complex RAM control to give the correct input to each processing element at the right time. As said before, such a design introduces pipeline stalls. In our design, a pipeline block performs a fixed part of the iterations. This leads to a simpler memory control and no need for stalling.

[2]Number of results by unit of time

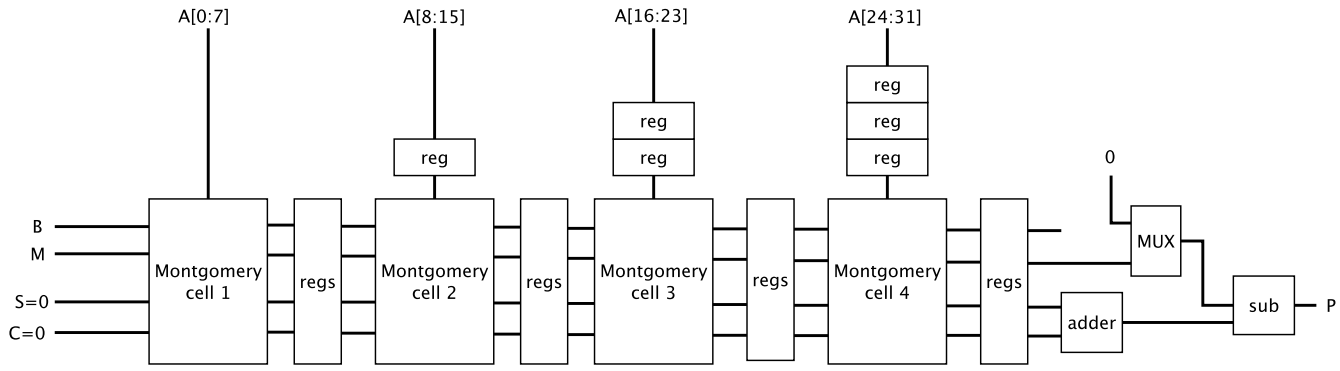


Figure 3.3: Basic structure of a 32 bit pipelined Montgomery multiplier with 4 pipeline stages

To make the module parametric without having to change the code, the Verilog construct `generate` is used. It enables the programmer to instantiate and connect submodules according to the values of the parameters.

Pipeline design and control

In order to explore as much design space as possible, I design a parametric pipeline that can run with any bitwidth and any number of pipeline blocks less than the bitwidth. The pipeline can also run at any fill rate.

Let us consider a multiplier with n-bit inputs. Let us call p the number of desired pipeline stages. My first idea was to use p blocks each performing $\lfloor \frac{n}{p} \rfloor$ iteration and add an extra block performing the last $n \bmod p$ iterations if $n \bmod p \neq 0$. However this idea turned out to be under-efficient. For $n \bmod p \neq 0$, an extra block is added leading to a non-negligible area overhead.

A less area-consuming solution is to add an iteration to the first $n \bmod p$ pipeline block. This leads to $n \bmod p$ blocks computing $\lfloor \frac{n}{p} \rfloor + 1$ iterations, and $(n - n \bmod p)$ blocks computing $\lfloor \frac{n}{p} \rfloor$ iterations. That is the solution I chose.

A pipeline control dealing with blocks of different sizes is challenging to design. This control manages the updates of two types of registers: the registers between blocks and the triangular register array. When every block takes the same number of clock cycles c to process its data and the pipeline always runs full, this control can basically be reduced to a clock running at a period equal to c time the period of the system clock. It is not the case for our problem.

The case of the registers between block is quite simple to deal with. We constraint the `start` signal of the pipelined multiplier to be asserted only during one clock cycle (see appendix A for more information of how to use this module). Then we update these registers at each `done` signal of the immediately preceding Montgomery cell. An example of this mechanism is shown in figure 3.4. It represents the pipeline filling phase of a 8-bit multiplier with 4 pipeline stages. The inputs and outputs of the multiplier (x_i corresponds to A and y_i to B) and the `start` and `done` signals of each cell are depicted. We can see that a cell is started one cycle after the `done` signal of the previous cell has been raised. This delay enables all the `start` signals to be aligned for regular cases. The `start` signal of the first cell corresponds to the `start` signal of the whole multiplier. The `done` signal of the multiplier corresponds to the `done` signal of the last cell. The `done` signal of a given cell asserts the `write enable` signal of the immediately following registers.

The register triangular structure consists of arrays of registers controlled as FIFOs. 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 has to leave when the cell has finished to use it, which is the case when its `done` signal is triggered. In practice, extra registers keep the first empty slot of 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

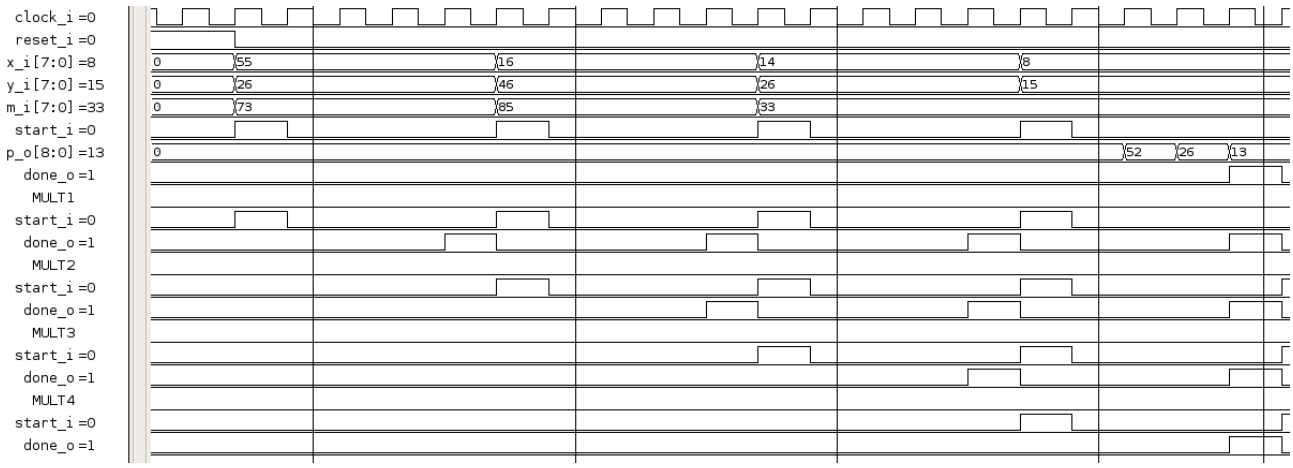


Figure 3.4: Control signals of the registers between blocks for a 8-bit multiplier with 4 pipeline stages

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. A complete example of how this control works is presented in figure 3.5. Figure 3.5a shows the filling phase of a four stage pipeline followed by a draining phase. Figures 3.5b to 3.5m represent the state of the FIFO register array at some important times.

3.3.3 Replicating the processing element

Pipelining improves the throughput of the design but not its latency, that is the time between the first input to enter the multiplier and the first result to be computed. To create a more flexible design, I added the possibility to replicate the CSA as many times as needed.

Replication consists in duplicating a hardware element several times in order to reduce the processing time. At equal frequencies, replication decreases the latency of the design by a factor of r, the total number of replicated processing elements[3]. The area overhead is less than r because only a part of the cell has to be replicated. In practice, replicating the design increases the critical path making the maximum frequency at which the multiplier can run decrease. Experimental results of this effect are presented in section 5.1.1.

We consider a Montgomery cell like the ones of figure 3.3. Figure 3.1 shows that the main processing element of the multiplier is the CSA. The idea is that instead of computing only one sum at each iteration, we can compute $r > 1$ sums, reducing the number of iterations by about of factor of r. This is equivalent to unrolling the loop of algorithm 6 by r. The corresponding algorithm is shown in algorithm 7.

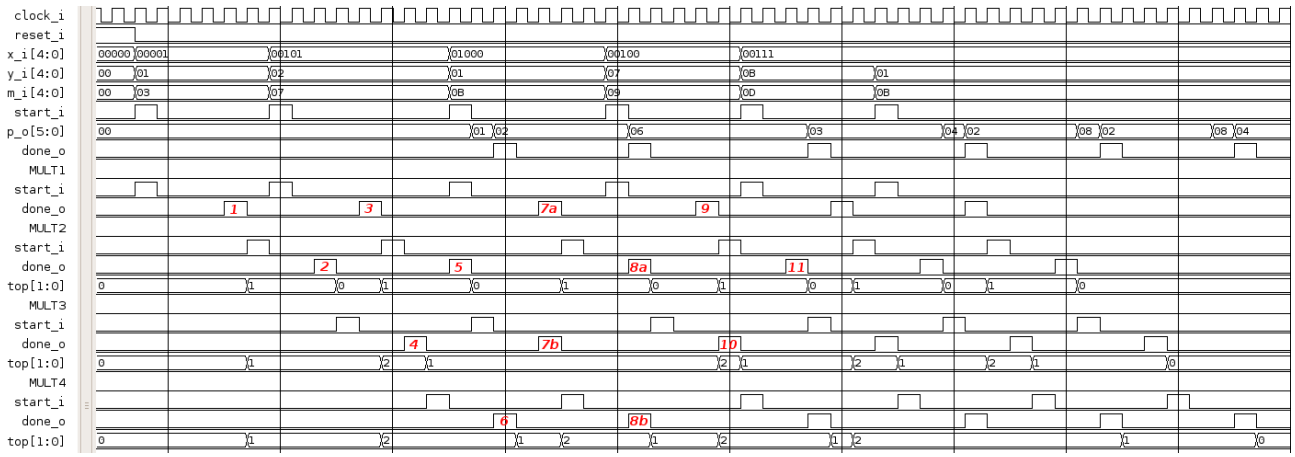
However, the data dependencies in the Montgomery algorithm prevent us from simply duplicating the CSAs and performing the iterations in parallel. Instead, several CSAs along with the shift logic are put in series. The I-selector is also replicated as the value of I is different for each CSA and at each iteration. An example schematic for $r = 2$ focusing on the CSAs and the I-selectors of a cell is presented in figure 3.6

3.3.4 Pipeline and replication

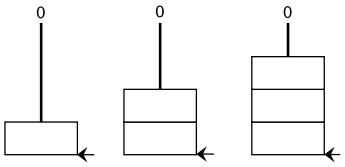
For a bitwidth of n bits, any level of pipelining p and replication r can be chosen. Each of the p pipeline blocks has r CSAs.

Pipelining the multiplier reduces the bitwidth of the input A processed by each pipeline block by a factor of p. Replicating each pipeline block reduces the number of iterations by about r. Hence

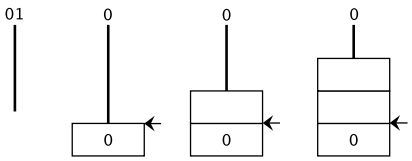
[3] $r = 1$ when the processing element has not been replicated



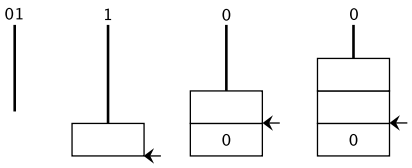
(a) Waveform of the example (the inputs are 5 bit width and the pipeline has 4 stages)



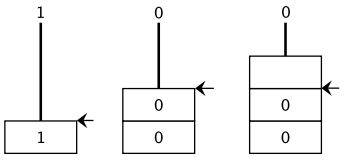
(b) The pipeline has been reset: the FIFO is empty.



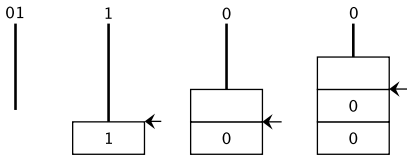
(c) The `done` signal of multiplier 1 is triggered (see number 1 in figure (a)): the inputs enter the FIFOs.



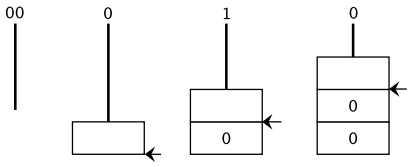
(d) The `done` signal of multiplier 2 is triggered (see number 2): the first element of its FIFO leaves the queue.



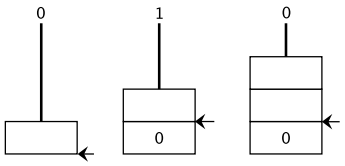
(e) The `done` signal of multiplier 1 is triggered (see number 3): the inputs enter the FIFOs.



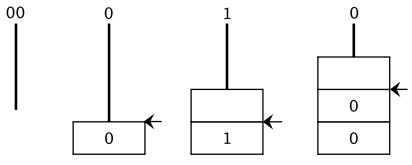
(f) The `done` signal of multiplier 3 is triggered (see number 4): the first element of its FIFO leaves the queue.



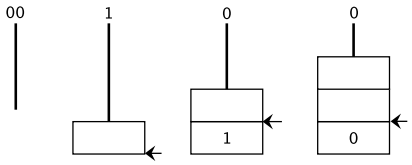
(g) The `done` signal of multiplier 2 is triggered (see number 5): the first element of its FIFO leaves the queue.



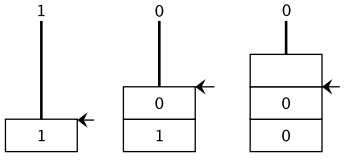
(h) The `done` signal of multiplier 4 is triggered (see number 6): the first element of its FIFO leaves the queue.



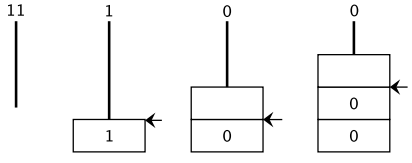
(i) The `done` signal of multiplier 1 is triggered (see number 7a): the inputs enter the FIFOs. At the same time the `done` signal of multiplier 3 is triggered (see number 7b): the first element of its FIFO leaves the queue.



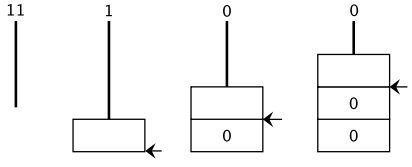
(j) The `done` signals of multiplier 2 (see number 8a) and 4 (see number 8b) are triggered: the first elements of their respective FIFOs leave the queues.



(k) The `done` signal of multiplier 1 is triggered (see number 9): the inputs enter the FIFOs.



(l) The `done` signal of multiplier 3 is triggered (see number 10): the first element of its FIFO leaves the queue.



(m) The `done` signal of multiplier 2 is triggered (see number 11): the first element of its FIFO leaves the queue.

Figure 3.5: Example of evolution of the triangular register structure

Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, $N = \sum_{i=0}^{n-1} n_i 2^i$, $a_i,\ b_i,\ n_i \in \{0,1\}$, $p = \lfloor \frac{n}{r} \rfloor$
Output: $P = A.B.2^{-n} \bmod N$

$S_0 = 0$
$C_0 = 0$
for $i = 0$ **to** $p - 1$ **do**
  for $j = 0$ **to** $r - 1$ **do**
    if $(s_{j,0} = c_{j,0})$ *and* $a_{i+j} = 0$ **then** $I_j = 0$
    if $(s_{j,0} \neq c_{j,0})$ *and* $a_{i+j} = 0$ **then** $I_j = N$
    if $(s_{j,0} \oplus c_{j,0} \oplus b_0) = 0$ *and* $a_{i+j} = 1$ **then** $I_j = B$
    if $(s_{j,0} \oplus c_{j,0} \oplus b_0) = 1$ *and* $a_{i+j} = 1$ **then** $I_j = B + N$
    $S_{j+1}, C_{j+1} = S_j + C_j + I_j$
    $S_{j+1} = S_{j+1}\ div\ 2$
    $C_{j+1} = C_{j+1}\ div\ 2$
  end
  $S_0 = S_r$
  $C_0 = C_r$
end
for $j = 0$ **to** $n\ mod\ r - 1$ **do**
  if $(s_{j,0} = c_{j,0})$ *and* $a_{p+j} = 0$ **then** $I_j = 0$
  if $(s_{j,0} \neq c_{j,0})$ *and* $a_{p+j} = 0$ **then** $I_j = N$
  if $(s_{j,0} \oplus c_{j,0} \oplus b_0) = 0$ *and* $a_{p+j} = 1$ **then** $I_j = B$
  if $(s_{j,0} \oplus c_{j,0} \oplus b_0) = 1$ *and* $a_{p+j} = 1$ **then** $I_j = B + N$
  $S_{j+1}, C_{j+1} = S_j + C_j + I_j$
  $S_{j+1} = S_{j+1}\ div\ 2$
  $C_{j+1} = C_{j+1}\ div\ 2$
end
$P = S_{(n \bmod r)} + C_{(n \bmod r)}$
if $P \geqslant N$ **then** $P = P - N$

Algorithm 7: Fast Montgomery Algorithm for modular multiplication with loop unrolling

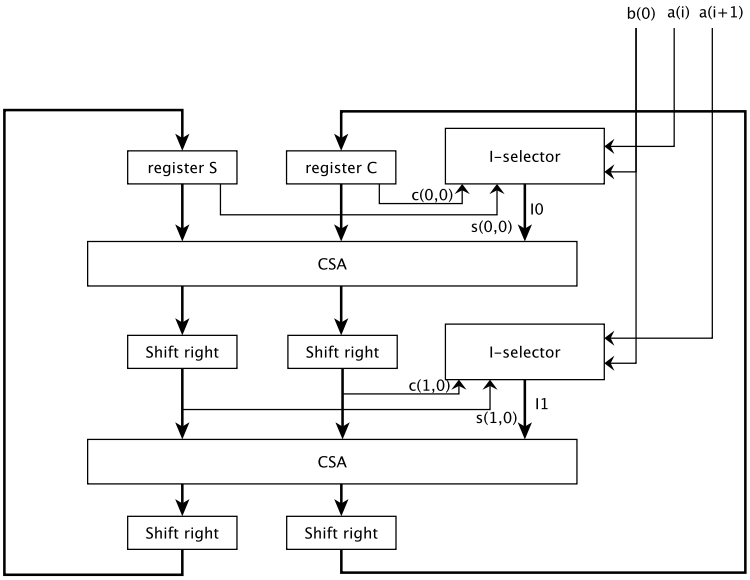


Figure 3.6: Focus on the CSAs and I-selectors of a multiplier cell for $r = 2$

the following constraint has to be respected when choosing r and p:

$$p.r \leqslant n \tag{3.1}$$

3.3.5 Latency and throughput of the multiplier

For a bitwidth of n, p pipeline stages and r replications, the latency of the multiplier in clock cycles is:

$$L_{mult}(n,p,r) = (n \bmod p)\left(\left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{r} \right\rceil + 2\right) + (p - n \bmod p)\left(\left\lceil \frac{\left\lfloor \frac{n}{p} \right\rfloor}{r} \right\rceil + 2\right) + (p-1) \tag{3.2}$$

where $\lceil x \rceil$ represents the ceil of x and $\lfloor x \rfloor$ the floor of x.

After the first result, if the pipeline is kept full, one result is given every:

$$N_{mult}(n,p,r) = \left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{r} \right\rceil + 2 \text{ clock cycles} \tag{3.3}$$

Hence the throughput of the multiplier is $\phi_{mult} = \dfrac{1}{N_{mult}}$ multiplication per clock cycle.

3.4 Exponentiator design

We implement algorithm 1 in hardware using the Montgomery multiplier presented earlier for all modular multiplications. The exponentiator makes full use of the pipeline and replication capabilities of the multiplier. We find the optimal number of pipeline stages for the multiplier making the exponentiator fast without wasting area.

3.4.1 Simple version

First a simple version with a non-pipelined multiplier block is designed. As mentioned earlier, the Montgomery algorithm computes $A.B.2^{-n} \bmod N$. To remove this factor, both operands have to be converted to N-residue, which amounts to Montgomery multiply each of them by a precomputed factor $N_r = 2^{2n} \bmod N$. So, the result of the multiplication is still in N-residue format.

To perform the exponentiation, P_0 and Z_0 of algorithm 1 are first converted to N-residue. The loop is then executed normally. At the end of the algorithm, Z_n is converted back by Montgomery multiplying it by 1. Hence, an exponentiation needs in the worst case $2n + 3$ multiplications.

However, implementing this algorithm directly is not safe. Let us suppose we use our algorithm to decrypt a ciphertext C with the private key (D, N). At each iteration i, if $d_i = 1$ two multiplications are performed. If $d_i = 0$ only one multiplication is performed. An opponent could therefore try to time the decryption operation and get information on D. To protect our design against such a timing attack, the second multiplication in the loop is always done even if it is not necessary. This leads to a total of $2n + 3$ multiplications for each exponentiation.

Design

The design of our exponentiator is represented in figure 3.7. At each iteration, P and Z are stored on a dual-read 1-bit address RAM. The representation of this RAM allows the designer to implement it on an available RAM block of the FPGA if he wants to take advantage of this feature. Otherwise this RAM will be implemented in registers. Usually, using available block RAMs reduces the area taken by the design but also reduces its speed.

Two multiplexers select the inputs of the Montgomery multiplier. The iterations are managed through a simple counter. The control logic controls the multiplexers. At iteration i, the control logic also sets the write and address signals of the RAM according to the value of e_i.

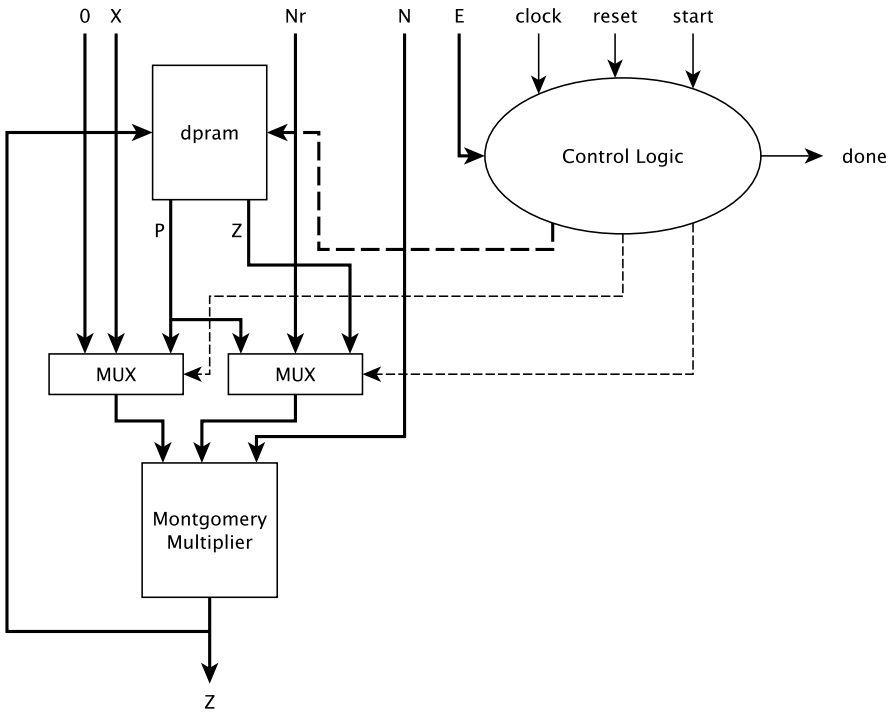


Figure 3.7: Montgomery exponentiator

Finite State Machine

The state machine of the exponentiator is given in figure 3.8. The `mult_done` indicates that the current multiplication is finished. The `exp_done` signal is asserted when all the iterations of the main loop have been performed.

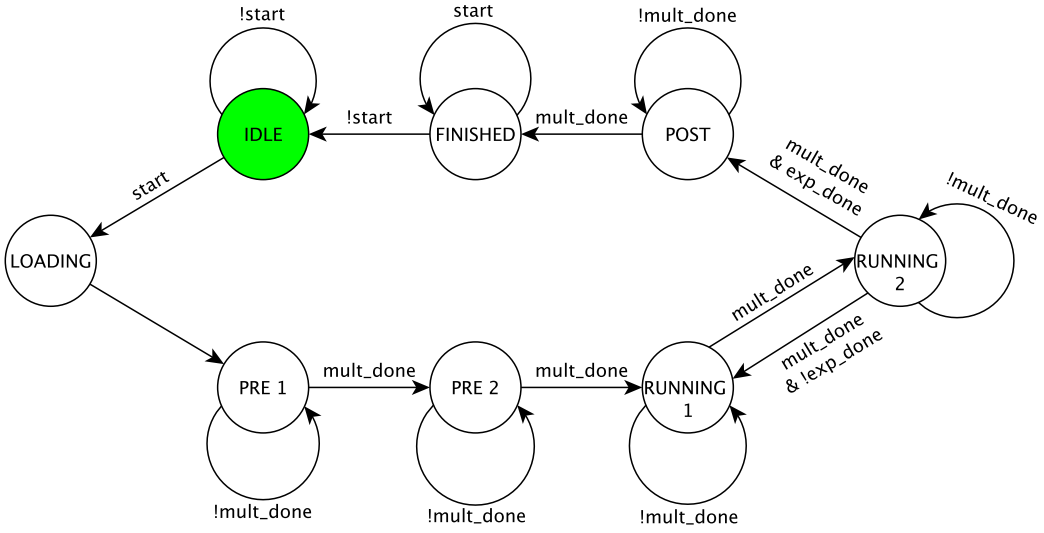


Figure 3.8: State machine of the simple exponentiator
(the `reset` signal is omitted to keep it simple)

The exponentiator has the following states:

- IDLE: the exponentiator is waiting for work.
- LOADING: the iteration counter is reset.

- PRE 1: Z_0 is converted to N-residue. The inputs of the multiplier are 1 and the constant N_r.
- PRE 2: P_0 is converted to N-residue. The inputs of the multiplier are X and N_r.
- RUNNING 1: At each iteration i, the new value of Z is computed. The inputs of the multiplier are Z and P. If $e_i = 1$ the RAM is updated with the new value of Z.
- RUNNING 2: At each iteration, the new value of P is computed. Both inputs of the multipliers are given the previous value of P. The RAM is updated accordingly.
- POST: all the iterations has been performed. Z is converted back to a normal representation. The inputs of the multiplier are Z and N_r.
- FINISHED: the exponentiation is finished. The `done` signal is raised.

Note that the main loop is mapped to the states RUNNING 1 and RUNNING 2 which are executed cyclically. At iteration i, the new value of Z (Z_{i+1}) is computed before the new value of P (P_{i+1}). This prevent us from saving the previous value of P (P_i) for the calculation of the new Z (Z_{i+1}).

3.4.2 Using the Montgomery multiplier with no final subtraction

As mentioned in section 2.3.3, when used with an n-bit exponentiator, the subtractor can be removed if a $n+3$ multiplier is used. The intuition is that performing some extra iterations should reduce the value of the accumulated product and make it less than the modulus.

More precisely, let us remove the last subtraction of algorithm 3 (line 7) and let us consider for instance line 3 of algorithm 1. After the first iteration, $P_1 \leqslant 2N$. Hence the inputs of the multiplier in the second iteration are less or equal to $2N$. That makes $P \leqslant 3N$ in the multiplier for $i < n$. For $i > n$, $P < 2N$ as a_i is equal to 0 in these last cases. Therefore, the useful part of Z in the exponentiator is always less than $2N$ and has a bitwidth less than $n + 1$.

The last conversion of Z back to a normal representation makes it less than the modulus. Recall that this conversion amounts to Montgomery multiply Z by $1 = (0...01)_b$. After the first iteration of this product, we have $P < 4N$. Then from the second iteration $P < 2N$, as $a_i = 0$. If at any iteration k, $p_0 = 0$, N will not be added to P. Hence this will lead to $P < N$ for all $i > k$ and the result will be strictly less than N. The probability that $p_0 = 1$ at each iteration is very low for the bitwidths used in RSA, making this trick safe.

The tradeoffs between using a subtractor with a n-bit multiplier or using a $n + 3$ bit multiplier with no subtractor will be explored in section 5.1.2.

3.4.3 Using the pipelined multiplier

The exponentiator described so far can use a multiplier with any number of replications and pipeline stages with or without final subtraction. However, the exponentiator waits for the last multiplication to be completed before starting a new one. Hence it does not use the multiplier's pipeline to speed up the computation. The final step of the design is to make it take full advantage of the multiplier's pipeline capabilities.

Optimal number of pipeline stages

We first analyse the dependencies in algorithm 1 to figure out how much the multiplier has to be pipelined for use with the exponentiator. We have the following dependencies:

1. P_{i+1} depends on P_i
2. Z_{i+1} depends on P_i and Z_i

These dependencies make it clear that only two pipeline stages can be used if we want to keep the multiplier's pipeline full. Moreover, P_i has to be calculated before Z_i as it is used by both Z_{i+1} and P_{i+1}. Hence, the pipeline is first filled with the conversion of P_0 to N-residue, then with the conversion of Z_0. At this point the pipeline is full and the iterations are performed, keeping it full. When the last operation, computing Z_n, enters the pipeline, this one begins to drain. The filling and draining phases of the pipeline are represented in figure 3.9.

The last multiplication is the conversion of Z_n back to normal representation. The pipeline is only used at half its maximum throughput for this last operation. We could possibly still reduce the time taken by an exponentiation by about $100/2n$ percent by allowing a new exponentiation to begin while this conversion is being performed. However, it would make the exponentiator more complex to gain less than 0.5% speed at a common bitwidth of 1024 bits. That is why it is not done here.

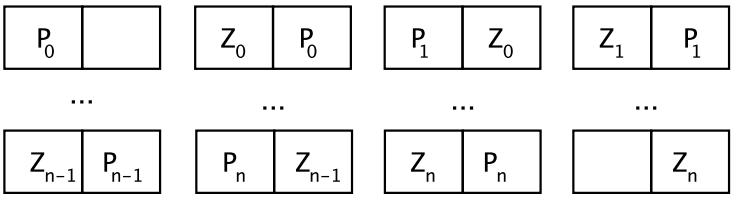


Figure 3.9: Filling and draining phases of the multiplier pipeline when used with the exponentiator

Design

Using the pipeline of the multiplier requires quite a few changes to the design of the exponentiator. First a counter is added to count the number of multiplications performed. It is used to stop filling the pipeline at the end of the main loop.

The FSM is modified. The RUNNING 2 state is removed. The exponentiator is in the RUNNING 1 state during the full loop. This state machine is now only used to reset the counters and set the `done` signal.

For control of the pipelined multiplier, another state machine is introduced. It has the following states and effects on the signals:

- EMPTY: the pipeline is empty
- P: P is alone in the pipeline (first diagram of figure 3.9)
- Z-P: P is in stage p of the pipeline and Z in stage l with $p > l$ (second diagram). When the multiplication is finished, P is written in the RAM.
- P-Z: P is in stage p of the pipeline and Z in stage l with $p < l$ (third diagram). When the multiplication is finished, if the Z in the pipeline corresponds to Z_0 of algorithm 1 (initialisation), Z is written in the RAM unconditionally. Otherwise it is written if $e_i = 1$.
- Z: Z is alone is the pipeline (last diagram). This only occurs for the computation of the last Z (corresponding to Z_n of algorithm 1). It is written in the RAM if $e_n = 1$.

With this new control, any bitwidth and any pipeline length can be used. Being able to use any pipeline length is useful to confirm experimentally that the best pipeline length is 2 (see section 5.2.1).

3.4.4 Critical path

By performing early syntheses of the design, I realised that the exponentiator was running at a very low maximum frequency (around 30 MHz for a bitwidth of 512 bits) compared to the multiplier

(around 200 MHz for the same bitwidth). An analysis of the results given by the synthesis tool puts forward two interesting things.

First the path from the C and S registers of the multiplier to the RAM of the exponentiator (saving the intermediate values of P and Z) passes through the final adder and subtractor of the multiplier. The latency of these adders/subtractors are clearly a bottleneck when the bitwidth becomes large as they have to be crossed by the signals in one clock cycle.

Second, the path from the exponentiator RAM to the C/S registers goes through the adder computing the value of I in the multiplier. For the same reasons the latency of this adder is also a bottleneck.

To cope with these two problems, these adders/subtractors are replaced by pipeline ones. The depth of the pipeline is a parameter. For instance, for a bitwidth of 512, if 4 pipeline stages are chosen for the adders, we only need to cross a $512/4 = 128$ bit adder at each clock cycle. However, this method leads to some extra cycles to compute an exponentiation.

The advantages and drawbacks of using the pipeline adders/subtractor are presented in section 5.2.1.

3.4.5 Time to perform an exponentiation

For a bitwidth of n, p numbers of pipeline stages (with $p \leqslant 2$) and r replications, the time to perform an exponentiation in clock cycles (neglecting the extra cycles introduced by the use of pipeline adders) is:

$$L_{exp}(n, p, r) = 2.L_{mult}(n, p, r) + (2n + 1).N_{mult}(n, p, r) + 2.(n + 2) + 1 \tag{3.4}$$

where $L_{mult}(n, p, r)$ and $N_{mult}(n, p, r)$ are defined in formulae 3.2 and 3.3 respectively.

3.5 Prime tester design

The prime tester designed gathers both the multiplier and exponentiator presented before. The pipeline and replication capabilities of the multiplier are optimally used thanks to an in-depth analysis of the Rabin-Miller primality test.

3.5.1 Algorithm

I implemented the deterministic variant of the Rabin-Miller primality test presented in algorithm 5. A more detailed version of this algorithm, closer to the implementation, is given in algorithm 8.

The first loop of line 3 finds d and s such that $p = 2^s d + 1$. The primality tests are performed in the outer most `for` loop (line 7). Lines 12 and 14 show that we need a modular exponentiator to perform these tests. The test of line 17 only requires modular multiplication. As a matter of fact, in line 12 we get the value of $a^d \bmod p$. Moreover the following relation holds:

$$a^{2^{j+1}d} \bmod p = (a^{2^j d} \bmod p)^2 \bmod p \tag{3.5}$$

Hence at each iteration j of the inner loop, we just need to Montgomery multiply $a^{2^j d} \bmod p$ by itself. As the exponentiator is not used during this loop, we only use one multiplier to save area. This multiplier is either used by the exponentiator or in standalone mode.

3.5.2 Design

Components

A diagram containing the basic blocks and connections of the prime tester is presented in figure 3.10. To keep it clear the control wires of the multiplexers and the logic computing s and d (basically a shifter and a comparator) are omitted.

The exponentiator is only used to compute $a^d \bmod p$. The multiplier is used both by the exponentiatior and to compute the values of $a^{2^j d} \bmod p$. These values are stored in the register `int`

Input: Prime number p to test, set P of the $|P|$ first primes
Output: *composite* if p is composite, *prime* if p is probably prime

```
1  d = p − 1
2  s = 0
3  while d_0 = 0 do
4      d = d >> 1
5      s = s + 1
6  end
7  for i = 0 to |P| − 1 do
8      a = P[i]
9      next = 0
10     if p = a then
11         return prime
12     if a^d mod p = 1 then
13         continue
14     if a^d mod p = p − 1 then
15         continue
16     for j = 1 to s − 1 do
17         if a^(2^j d) mod p = p − 1 then
18             next = 1
19             break
20     end
21     if next = 0 then
22         return composite
23 end
24 return prime
```

Algorithm 8: Detailed Rabin-Miller deterministic algorithm

`value`. The first value stored ($j = 0$) comes from the exponentiator and the others ($j > 0$) from the multiplier. The selection of the value to store in the `int value` register is done by a multiplexer.

The multiplier is also used for various conversions:

- The conversion of $p - 1$ to p-residue for comparison with $a^{2^j d} \bmod p$ in the inner most loop. This value is computed once at the beginning and stored in a register.
- The conversion of $a^d \bmod p$ to p-residue before it is used to compute $a^{2d} \bmod p$ in the inner most loop

The operands of the modular multiplication are chosen by two multiplexers. Two multiplexers are also used for selection of the operands of the comparator. The comparator is used for four different comparisons (lines 10,12, 14 and 17 of algorithm 8).

The values of the first prime numbers are stored in a ROM whose address is selected by the control logic. The control logic manages the iterations of each loop through counters. It contains the FSM of the prime tester.

The `reset` signal resets all the registers and the FSM of the module. The `start` signal starts the primality test and the `done` signal is raised when the test is finished. The `result` signal is set to 0 is the number is composite and 1 if it is probably prime.

Finite State Machine

The FSM of the prime tester is given in figure 3.11. The `reset` signal, which sets the module back to IDLE state, is omitted.

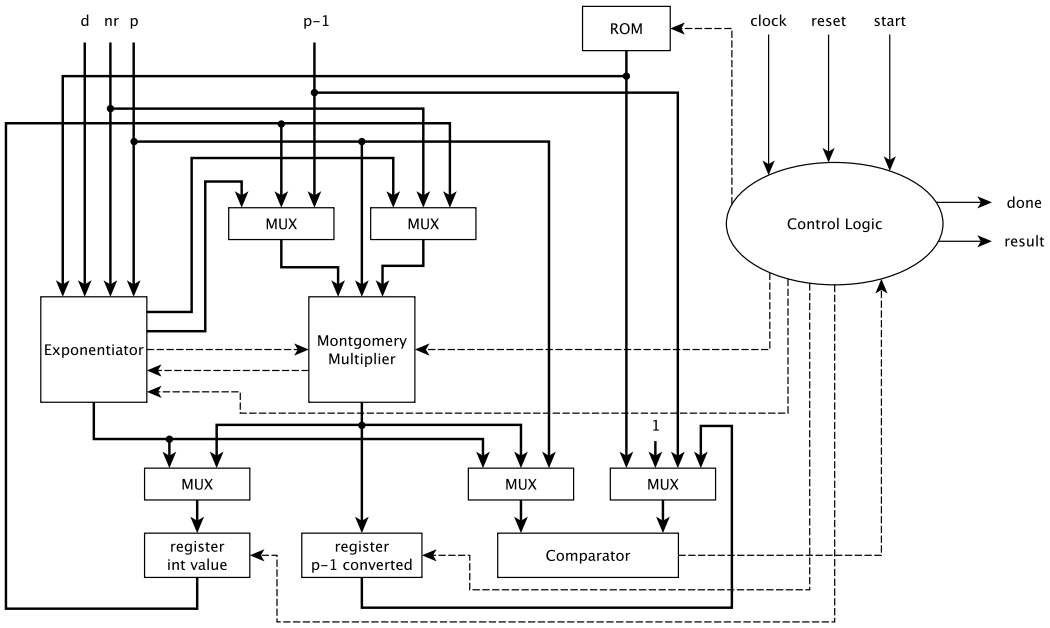


Figure 3.10: Schematics of the prime tester

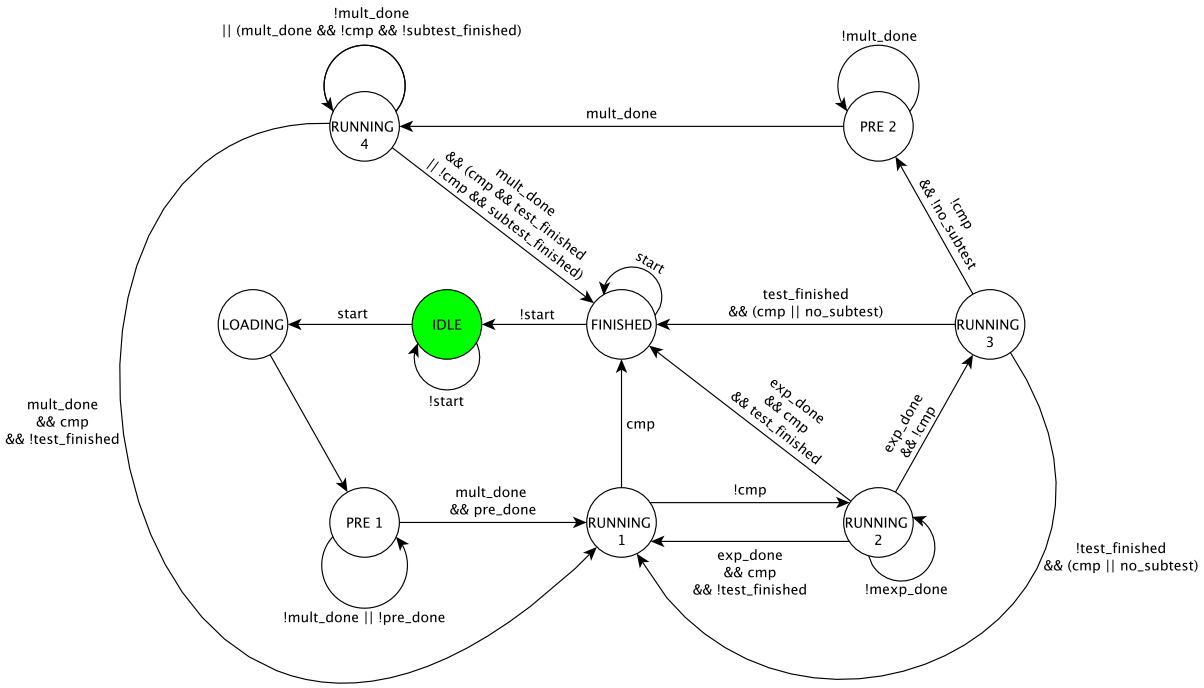


Figure 3.11: Finite state machine of the prime tester

The states of the prime tester are:

- IDLE: the module is waiting for work.
- LOADING: s is initialized to 0.
- PRE 1: the values of d and s are found and $p-1$ is converted to p-residue.
- RUNNING 1: $p = a$ is tested.
- RUNNING 2: $a^d \bmod p$ is calculated and $a^d \bmod p = 1$ is tested.
- RUNNING 3: $a^d \bmod p = p - 1$ is tested.

- PRE 2: a^d mod p is converted to p-residue.
- RUNNING 4: the inner most for loop is performed.
- FINISHED: the primality test is finished. The `done` signal is raised and `result` is set accordingly.

The transition signals are:

- `mult_done`: the current multiplication has just finished.
- `exp_done`: the current exponentiation has just finished.
- `cmp`: if 1 the inputs of the comparator are equal.
- `test_finished`: the outer most loop is finished.
- `subtest_finished`: the inner most loop is finished.
- `no_subtest`: the inner most loop has no iteration (it is the case when $s < 2$).

When the prime tester is waiting for work in IDLE state, a `start` signal changes the state to LOADING. s is initialized to 0 in one clock and the module goes to the PRE 1 states where the values of s and d are calculated and $p - 1$ is converted to p-residue. When this is done, the state changes to RUNNING 1.

In RUNNING 1, if $p = a$, p is declared prime: the prime tester goes to the FINISHED state and `result` is set to 1. Otherwise its goes to RUNNING 2 where a^d mod p is computed.

In RUNNING 2, once a^d mod p has been computed:

- If a^d mod $p = 1$ and there is no more iteration of the outer most loop left, the test is finished and p is declared prime.
- If a^d mod $p \neq 1$, the state changes to RUNNING 3.

In RUNNING 3 :

- If a^d mod $p = p - 1$ and there is no more iteration of the outer most loop left, the test is finished and p is declared prime.
- If a^d mod $p = p - 1$ and there is more iterations to execute, we go back to RUNNING 1 where the next iteration begins.
- If there is no more iteration of the outer most loop left and the inner most loop does not need to be executed, the test is finished and p is declared prime if the last a^d mod p computed was equal to $p - 1$, composite otherwise.
- If a^d mod $p \neq p - 1$ and the inner loop needs to be executed, we go to PRE 2.

In state PRE 2, a^d mod p is converted to p-residue and the state changes to RUNNING 4.

In RUNNING 4, the iterations of the inner most loop are done:

- If $a^{2^j d}$ mod $p = p - 1$ for one j and the outer most loop is not finished, we go back to RUNNING 1 where the next iteration begins.
- If $a^{2^j d}$ mod $p = p - 1$ for one j and the outer most loop is finished, the test is finished and p is declared prime.
- If $a^{2^j d}$ mod $p \neq p - 1$ for all j, the test is finished and p is composite.

Using the pipeline multiplier

The multiplier of the prime tester can be used with any number of replications and pipeline stages. So far the pipeline capabilities of the multiplier is only used by the exponentiator, that is for the calculation of $a^d \bmod p$. We could adapt the prime tester to take advantage of the pipeline capabilities of the multiplier for the calculation of the $a^{2^j d} \bmod p = p - 1$. However we can easily show that for n large enough, most of the time is spent in the calculation of $a^d \bmod p$. As a matter of fact, if the number under test is a random odd number, the mean value of s is:

$$1 + \sum_{k=0}^{n-2} \frac{1}{2^{k+1}} + \frac{n-1}{2^{n-1}} \tag{3.6}$$

For $n > 32$, this sum is very close to 2. Hence the average number of modular multiplications performed in the inner most loop is $s - 1 = 1$, whereas $2n + 3$ modular multiplication are performed by the exponentiator at each iteration of the outer most loop. Using the pipeline capabilities of the multiplier for the inner loop multiplications is therefore irrelevant. On average the pipeline would not be full anyway.

3.6 Tools used and design cycle

I use Verilog for all the RTL implementations. It is the most commonly used language in the design of logic chips at the RTL level. The open source verilog compiler `Icarus Verilog 0.9` [17] is used to compile and simulate the design.

I chose to program each module using a lot of small iterations, most of them consisting of a full cycle from design to synthesis. First, a small non-working version of the module is written in Verilog. This version defines the parameters and the IOs. Then, for complex modules two or three types of tests are written.

The first one is a behavioural simulation of the module under test written in Verilog. This test generates waveforms representing the different signals of the design against the simulation time. This test does not intend to validate the accuracy of the design. It is used for debugging purpose only.

The second type of tests aims at validating each module through extensive testing. The cosimulation features of `MyHDL v0.6` [18] are used. `MyHDL` is a python package allowing description of a design in a HDL language very close to Verilog. Cosimulation allows one to write the benchmark for the design under test in Python and to interface it with the Verilog module, simplifying the testing process. Each important sub-module of the main module is tested. For modules with few possible inputs, every combination is verified. For other modules, the number of tests and the values of the parameters can be chosen and the tests are run with random inputs. Small bitwidths (up to 128 bits) are used for the tests to run in a reasonable amount of time. The use of such a testing method enableds me to verify my hardware design in a way similar to what can be done for a software implementation. An example of the output of a cosimulation test is given in figure 3.12. It is very similar to the output of a standard software test package as `JUnit` for Java.

The last type of tests is used for large bitwidths when a Verilog simulation using `Icarus` turns out to be too computationally expensive. These tests use `Verilator` [19]. `Verilator` is a Verilog cycle-accurate simulator which is up to 100 times faster than `Icarus`. It compiles Verilog synthesizable code into optimized C++ code. I wrote C code interfacing to this simulator to test the multiplier, exponentiator and prime tester modules for bitwidths greater than 128 bits. These tests were written after implementation of the modules.

3.7 Summary

In this chapter we have presented our parametric design of the Montgomery modular multiplier, the module exponentiator and the prime tester. We have described a new way of pipelining the Montgomery multiplier. The exponentiator and the prime tester are designed so that they can take full

```
Tests that the module CSA gives the expected results ...      Width: 32
        Sub-Test 1: OK
ok
Tests that the adder module gives the expected results ...      Width: 32
        Sub-Test 1: OK
ok
Tests that the mexp module gives the expected results ...      Width: 64
        Pipeline stages: 1
            Number of replications:1
                Add/Sub stages:1
Nr 635674421822711691
X: 14496276229477140291
E: 6076922425146223785
M: 2518536961650531611
Result: 867809050818571727
Clock cycles: 9172
        Sub-Test 1: OK
ok
----------------------------------------------------------------------
Ran 3 tests in 7.056s

OK
```

Figure 3.12: An example output of a cosimulation test

advantage of this pipeline. We have also applied replication to the pipeline blocks of the multiplier in an original way introducing a great potential for speeding up the modular multiplication.

These three designs are highly parametric in the sense that the following parameters can be given almost any value:

- the bitwidth of the inputs of each module
- the number of pipeline stages of the multiplier
- the number of replication of the carry-save adder in each pipeline block of the multiplier
- the pipeline depths of the adders/subtractors of the multiplier
- whether or not the final subtraction in the multiplier is performed when used by the exponentiator for RSA encryption/decryption
- the number of primes used in the Rabin-Miller primality test

The next chapter develops a model of our multiplier allowing the designer to quickly tune this module to its design goals by giving insights about the optimal values of the number of pipeline stages and replications. In the last chapter, our three designs are evaluated quantitatively and qualitatively.

Chapter 4

Design space exploration and performance tuning tools

In this chapter we develop a model of our multiplier giving insights about the variations of area and throughput with the pipeline depth and the number of replications. Section 4.1 presents our model. In section 4.2, our model is integrated in an easy-to-use software. Section 4.3 sums up the chapter.

4.1 Modelling a design with pipelining and replication

In this section, we quantify the impacts of pipelining and replication on a design. Our goal is to have information about the best parameters to choose in order to synthesize our Montgomery multiplier on a given FPGA. Using a model reduces the number of syntheses needed to find the optimal parameters of the multiplier. Therefore, it also reduces the integration time of the module. This analysis is inspired by [20].

Let us first do a recap on pipelining and replication.

4.1.1 Pipeline and replication

The configuration chosen for this analysis is depicted in figure 4.1. It is similar to the design of our multiplier.

Pipelining introduces an area overhead due to the need of registers between pipeline blocks, and a triangular register structure for some IOs. Pipelining increases the throughput of the design and the maximum clock speed.

Replication consists in duplicating a hardware element several times in order to reduce the processing time. Replication decreases the latency of the design by a factor of r, the total number of replicated processing elements, with an area overhead of $\rho.r$. This overhead is less than r, that is $0 < \rho \leqslant 1$. That is due to the fact that the control logic and internal registers may not all need to be duplicated.

4.1.2 Latency and Throughput

We consider the general case of a block which needs to perform s iterations to complete a particular operation (a multiplication for example) if no replication is introduced. The latency of such an iteration is $t_{p,e}$. This corresponds to c clock cycles at a frequency f:

$$t_{p,e} = \frac{c}{f} \tag{4.1}$$

The main processing element of this block can be replicated by a factor of r in order to decrease the latency of the block. Hence the total latency of an operation is:

$$t_p = \frac{s}{r}.t_{p,e} \tag{4.2}$$

Latency corresponds to the time taken by the system to compute the first result. Pipelining often increases the latency as extra cycles are needed to manage the pipeline. In our case this phenomenon can be neglected compared to the time spent in real computation.

The throughput of the block depends on both the pipeline length p and the replication factor r [1]:

$$\phi_{enc} = \frac{p.r}{t_{p,e}.s} \tag{4.3}$$

Replicating and pipelining a design reduces the effective number of iterations needed to be computed in each block, leading to the following constraint:

$$r.p \leqslant s \tag{4.4}$$

We assume here that the total size S_p of the inputs whose processing is dispatched between pipelined blocks is equal to s. One iteration uses one bit of this input.

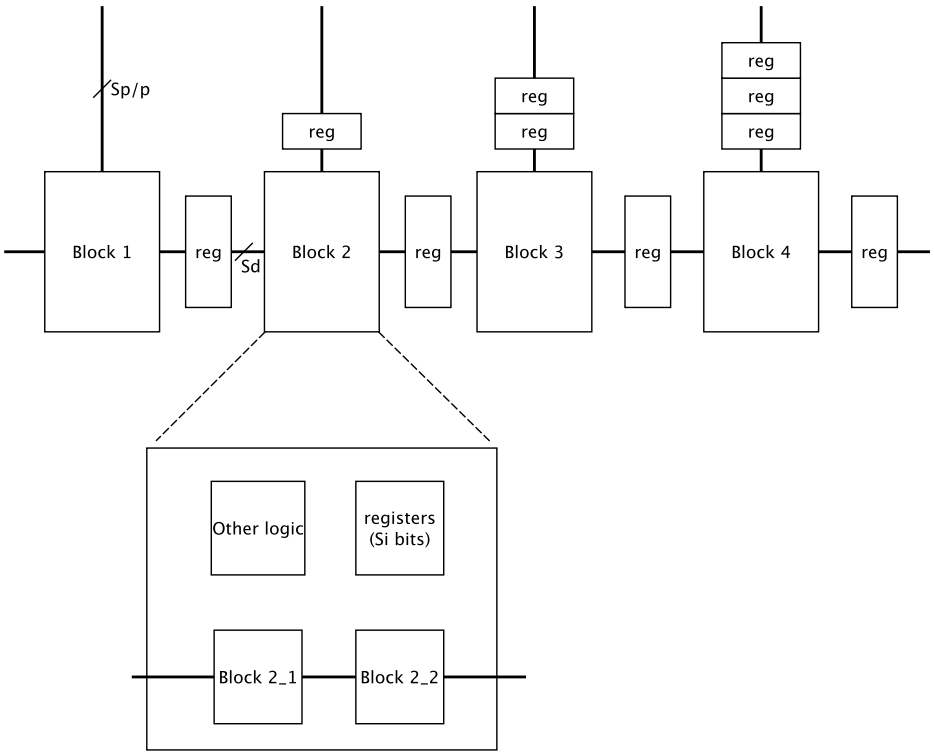


Figure 4.1: Design for $p = 4$ and $r = 2$

4.1.3 Frequency

In our design, we pipeline the iterations performed by the algorithm. For instance if $s = 1024$ and $p = 4$, each pipeline stage will perform 256 iterations, the size of the "non-pipelined" inputs remaining the same. Hence the size of the processing element in each sub-block is the same for all p. Even if the size of the hardware managing the iterations decreases, the critical path is not likely to be reduced a lot. Therefore we assume that f does not depend on p.

[1] This formula does not consider the extra cycles introduced by particular implementation as it is the case for the formulae given in section 3.3.5. It is therefore an approximation of the real throughput of our multiplier.

On the contrary, replication is more likely to have a negative effect on the critical path. The processing element is replicated and extra hardware is needed to manage the iterations. Therefore, let us suppose that the frequency variations can be modelled the following way:

$$f(r) = \frac{f_0}{1 + \lambda(r-1)} \tag{4.5}$$

with f_0 the frequency for $r = 1$ and λ representing the stiffness of the decrease. This represents a parabolic decrease of the frequency with r and particularly fits our multiplier.

4.1.4 Area

We decompose the area taken by our design into two parts:

- A_l the area taken by logic
- A_r the area taken by registers

A_l can be approximated by:

$$A_l = s.p(1 + (r-1)\rho)A_{l,e} \tag{4.6}$$

where $A_{l,e}$ is the area taken by the logic for $s = 1$, $p = 1$ and $r = 1$.

This approximation is relevant when s is proportional to the width of the inputs as it is the case for modular multiplication and exponentiation algorithms. If this condition holds, doubling s doubles the size of the adders, multiplexers, etc, approximately doubling the area.

To calculate A_r we need to derive the total register size (see figure 4.1):

$$\begin{aligned} S &= p.S_d + \frac{1}{p} S_p \sum_{i=0}^{p-1} i + p.S_i \\ &= p.S_d + \frac{1}{2}(p-1)S_p + p.S_i \end{aligned} \tag{4.7}$$

S_i is the total size of the registers internal to a pipeline block. S_d is the size of a register between two pipeline blocks.

Hence:

$$A_r = S.A_{r,e} \tag{4.8}$$

where $A_{r,e}$ is the area taken by a one-bit register.

4.1.5 Constraints

The following constraints are used:

- Maximum available area for logic $A_{l,max}$
- Maximum available area for registers $A_{r,max}$
- Minimum frequency f_{min} at which the design has to run

4.1.6 Optimizations

Throughput

The problem of optimizing the throughput of the design can be formulated this way:

maximize:

$$\phi_{enc} = \frac{p.r}{s.c} \frac{f_0}{1 + \lambda(r-1)} \tag{4.9}$$

such that:

$$A_l \leqslant A_{l,max} \tag{4.10}$$

$$A_r \leqslant A_{r,max} \tag{4.11}$$

$$f \geqslant f_{min} \tag{4.12}$$

$$p.r \leqslant s \tag{4.13}$$

Given a search interval for r and p, this problem can be solved easily. For instance, a graphical approach is presented in section 5.5.1.

4.1.7 Application to the multiplier

If we apply this model to the pipelined multiplier, each block of the pipeline (Block i of figure 4.1) corresponds to a multiplier cell. The replicated element (Block i_j of figure 4.1) is the CSA with the associated multiplexers.

Some parameters are fixed by the way the multiplier is designed. As a matter of fact, if the bitwidth of the multiplier is equal to n:

- $S_p = n$ corresponds to the bitwidth of A
- $S_d = 2n$ corresponds to the bitwidth of B plus the bitwidth of M
- $S_i = 2n$ corresponds to the bitwidth of C plus the bitwidth of S which are stored internally

One iteration is performed at each clock cycle. Hence $c = 1$. A_l and A_r depends on the FPGA used. For instance, they can correspond respectively to the number of Slice LUTs and number of Slice registers available if we use a Xilinx FPGA. f_{min} is fixed by the designer.

The other parameters are harder to determine. For a Xilinx FPGA it is save to fix $A_{r,e} = 1$ Slice Register. $A_{l,e}$ and ρ can be approximated without synthesis if we have a good knowledge of the FPGA used (basically if we known how many LUTs each sub-block of the multiplier takes). However it is hard to find f_0 and λ without experiments. To find f_0, at least a synthesis with $r = 1$ has to be done. To find λ, we can run few syntheses for some values of r and do an interpolation. Hence, even if some syntheses have to be performed, the integration time of the multiplier can still be reduced by using our model.

4.2 Performance tuning software

In this section, we present a software integrating our multiplier model. It enables the designer to quickly tune the multiplier to the needs of its project.

4.2.1 Design

The "Multiplier parameters finder" software is coded in Python. The user interface is defined in XML with Glade. Using Glade totally separates the user interface definition from the core of the program, making the program clearer.

The "Multiplier parameters finder" has two main features:

1. Find the optimal number of replications and pipeline stages of the multiplier according to our model.

2. Run syntheses around the values of the r and p given by our model in order to possibly find a better optimum.

The inputs of the first feature are the different parameters of our model, discussed in the previous section. The outputs are the values of r and p given by our model and the corresponding logic/register area, maximum frequency and throughput values.

The inputs of the second feature are the minimum and maximum values of r and p for which we want to run syntheses. The outputs are the optimum synthesized values of r and p in this range together with the corresponding logic/register area, maximum frequency and throughput values.

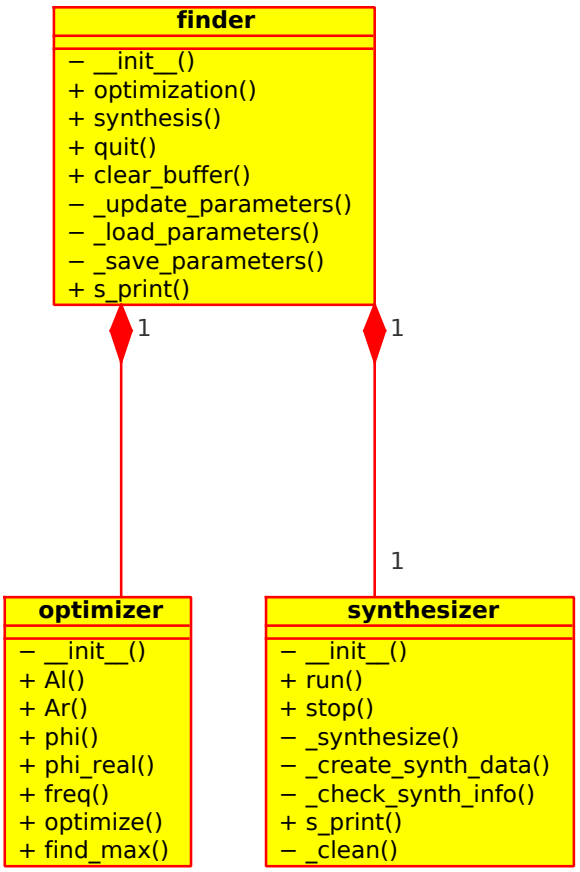


Figure 4.2: UML diagram of the "Multiplier parameters finder" software

The UML diagram of our software is given in figure 4.2. The `finder` class associates functions to the user interface events and is the entry point of the program. It is composed of an `optimizer` object which performs the optimization of our model through the `optimize()` function. This function is called by the `optimization()` function of the `finder` class with the parameters given by the user. Its results are printed in the user interface.

The `finder` class also has a `synthesizer` object whose role is to interface with the synthesis software[2] in order to perform the syntheses asked by the `finder`. The synthesis process can be run and stop thanks to the eponymous functions. This behaviour is managed by the `finder` through the `synthesis()` function. This function:

- takes the ranges for r and p given by the user

- runs the simulation and gets their results

[2]The synthesizer interfaces with ISE synthesis tools.

- finds the optimal values of r and p for the synthesized designs thanks to the `optimizer`'s `find_max()` function
- prints the results in the user interface

Note that the function `optimize()` uses our model's approximation of the throughput (returned by the function `phi()`) whereas `find_max()` uses the real multiplier throughput (returned by the function `phi_real()`).

4.2.2 User interface

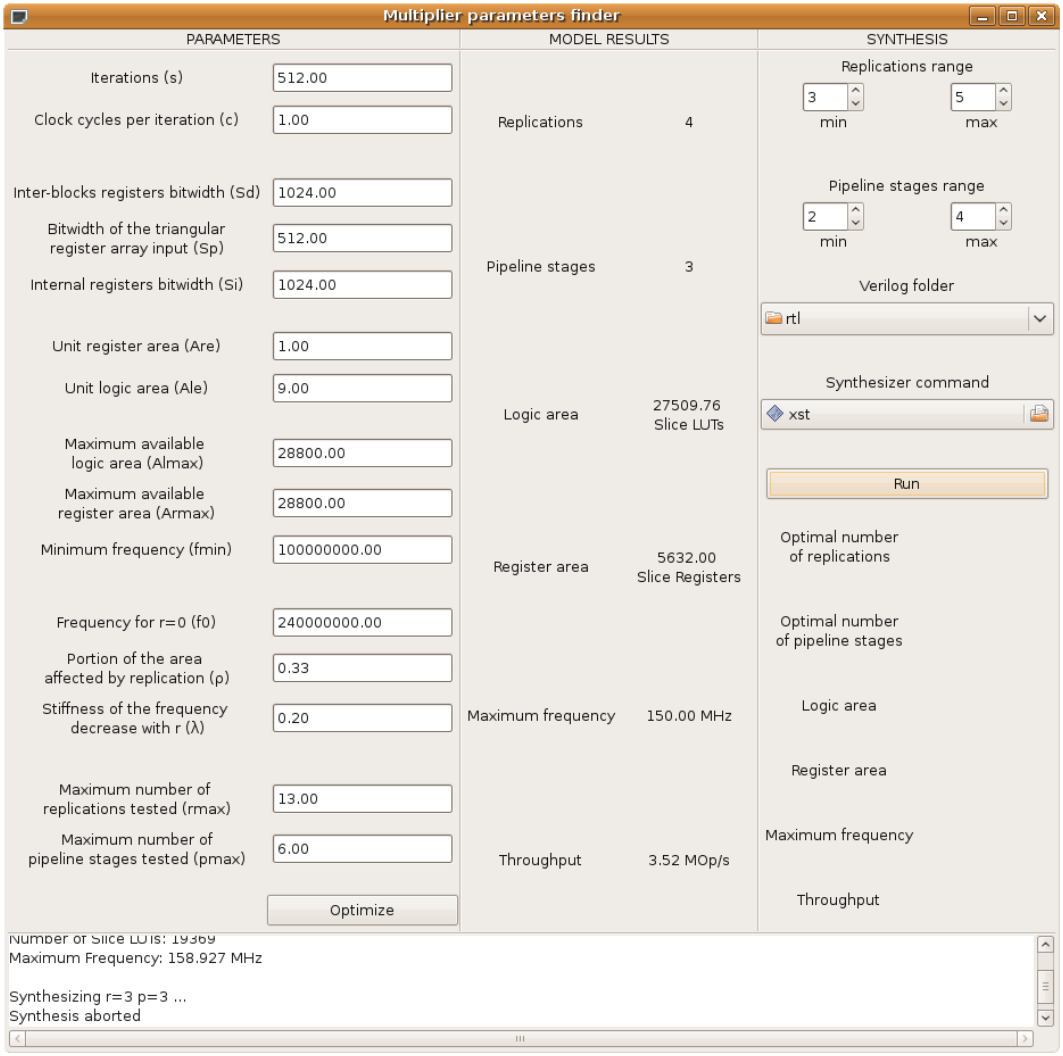


Figure 4.3: User interface of the "Multiplier parameters finder" software

The user interface is represented in figure 4.3. It consists of three panels. The left most panel enables the user to choose the parameters of the model and to run the optimisation by clicking on the `Optimize` button. The central panel gives the results of the optimisation. Finally the right most panel gives the necessary controls to synthesize the multiplier for different values of r and p around the values given by the model. The user chooses the directory containing the Verilog files of the multiplier and the synthesizer. Then the user clicks on the `Run` button. Once the syntheses are started, this button is changes to a `Stop` button enabling the user to abort the synthesis. The optimal values of r and p are also printed in this panel after all the syntheses have been performed.

The text area at the bottom gives the following information about each synthesis:

- for which values of r and p the multiplier is being synthesized

- the state of the synthesis: done or aborted
- the maximum frequency, logic and register area for each synthesis performed

4.3 Summary

In this chapter, a model of a design using our pipelining and replication techniques has been elaborated. With some a priori information about the design, it can deduce an approximation of the optimal numbers of pipeline stages and replications optimising the throughput of the design.

To verify that this model fits well our Montgomery multiplier, we have created a simple program putting together our model and some synthesis functionalities of the ISE toolsuite. Our software is also useful to the designer who wish to integrate our Montgomery multiplier into its design. In that case, one can use it to quickly tune the multiplier to the design requirements.

Chapter 5

Results and evaluation

In this chapter, our designs and multiplier's model are evaluated quantitatively and qualitatively. Section 5.1 shows the design space explored by our multiplier and compares an n-bit multiplier with final subtraction against the (n+3)-bit version without final subtraction. Section 5.2 evaluates our exponentiator. We highlight the benefits of using pipelined adders and subtractors for its multiplier. We also study the effects of pipelining and replicating the multiplier on the performance of the exponentiator. Section 5.3 shows results about the design space explored by our prime tester and its performance against the number of pipeline stages and replications of the multiplier. Section 5.4 is a short analysis of the power consumption of our three modules. Section 5.5 evaluates our multiplier's model against real synthesis results. Finally, section 5.6 compares the performance of our synthesized designs against existing software and hardware implementations and section 5.7 sums up the chapter.

For all experiments involving synthesis, we use Xilinx ISE WebPack 11.1 with "speed" as the optimisation mode and "normal" as the optimisation level. We use a Xilinx XC5VLX50T FPGA.

5.1 Multiplier

5.1.1 Impacts of pipelining and replication

We set $n = 512$ and synthesize the multiplier (with no final subtraction) for r from 1 to 12 and p from 1 to 6. We do not use larger bitwidths for our experiments because it would take too long to run all the syntheses.

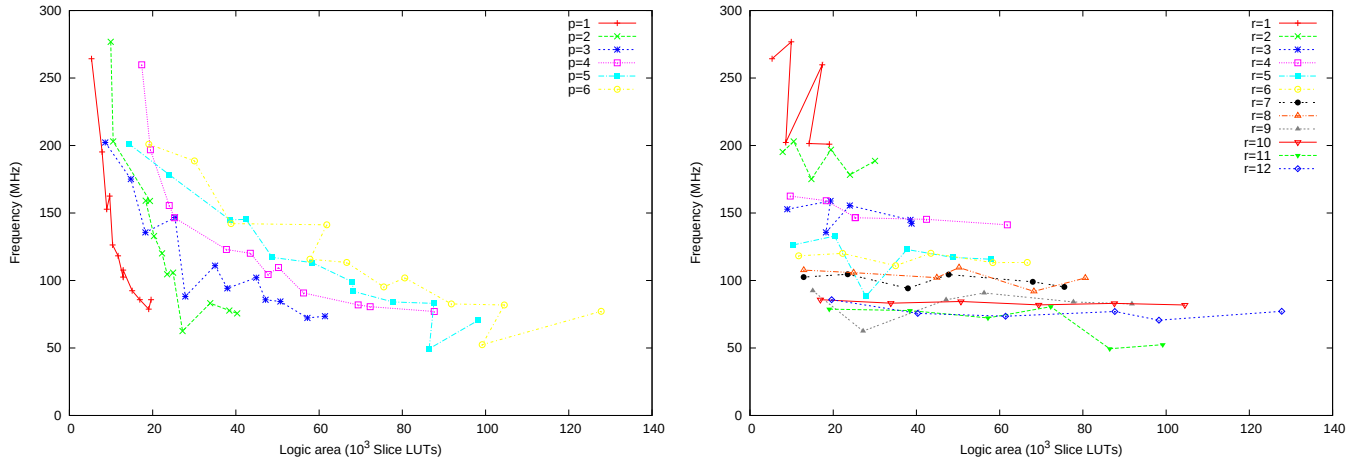


Figure 5.1: Maximum frequency of the multiplier against the logic area for $n = 512$, $p \in [1, 6]$ and $r \in [1, 12]$

The plots show respectively how the number of pipeline stages (p) and the number of replications

(r) affect the area and speed of the multiplier.

Figure 5.1 shows the maximum frequency at which the design can run against the area taken by the logic. We can see that the area increases with both the number of pipeline stages and the number of replications. As predicted, the first plot shows that when the number of pipeline stages doubles, the logic area doubles. The second plot shows that the variation of area with the number of replications is less important. In fact, only a portion of the total area is affected by the replication. The second plot also confirms that the maximum frequency does not depend a lot on the number of pipeline stages.

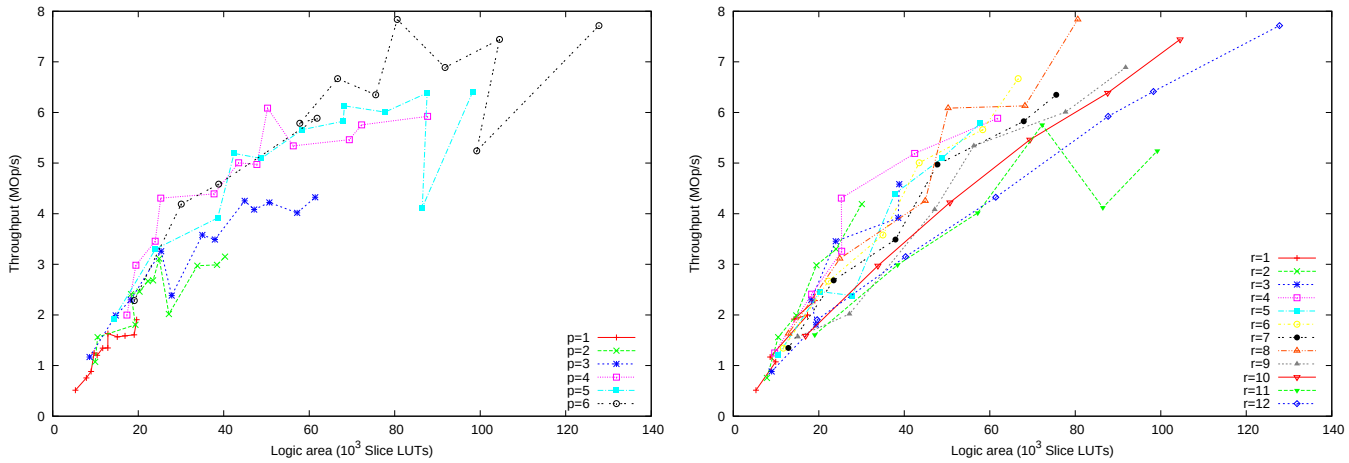


Figure 5.2: Design space explored by the multiplier for $n = 512$, $p \in [1, 6]$ and r in $[1, 12]$

Figure 5.2 puts forward the design space explored by our multiplier by plotting the achievable throughput against the logic area for the considered values or r and p. The second plot shows that at equal r, the throughput always increases with the number of pipeline stages. Apart from some outliers, the first plot shows that the throughput also increases with the number of replication at equal p for $r \leqslant 10$. However, for $r > 10$, the throughput seems to reach an asymptote and even decrease for certain values of p. This behaviour happens when the lose of maximum frequency due to the increase of the critical path outweighs the decrease in the number of iterations needed to compute a multiplication.

The first plot of figure 5.2 shows some surprising points. For example, for $p = 5$ and $r = 12$, the throughput and the area are abnormally low compared to the general trend. We suspect this behaviour to be due to the optimisations performed by the synthesizer which basically tries to optimise the speed of the module under area constraints. These optimisations are not under full control of the user indeed.

5.1.2 Final subtraction against $n + 3$ multiplier

Figure 5.3 represents the throughput and the latency of the multiplier against the area taken by the logic for a 512-bit multiplier with final subtraction and a 515-bit multiplier without final subtraction.

We see that using the $n+3$ multiplier instead of the n multiplier with final subtraction decreases the throughput of our multiplier. This is not due to the fact that we perform 3 more iterations but to a decrease in the maximum frequency. In fact using a 515 bit multiplier increases the critical path of the multiplier. More surprisingly, the area taken by a multiplier without subtraction is not always less than the area taken by a multiplier with final subtraction. This is only the case for $r \in \{1, 3, 5\}$. This phenomenon is certainly due to the fact that the synthesis software more easily optimizes the area for submodules bitwidths that are a multiple of 2, leading to the same area performance for $n = 512$ as for $n = 515$ when $r \in \{2, 4\}$, even if the subtractor is removed for $n = 515$.

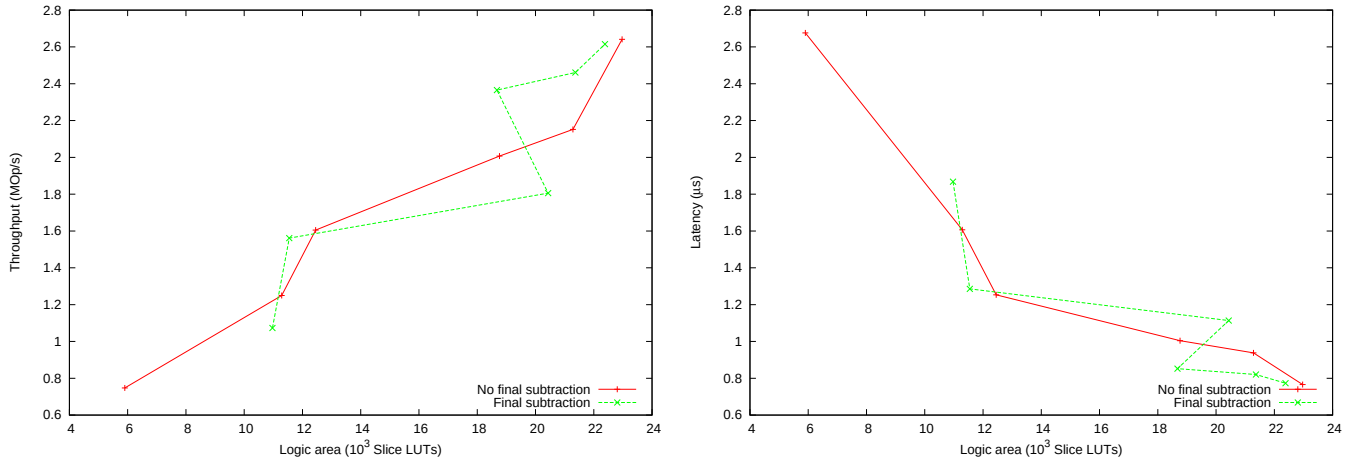


Figure 5.3: Final subtraction versus $n + 3$ multiplier for $p = 2$ and r in $[1, 6]$

5.2 Exponentiator

5.2.1 Overhead introduced by the exponentiator logic

The critical path problem

As predicted, the exponentiator takes about the same logic area as the multiplier, the area taken by the control logic of the exponentiator being negligible (see figure 5.4). It is not the case for the maximum frequency which is reduced considerably if pipeline adders/subtractors are not used. This is shown in figure 5.5.

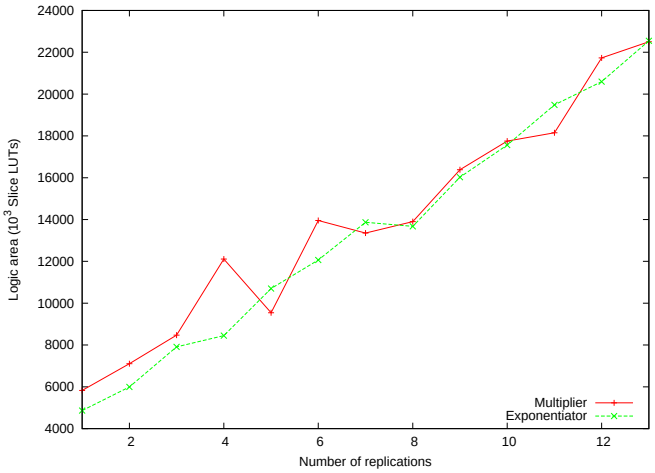


Figure 5.4: Logic area of the multiplier and the exponentiator against r for a bitwidth of 512 bits and $p = 1$

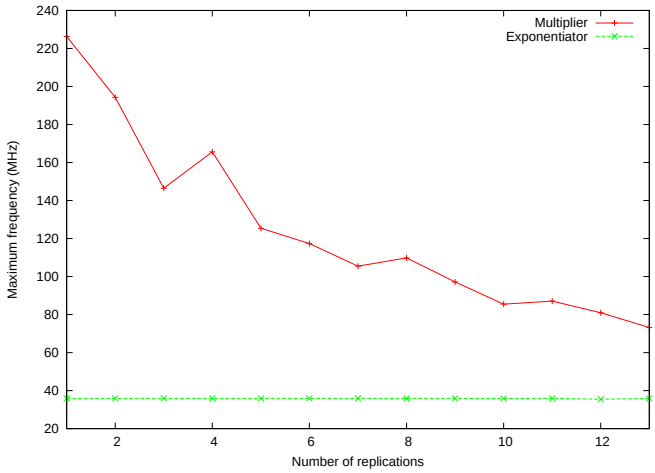


Figure 5.5: Maximum frequency of the multiplier and the exponentiator against r for a bitwidth of 512 bits and $p = 1$

Advantages of the pipeline adders

Figure 5.6 compares the maximum frequencies reachable for different depths of the adders' pipeline. If we use normal adders, the frequency is very low and does not depend on r. Using 2-stage pipeline adders increases the frequency by a factor of 2 to 3. We can see that for above 8 pipeline stages, the latency of the adders are no longer affecting the critical path. The gain of increasing the number of adder pipeline stages is almost null.

Another interesting observation is that for more than 2 pipeline stages, all the frequencies seem to converge when r increases. An analysis of the synthesis results shows that the more r increases,

the more the delay due to routing increases along the critical path. For $r = 13$ the routing delay overweights the gate delay through the adders, making all the frequencies converging to the same point.

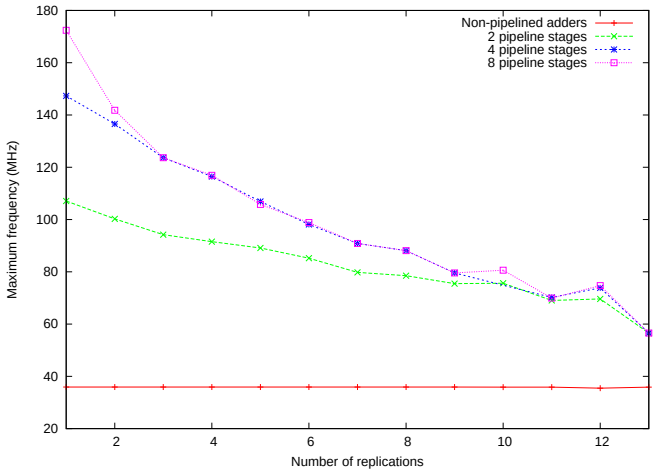


Figure 5.6: Influence of the pipeline adders on the maximum frequency of the exponentiator

Effect of the pipeline adders on the latency and throughput of the design

Figures 5.7 shows that the number of cycles taken by an exponentiation slightly increases with the pipeline depth of the adders. However, even if the numbers of clock cycles increases, the latency of the exponentiator clearly decreases when we use the pipelined adders as shows figure 5.8. This last figure also confirms that using more than 8 pipeline stages for the adders is useless in the conditions of our experiment and can even have a negative impact on the latency. As a matter of fact, the number of clock cycles keeps on increasing whereas the maximum frequency does not increase any more.

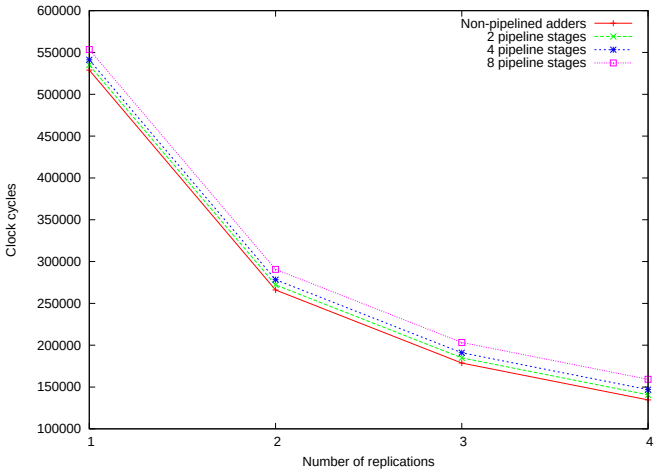


Figure 5.7: Clock cycles taken by the exponentiation with different pipeline depths for the adders (p=1 and $r \in [1, 4]$)

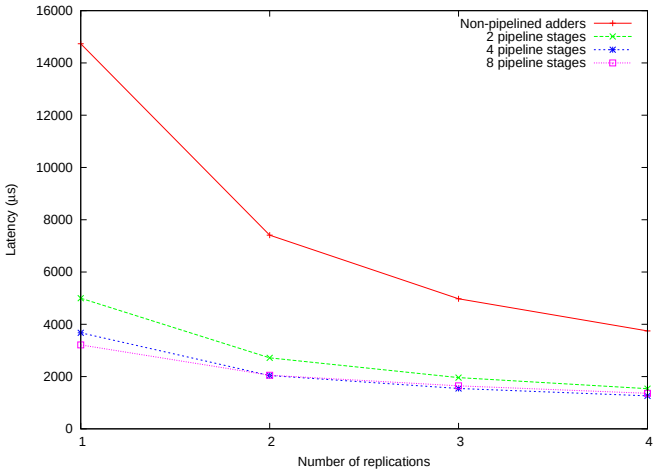


Figure 5.8: Latency of an exponentiation with different pipeline depths for the adders (p=1 and $r \in [1, 4]$)

5.2.2 Using pipelining and replication

In this section, we set the pipeline depth of the adders to 4. For a bitwidth of 512 bits, this value is relevant as shown in the last section.

Figures 5.9 and 5.10 show the influence of the number of multiplier pipeline stages (p) and replications (r) on the exponentiator. Compared with a non-pipelined multiplier, using a multiplier with 2 pipeline stages decreases the latency of the exponentiator by almost a factor of 2. For $p > 2$ the number of clock cycles and the latency is no longer improved by increasing the number of pipeline stages. As a matter of fact, the exponentiator cannot keep the multiplier's pipeline full as was demonstrated in section 3.4.3. Therefore, there is no advantage to using more than 2 pipeline stages for the multiplier in that case, as this will increase the area taken by the design with no or even a negative effect on its latency.

Figure 5.9 also shows that the number of clock cycles is almost divided by 2 when the number of replications doubles. Figure 5.10 highlights another interesting point: when r increases the latency seems to reach an horizontal asymptote. This behaviour has already been observed with the throughput of the multiplier. We recall that this phenomenon happens when the decrease of maximum frequency with r begins to overweight the decrease in clock cycles.

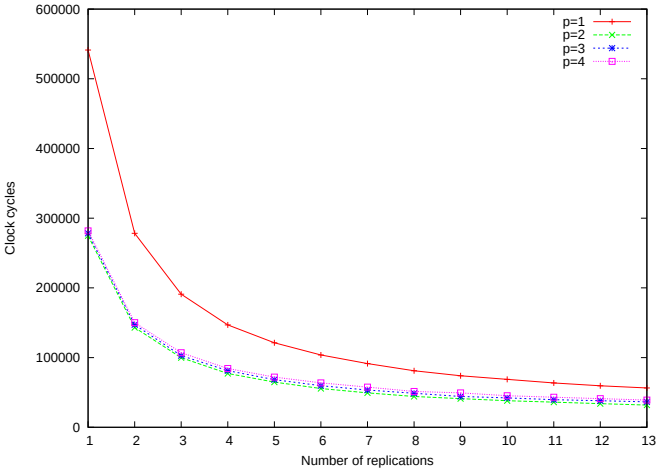


Figure 5.9: Clock cycles taken by the exponentiation against p and r
(the adders pipeline depth is set to 4)

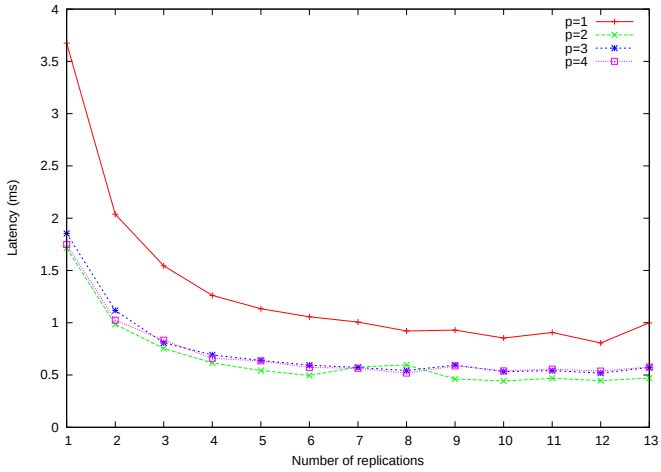


Figure 5.10: Latency of an exponentiation against p and r
(the adders pipeline depth is set to 4)

5.3 Prime tester

5.3.1 Global performance

Figure 5.11 shows the design space explored by the prime tester. It plots the maximum frequency against the logic area taken by our design.

We see that the prime tester takes more area than the exponentiator at equal p and r. This increase in area is due to the extra logic needed to perform the primality test: in particular an n-bit comparator and an n-bit subtractor (to compute $p - 1$).

The maximum frequency of the prime tester is slightly lower than the one of the exponentiator. This certainly comes from the increase in the routine delay introduced by the extra hardware.

5.3.2 Use of the pipelined multiplier

Figure 5.12 and 5.13 show the mean number of clock cycles taken by a prime test on a 512-bit number together with its latency for different values of p and r. A pipeline depth of 4 is used for the adders/subtractors of the multiplier.

Unlike the other modules, the number of clock cycles taken by the prime tester depends on the number under test. The results of clock cycles are obtained by simulation of the design using Verilator. Note that the simulation of a prime test takes a very long time for a 512-bit number. Hence, each point corresponds to a mean on only 10 tests.

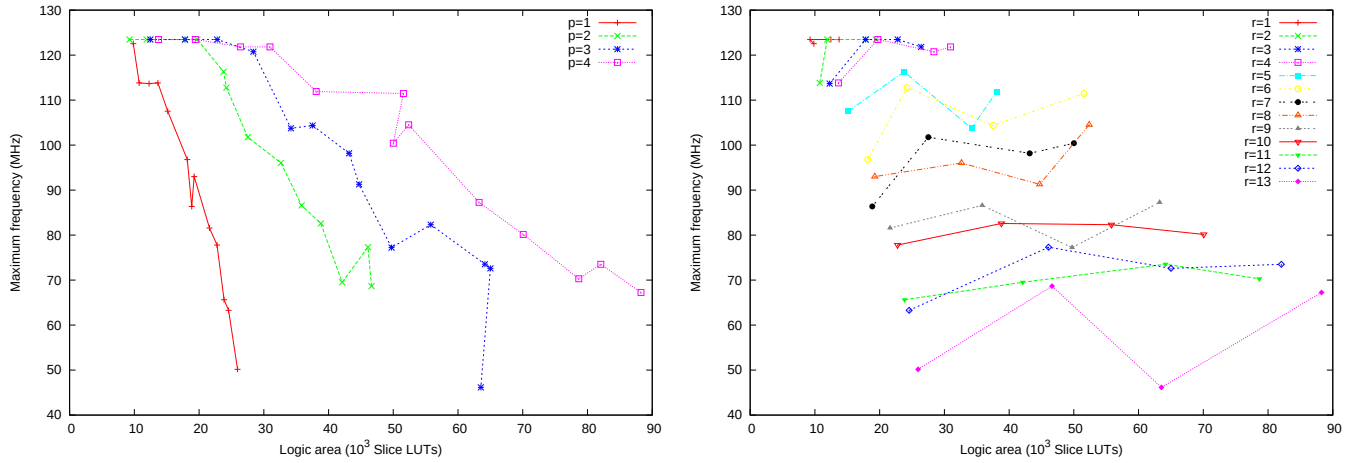


Figure 5.11: Design space explored by the prime tester for $n = 512$, $p \in [1, 6]$ and r in $[1, 13]$ (the pipeline depth of the adders is set to 4)

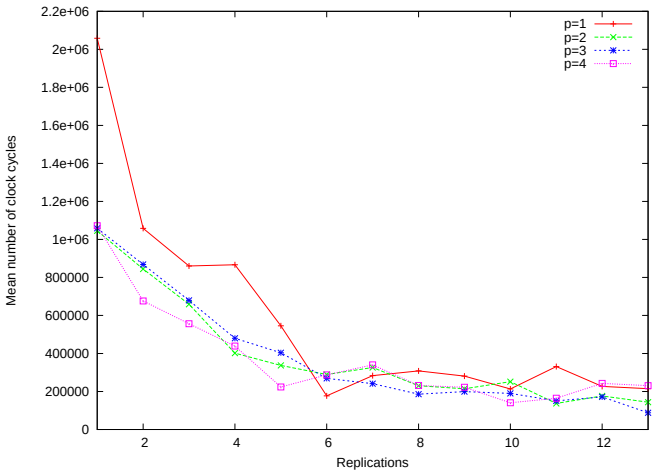


Figure 5.12: Mean clock cycles taken by the prime test against p and r (the adders pipeline depth is set to 4)

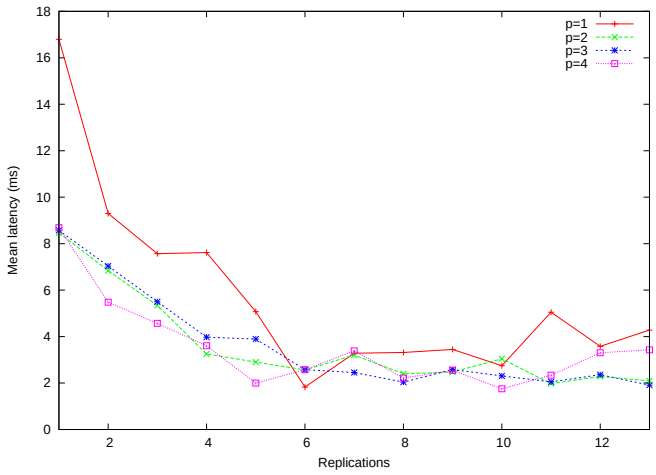


Figure 5.13: Mean latency of a prime test against p and r (the adders pipeline depth is set to 4)

Figure 5.12 confirms that using the pipeline capabilities of the multiplier through the exponentiator decreases the number of clock cycle needed to perform a prime test in a non-negligible maniere, at least for $r \leqslant 5$. The fact that we cannot distinguish this trend for $r \geqslant 6$ may be due to the fact that not enough points could be taken to compute the mean number of clock cycles.

For the same reasons as the exponentiator, using a pipeline depth of more than 2 is useless. When we double the number of replications, the number of clock cycles is divided by a bit less than 2. The horizontal asymptote is less obvious in figure 5.13 than it was for the exponentiator and the multiplier. However, the prime tester uses these two modules. Therefore, its latency cannot decrease infinitely with the number of replications as figured out in section 5.2.2.

5.4 Power consumption

In this section we use Xilinx XPower Estimator to get a gross approximation of the power consumed by our three modules. We only consider the power consumed by the logic cells and the RAMs. To allow a fair comparison we keep the frequency constant at an arbitrary value of 150 MHz. The leakage power is equal to 560 mW for every module and every value of r and p. This analysis based on the synthesis results has a limited accuracy and only the relative consumptions can be considered safely.

Figures 5.14 to 5.16 show that the general trend is an increase of power consumption with the number of replication and the number of pipeline stages. This increase is directly linked to the increase of the number of Slice LUTs and Slice registers taken by the module. However for $r \leqslant 2$ the power consumption of the prime tester is higher with 1 than with 2 pipeline stages. This phenomenon also appears in the exponentiator power consumption for $r = 1$. The way the exponentiator control is designed for $p = 1$ makes the synthesis tool implement the registers saving the intermediate values of P and Z in a RAM, whereas they are implemented as simple registers for $p = 2$. For $r \leqslant 2$ and $p = 1$, the power consumed by this RAM overweights the increase of logic power consumption for $r \leqslant 2$ and $p = 2$, which explains this strange trend.

In figure 5.17, we can see that our prime tester consumes more than our exponentiator which consumes more than our muliplier at equal p and r. This trend is simply explained by the extra hardware introduced in the upper level modules, resulting in an increase in logic area and therefore in power consumption.

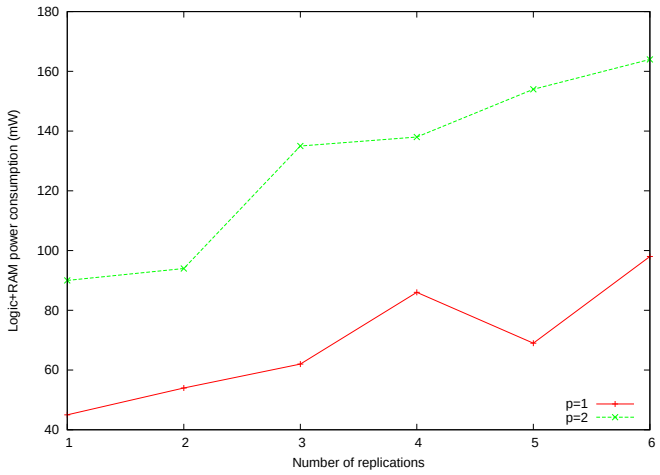


Figure 5.14: Total logic+RAM power consumption of the multiplier against p and r
(the adders pipeline depth is set to 4)

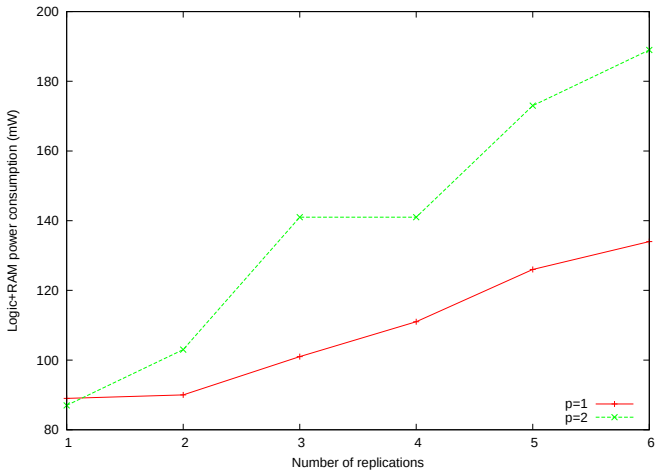


Figure 5.15: Total logic+RAM power consumption of the exponentiator against p and r
(the adders pipeline depth is set to 4)

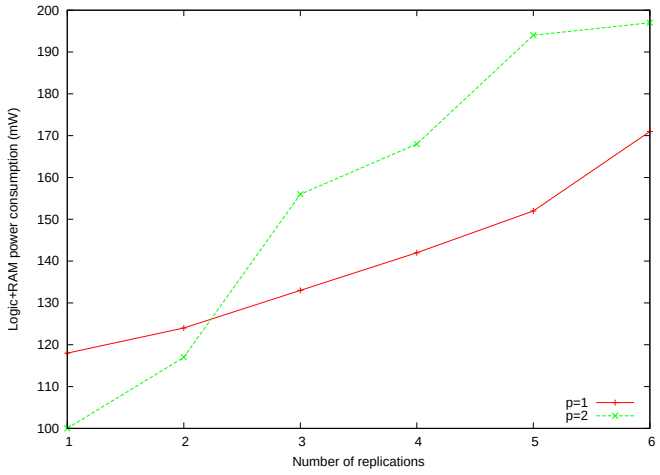


Figure 5.16: Total logic+RAM power consumption of the prime tester against p and r
(the adders pipeline depth is set to 4)

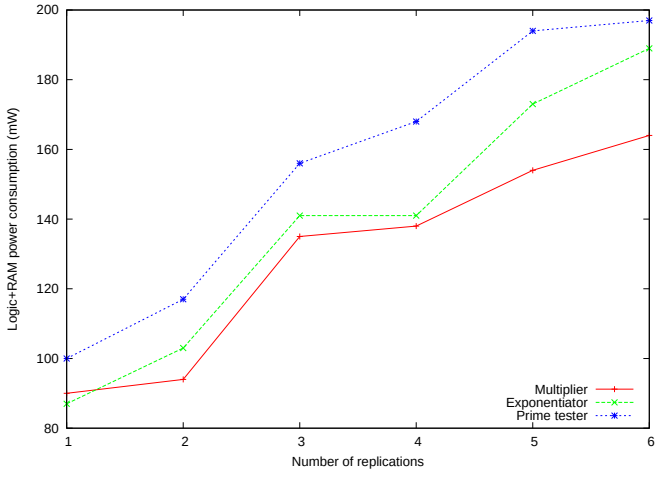


Figure 5.17: Comparison of the power consumptions of our modules for $p = 2$ and different values of r

5.5 Evaluation of our performance tuning tools

5.5.1 Accuracy of the model

Parameters

Let us consider our Montgomery multiplier with final subtraction. We choose a bitwidth of 512 bits leading to:

- $s = 512$
- $S_d = 1024$
- $S_p = 512$
- $S_i = 1024$

We still use a Xilinx XC5VLX50T FPGA, for which:

- $A_{l,max} = 28\ 800$ Slice LUTs
- $A_{r,max} = 28\ 800$ Slice Registers

The algorithm used imposes $c = 1$ and by running pre-syntheses we determine:

- $A_{r,e} = 1$ Slice Registers
- $\rho \approx 33\%$
- $A_{l,e} \approx 9$ Slice LUTs
- $\lambda = 0.20$

Finally we do a synthesis for $r = 1$ and $p = 1$ to determine $f_0 = 226$ MHz and we impose $f_{min} = 100$ MHz.

Frequency

Figure 5.18 compares the variations of the maximum frequency obtained experimentally to our model. It shows that our model is quite accurate for most values of p. However, we had to run pre-syntheses to get the right trend of the frequency. Getting this trend without prior syntheses turns out to be very hard. The idea is to get some points by doing several syntheses and extrapolate in order to get the global frequency trend.

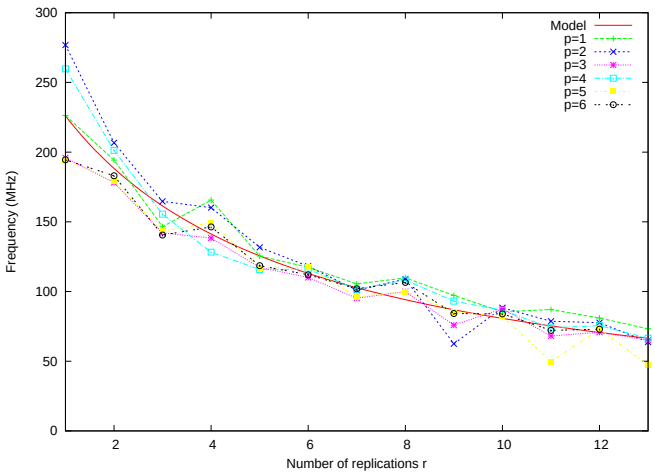


Figure 5.18: Frequency against r for different values of p

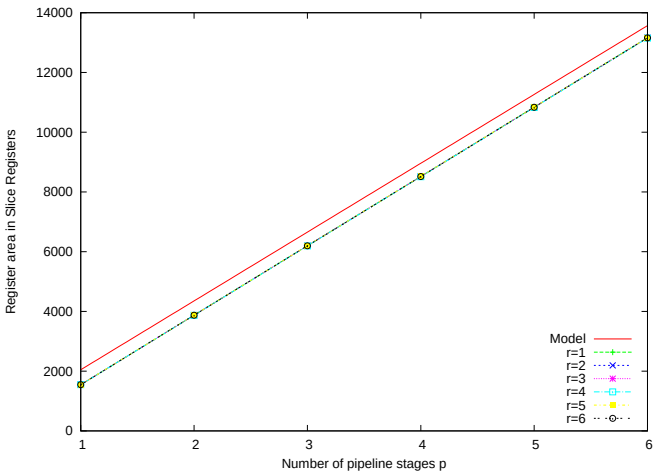


Figure 5.19: Area taken by the registers against p for different values of r

Area

The predicted areas taken by the registers for different values of p are very close to the real ones as shows figure 5.19. The mapping one-bit register/one slice register is always respected indeed.

The offset of 512 bit is due to the fact that in our actual design, the last register of the pipeline only saves M instead of B and M. As a matter of fact, B is not used by the logic (final addition and subtraction) connected to the output of the pipeline. Note that the registers used for the state machine are also not taken into account.

Our model is less accurate to predict the area taken by the logic (figures 5.20 and 5.21). These figures show that the variations of the number of Slice LUTs are not perfectly linear with r and p. It should be in part due to the optimisations performed by the synthesis software which tries to find the best occupation ratio of the slices.

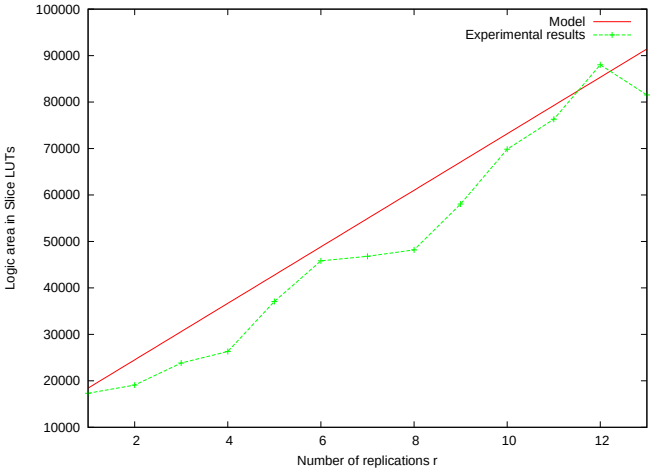


Figure 5.20: Area taken by the logic against r for $p = 4$

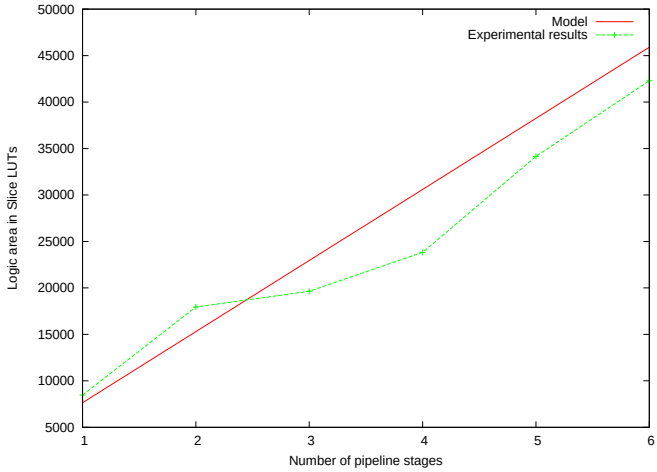


Figure 5.21: Area taken by the logic against p for $r = 3$

Throughput

Figure 5.22 represents the area taken by logic (shaded color tones) in Slice LUTs and the throughput predicted by our model (at each intersection of the grid) in Mop/s against r and p. The values of r and p optimising the throughput under the constraints of equations 4.10 to 4.13 are, according to our model:

$$r_{model} = 4$$
$$p_{model} = 3$$

	Model	Experiments
f (MHz)	141	138
ϕ_{enc} (Mop/s)	3.31	3.08
A_l (Slice LUTs)	27509	27102
A_r (Slice registers)	6656	6193

Table 5.1: Theoretical and experimental results for $r_{model} = 4$ and $p_{model} = 3$

Table 5.1 sums up our theoretical and experimental results for these values of r and p. All the parameters are predicted will less than 10% of error. The real optimum is actually:

$$\phi_{enc,opt} = 3,8 \text{ Mop/s}$$

for:

$$r_{opt} = 4$$
$$p_{opt} = 4$$

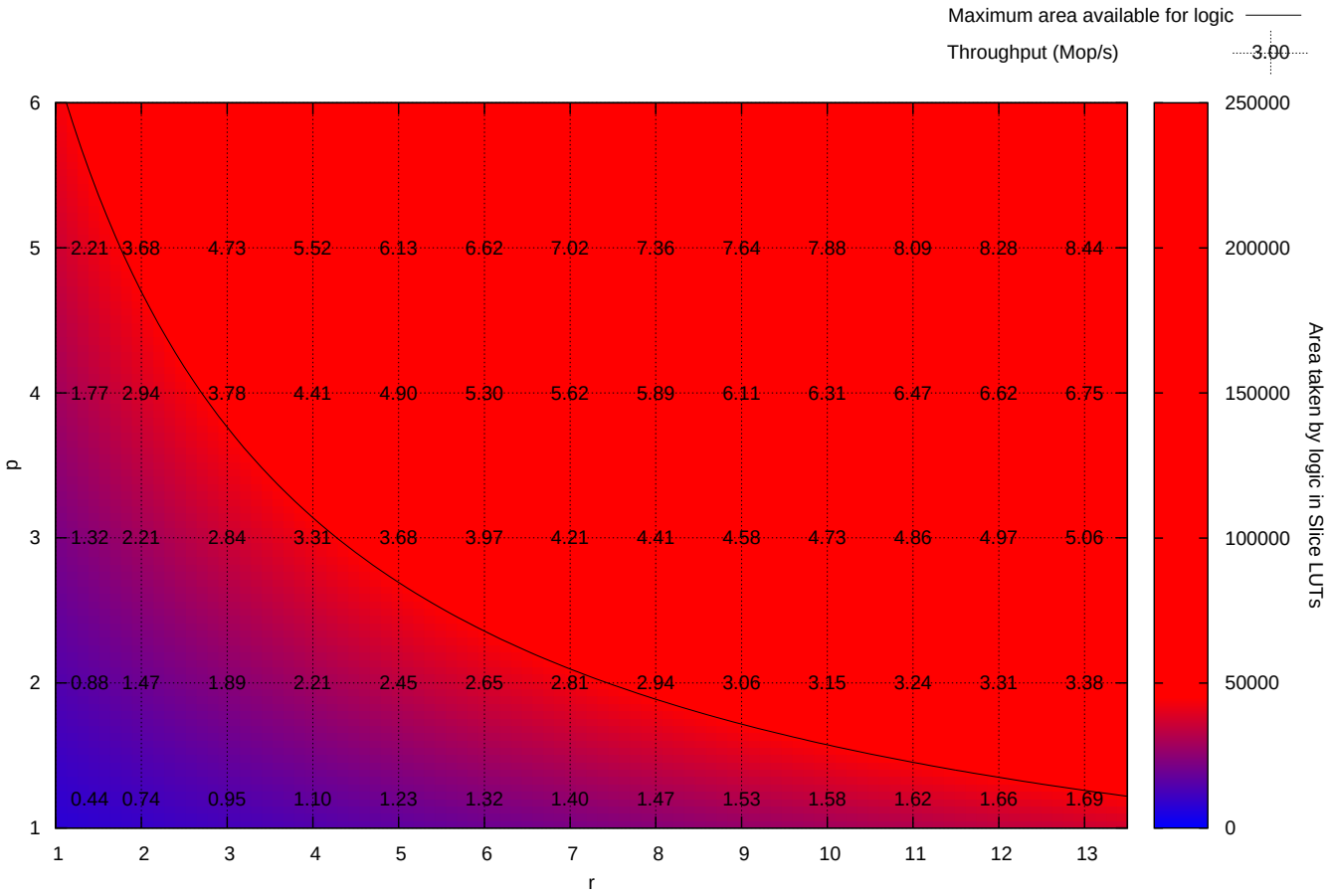


Figure 5.22: Area (in Slice LUTs) and throughput (in Mop/s) against r and p

and for which:

$$f_{opt} = 128 \text{ MHz}$$
$$A_{l,opt} = 26306 \text{ Slice LUTs}$$
$$A_{r,opt} = 8511 \text{ Slice registers}$$

Our model is therefore closed to the real optimum. It did not actually find the real optimum as the optimisations performed by the synthesis tool for $r = 4$ made the multiplier fit in less logic area for $p = 4$ than for $p = 3$ (26306 Slice LUTs versus 27509 Slice LUTs).

5.5.2 Shortcomings of this model

Even if this model is detailed, it has several shortcomings. First, we assumed that increasing the length of the pipeline increases the value of the throughput by the same factor. That is true if we can always maintain the pipeline full. In practice, this is limited by the data dependencies between consecutive values as shown in section 3.4.3.

Second, we realised in sections 5.5.1 and 5.5.1 that the area taken by the logic does not follow a simple law and is dependent to the optimisations made by the synthesis tool.

Finally, the values of ρ, f_0, λ, $A_{r,e}$ and $A_{l,e}$ are quite hard to determine without running pre-syntheses.

5.5.3 Possible uses

This model can help us tuning our parametric RSA design when targeting a particular FPGA. By giving insights about the speed/area tradeoffs, the use of this model clearly reduces the time-consuming experiments needed to find the best parameters. In that case a rough estimation of the

frequency and area are sufficient. What matters is to capture the trends of f, A_l and A_r. Section 5.5.1 demonstrate that these requirements are met.

Our model can also be useful to find the best values for p and r at an early stage of the design, that is without being able to do experiments yet. In that case, a sub-optimal solution fitting into the FPGA (as in section 5.5.1) may be preferred to a better one whose area consumption turns out to be under-estimated. This constraint can be met by taking care of choosing values for $A_{l,e}$ and ρ that are over-estimated enough and a gross approximation for f_0 and λ.

5.6 Is our design better than existing implementations?

5.6.1 Software

We compare the implementation of our modular multiplier, modular exponentiator and primality test on a XC5VLX50T FPGA with the software implementations of the GMP multiprecision arithmetic library [21] version 4.2.4. The functions of this library performing modular exponentiation and primality test are respectively `mpz_powm` and `mpz_probab_prime_p`. The software modular multiplication is performed with low level multiplication and modulus functions. All these functions are very optimised and fast.

The machine used has an Intel Core 2 Duo CPU E7400 running at 2.80GHz. Each core has a L1 32 KB instruction cache and 32 KB data cache. A 3MB 8-way set associative cache is shared by both cores. The machine has 3 GB of DDR2 RAM. The test program is compiled with the Intel Compiler v11.0 using the compilation option `fast`. The runtimes given are an average on 10 000 tests with random inputs on 512 bits.

Multiplier

The mean runtime for the software to perform a 512-bit multiplication is 1.96 μs. Figure 5.23 gives the speedup obtained by using our hardware multiplier over the software version for different values of r and p. It shows that the more we increase the number of pipeline stages, the higher speedup we get. This is also true for the number of replications until we reach the asymptote evoked previously.

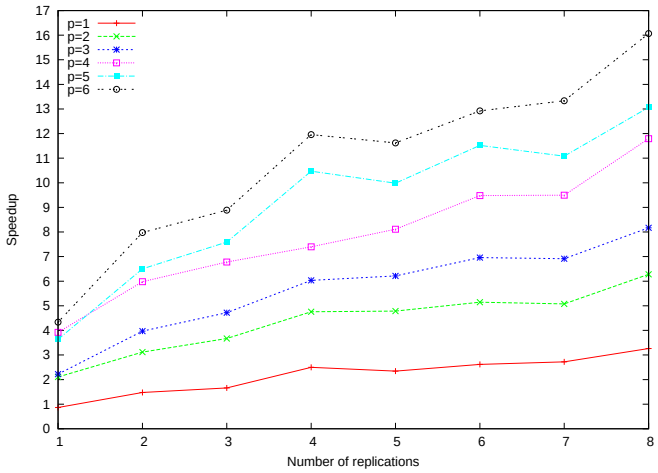


Figure 5.23: Multiplier: speedup obtained with our hardware version over an optimised software version

The best results given by this experiment is a speedup of 16 for $p = 6$ and $r = 8$. This speedup can be improved as desired by increasing the number of pipeline stages, the only limitation being the area available on the FPGA.

Exponentiator

The mean runtime for the software version to perform a 512-bit exponentiation is 0.65 ms. Figure 5.24 compares this mean runtime to the latency of our hardware exponentiator for $p \in [1, 2]$ and a pipeline depth of 4 for the adders.

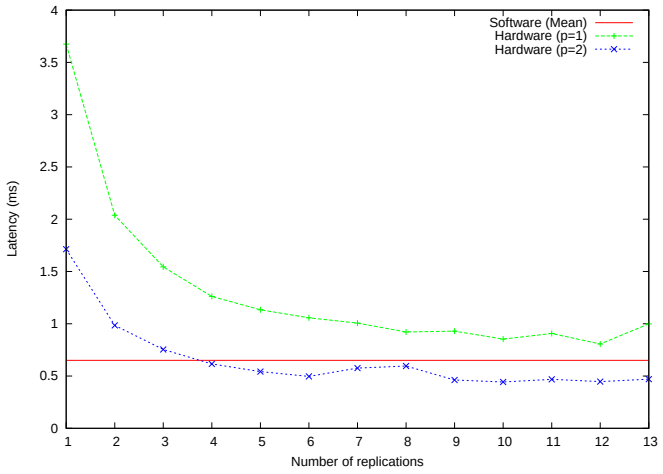


Figure 5.24: Exponentiator: software versus hardware version

For $p = 1$, our design cannot beat the software implementation. For $p = 2$, from $r = 4$ our design running at around 100 MHz and taking about 16 000 Slice LUTs is faster than the software version on a Core 2 Duo running at 2.80 GHz with a 1.5 speedup. However the asymptote reached by the latency could prevent us from finding an r such that our exponentiator can beat every software implementation on every processor.

Note that we could certainly improve the way the exponentiator manages the pipelined multiplier to gain some clock cycles and still reduce the latency of the exponentiator.

Prime tester

The mean runtime to test a random number in software is 0.15 ms. At first glance, this result seems very surprising. In fact, on average the software takes less time to perform a primality test than to compute a single exponentiation! It turns out that the primality test implemented in the `mpz_probab_prime_p` is done in two stages:

- First, some trial divisions are performed on the number under test
- If the number under test is not divisible by any of the numbers tried in the first test, the Rabin-Miller test is performed

Note that a last test may also be performed to confirm that the number is prime. This last step can for instance be a Lucas test. Such a test is not done by the `mpz_probab_prime_p` function [21].

To understand the results given by the software, let us do a quick analysis of a three-stage prime tester. A possible configuration is represented in figure 5.25. The first block performs trial divisions. It takes a time t_1. The second block is a primality test (like a Miller-Rabin test) and takes time t_2. The third block confirms the result of block 2 when the latter finds a possible prime number. It takes time t_3.

Let $P = \{p_0, .., p_k\}$ the set of the $k+1$ first prime numbers with $p_0 = 2$ and N a random number of size n. We have the following probability[1]:

$$P(p \nmid N \quad \forall p \in P) = \prod_{i=0}^{k} (1 - \frac{1}{p_i}) \tag{5.1}$$

$$\tag{5.2}$$

[1] $|$ meaning “divides”

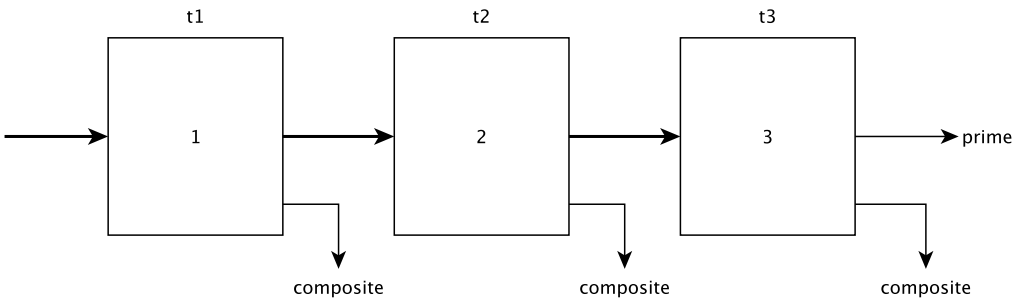


Figure 5.25: A prime number generator

From the prime number theorem, if N is close to 2^n, we also have:

$$P(N \text{ is prime}) \approx \frac{1}{ln(2^n)} = \frac{1}{n.ln(2)} \tag{5.3}$$

Hence neglecting the fact that test 2 is not 100% accurate, the mean time taken to determine if N is prime is:

$$t_m \approx t_1 + \prod_{i=0}^{k}(1 - \frac{1}{p_i})t_2 + \frac{1}{n.ln(2)}t_3 \tag{5.4}$$

Figures 5.26 and 5.27 give the mean time spent in block 2 and 3 for different values of n and k. The time spent in block 1 is always t_1.

The time spent in block 3 quickly decreases with the bitwidth. For a common bitwidth of 512, it is less than 1% of t_3.

The time spent in block 2 decreases with the number of primes used for trial division. For trial divisions with 20 prime numbers, it is around 13% of t_2. Note that it also means that by using trial divisions with the 20 first prime numbers, more than 80% of the non-primes are declared composite in the first stage. If we manage to perform fast trial divisions, we can clearly reduce the average runtime of a prime tester to less than the time taken by an exponentiation. That is why the software implementation is so fast.

This result also shows that the simple trial division process is very important and should be implemented before integrating our Rabin-Miller prime tester into a complete RSA cryptosystem.

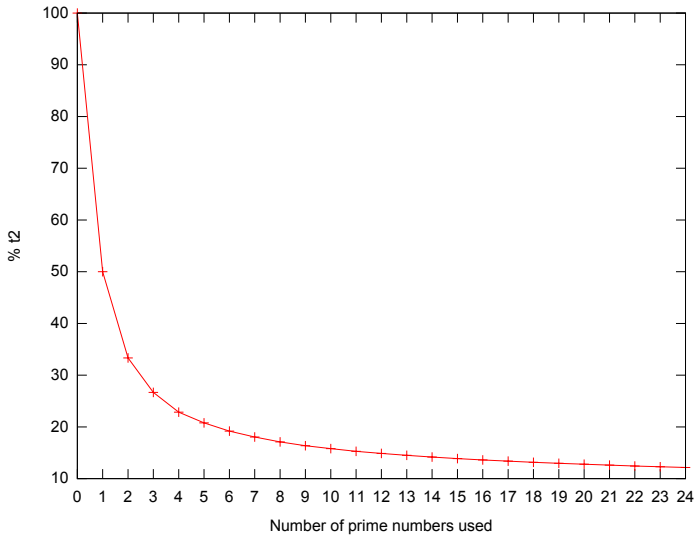


Figure 5.26: Time spent on block 2 for different values of p

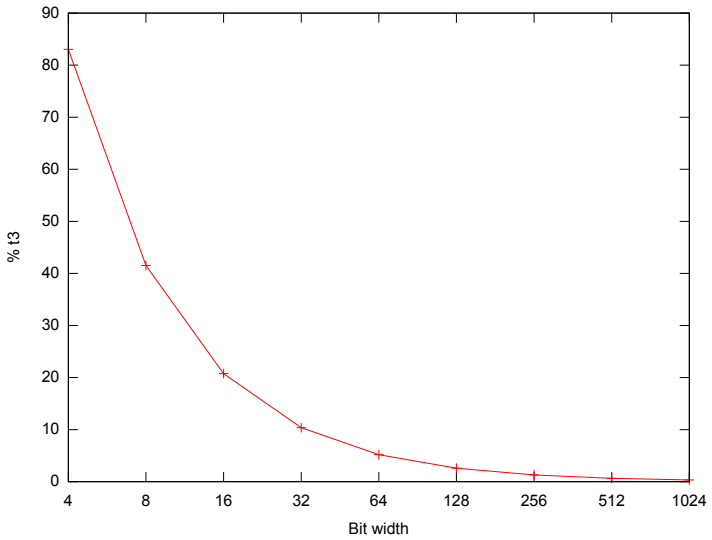


Figure 5.27: Time spent on block 3 for different values of n

To confirm the previous analysis we now run tests giving only prime numbers as input to the prime tester. Doing so obliges the software implementation to perform the Rabin-Miller test and enable us to fairly compare both implementations. This time, the mean software runtime increases to 5.8 ms.

Figure 5.28 compares our hardware implementation with the GMP software implementation when only prime numbers are given as inputs to test. As with the exponentiator, a pipeline depth of 1 cannot beat the software implementation. For $p = 2$, our design is faster than the software implementation when r is greater than 4 with a speedup of around 1.6. The asymptote, due to the fact that we cannot increase the number of replications infinitely and keep reducing the latency still appears.

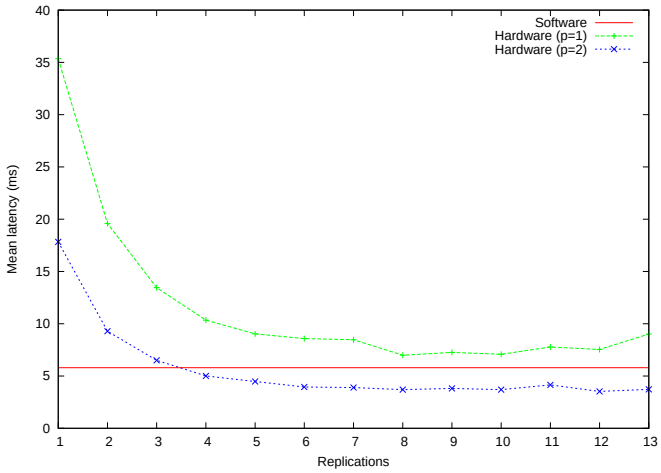


Figure 5.28: Prime tester: software versus hardware version

5.6.2 Hardware

Scaling

For speed comparison of our module with existing implementation, we need to scale our results to the FPGAs used. We use the results of table 5.2 which gives the critical path and maximum frequency after synthesis of our 512 bit exponentiator on different FPGAs.

	Critical path	Maximum frequency
VirtexE XCV600E	27.323ns	36.599MHz
Spartan3 XC3S2000	22.397ns	44.649MHz
Virtex5 XCV5VLX507	6.835ns	146.304MHz

Table 5.2: Scaling data

Exponentiation speed

Table 5.3 compares our exponentiator speed with the implementations presented in section 2.3.3. We use a pipeline depth of 4 for the adders of the Montgomery multiplier.

This table shows that our exponentiator design is faster than the one-CSA design of [1]. Without scaling the results, our exponentiator is also faster than any other design presented in section 2.3.3. Even if we could not scale Fry's design, the speedup comparison is quite fair as the ARSA core was synthesized on a modern Altera FPGA. On the contrary, we cannot compare fairly our design with Blum's design which was synthesized in 1999 on a FPGA that is no longer supported by Xilinx tools. If we had implemented Blum's design on a modern FPGA for comparison, the speedup would have been certainly less than 2.6.

Implementation	FPGA	Clock (MHz)	Time (ms)	Speedup	Scaled Time (ms)	Scaled Speedup
Our design ($p = 1$, $r = 1$)	XC5VLX50T	147.3	3.6	1	-	-
Amanor [1]	XVC2000E-6	43.3	16.1	4.5	14.4	1.1
Blum (fastest) [7]	XC4000	56.5	9.4	2.6	N/A [a]	N/A [a]
Fry (ARSA 128) [8]	Stratix/Cyclone	200	80.0	22.3	N/A [b]	N/A [b]

[a] The scaling could not be done because the XC4000 is no longer supported by Xilinx synthesis tools
[b] Not enough information is given in [8] on the FPGA used to be able to scale our result

Table 5.3: Comparison of our exponentiator with other implementations for n=512 bits

Prime tester

Table 5.4 compares our prime tester implementation (the adders pipeline depth is still set to 4) with the prime tester of [11] which also uses pipelining and implements the deterministic variant of the Rabin-Miller test.

Implementation	FPGA	Logic Area (Slices)	Clock (MHz)	Time (ms)	Scaled Time (ms)
Ours ($p = 1$, $r = 1$)	XC5VLX50T	9 840	122.5	16.8	-
Ours ($p = 1$, $r = 2$)	XC5VLX50T	10 744	113.8	9.3	-
Ours ($p = 1$, $r = 4$)	XC5VLX50T	13 638	113.8	7.6	-
Ours ($p = 2$, $r = 1$)	XC5VLX50T	9 254	123.5	8.5	-
Ours ($p = 2$, $r = 2$)	XC5VLX50T	11 907	123.5	6.8	-
Ours ($p = 2$, $r = 4$)	XC5VLX50T	19 731	123.5	3.3	-
Cheung [11] (non-scalable design)	XC3S2000	10 153	10.5	25.3	7.7
Cheung [11] (scalable design 8 PE)	XC3S2000	2 736	25.2	937.8	285.9
Cheung [11] (scalable design 32 PE)	XC3S2000	9 146	26.7	378.1	115.3

Table 5.4: Comparison of our prime tester with the implementations done in [11] for n=512 bits

Our design is faster than Cheung's non scalable design for $p = 2$ and $r \geqslant 2$. For $r = 2$, it is 10% faster. However it takes a bit more area. For $r = 4$, by less than doubling the area, we divide the time to perform a test by more than 2 and get a speedup of 2.3 over Cheung's non scalable design.

If we compare our design with Cheung's scalable prime tester, for $p = 2$ and $r = 1$, we get a speedup of 13.5 over the 32 PE version, both design taking the same area.

This table also highlights a shortcoming of our design which cannot fit in less than 9000 slices for a 512-bit prime test. Cheung's scalable design can fit in 2000 slices. This is due to the fact that Cheung's multiplier enables the total size of the processing elements to be less than the number of iteration to be performed. In that case the last processing element transfers its result to the first to continue the computation of the multiplication.

In a context where the area constraint is much stronger than the speed contraint, Cheung's scalable design can turn out to be better than ours. On the contrary, if we want to perform a fast prime test with a moderate area constraint, our design is better.

5.7 Summary

In this last chapter, our hardware designs and implementations have been evaluated in depth. The design space explored by our design together with the speed improvements of the multiplier with the number of pipeline stages and replications have been put forward.

We have confirmed experimentally the analyses done in section 3.4.3, 3.4.4 about:

- the optimal number of multiplier pipeline stages when using it with the exponentiator

- the benefits of using pipelined adders/subtractors on the critical path

Our model of the multiplier has been evaluated against experimental results and proves to be useful to give insights about the variations of area and throughput with the number of pipeline stages and the number of replications.

Finally our multiplier, exponentiator and prime tester have been compared to existing software and hardware implementations. They all turn out to be faster than the equivalent software implementations on a modern CPU. Our Montgomery multiplier implemented on a 150 MHz XC5VLX50T FPGA is indeed more than 16 times faster than the optimised software implementation on a 2.8 GHz Core 2 Duo E7400. The synthesis results of our exponentiator give relevant speedups of up to 22.3 against existing hardware implementations. With moderate area constraints, our Rabin-Miller prime tester is also faster than previous hardware designs [11] of the same algorithm with a speedup of 13.6 times with less than 10% area overhead.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we have presented a new implementation of three modules which are the core of many encryption algorithms: the Montgomery modular multiplier, the modular exponentiator and the prime tester. Our multiplier is the first Montgomery multiplier design with variable pipeline stages and variable parallelism by replication. A flexible pipeline has been designed for the multiplier. This pipeline supports any number of pipeline stages and can run at any fill rate. Inside a pipeline block, the carry-save adders have been replicated and put in series. This allowed us to treat consecutive iterations by blocks, avoiding data dependency problems. Both features make this multiplier highly parametric and capable of exploring a large design space, from a slow but area-efficient to a very fast but area-consuming multiplier. Our multiplier can fit in a large family of FPGAs. Its parametric bitwidth, number of pipeline stages and replication also makes it suitable for use on future FPGAs and with future public-key designs where the key size will reach more than 4096 bits. In terms of performance, the implementation of our multiplier on a 150 MHz XC5VLX50T FPGA is more than 16 times faster than the optimised software implementation on a 2.8 GHz Core 2 Duo E7400 CPU. This speedup can still be increase by increasing the number of pipeline stages. The only limitation is the area available on the FPGA used. However, the multiplier cannot be replicated infinitely without any negative effects on the critical path. This result shows that pipelining is more scalable that replication.

We have modelled the effects of pipelining and replication on the area, maximum frequency and throughput of a particular design. This model particularly fits our multiplier but can be easily extended to any design which uses our methods for pipelining and replication. It gives valuable insights about the variations of logic area, register area and throughput with the number of pipeline stages and replications. The model has been integrated in a software, allowing rapid parameterisation of our multiplier.

We have designed a modular exponentiatior integrating our mutiplier. Modular exponentiation is a very important operation in cryptography. It is used in RSA encryption/decryption, in the Diffie-Hellman key exchange protocol and most prime testers. During this integration, pipelined adders and subtractos were added to the multiplier. The use of pipelined adders and subtractors reduces the critical path and enables the exponentiator to run faster with a negligible area overhead. By taking full advantage of the pipelining and replication features of the multiplier, our modular exponentiator is faster than a very optimised software version running on a 2.8 GHz Core 2 Duo E7400 CPU by a factor of 1.5. This speedup is limited by two factors. First, increasing the number of replications increases the critical path which can create a decrease in the speed of the design for a high number of replications. Second, the data dependencies in the exponentiation algorithm used limit the maximum number of multiplier's pipeline stages to 2 in order to keep the pipeline full. Speedups from 1.1 up to 22.3 over existing hardware implementations have also been put forward. These last results make our exponentiator particularly well-suited for high throughput

cryptosystem designs as well as any other design requiring fast hardware modular exponentiation.

Using our parametric version of modular multiplication and exponentiation, a hardware version of the Rabin-Miller strong pseudoprime test has been implemented. The pipeline depth of the multiplier and its number of replications, together with the number of first primes used for the test are parametric. This module covers a large design space and is faster than a very optimised software implementation on a 2.8 GHz Core 2 Duo E7400 CPU by a factor of 1.6. This speedup is limited by the same two factors as the exponentiator. Our prime tester is also faster than existing hardware implementations with speedups from 1.1 times up to 13.6 times with only a 10% area overhead. The parametric nature of our prime tester makes it suitable for various applications from RSA key generation to the test of large primes such as Mersenne prime numbers [22].

6.2 Future Work

Future extensions to the work done in this thesis are planned. First, we could try to make our multiplier still more parametric by trying to decompose it as a systolic array as done in [7]. Trying to combine pipelining, replication and a systolic array for each pipeline block could enlarge the design space for our multiplier. We could for instance represent each pipeline block as a systolic array and replicate inside a cell of the systolic array. It would still increase the granularity of our design.

To improve the speed of our design, we could try to recursively apply pipelining and replication to the exponentiator and prime tester. In fact, as using more than 2 pipeline stages for the multiplier when used inside the exponentiator is useless, replicating and pipelining the exponentiator itself could solve this speed limitation. Of course, the number of pipeline stages of the exponentiator would now be limited by the data dependencies inside the prime tester, which itself could be pipelined and replicated. It would be interesting to extend our model to take into account these different levels of pipelining and replication and to develop a software allowing the optimisation and configuration of our three modules. To extend the design space towards very low area, a method reusing the same pipeline blocks alternately, the last block looping back to the first one, could also be tried as in [11]. Another interesting idea would be to enable the user to implement any storage element either as simple registers or as RAMs to better exploit device specific resources. New FPGAs are doted with more and more dedicated block RAMs indeed.

With these new features, our modular multiplier, modular exponentiator and Rabin-Miller prime tester could be put together to produce a highly parametric RSA coprocessor. Using the right parameters, such a coprocessor could meet any speed/area requirement and fit in almost any FPGA.

Our pipelining and replication methods together with the corresponding model are generic. Eventually, we could automate this technique and apply it to any design such as DES encryption, FIR filter, etc. The idea would be to create a system taking as inputs the design's basic cell as well as information about the part of the cell which can be replicated. This information could for example be given directly in the Verilog file as special comments. The system could then automatically generate Verilog files corresponding to a design pipelined and replicated any number of times. Ultimately, with some information about the FPGA targeted, the design with the optimal number of pipeline stages and replications could be generated and synthesized automatically from a basic cell.

A power consumption model could also be integrated in order to take into account the three main constraints of any hardware design: speed, area and power consumption. The idea would be to give insights on the effects of pipelining and replication on the power consumption of a design. As is done for finding the trends of the maximum frequency, we would certainly have do run some

pre-syntheses and even place-and-route operations for some values of r and p and then interpolate. Another solution would be to interface with the tools proposed by the vendors as Xilinx XPower Estimator. These tools can give useful power consumption information early in the design process. A combination of both approaches could be considered. Still in terms of power analysis, we could also try to figure out if the glitch effects [23] enable our pipelined design to consume less than a non-pipelined version [24].

A last possible and challenging extension of our work would be to use the runtime reconfiguration capabilities of FPGAs. Instead of fixing the parameters of our modules at synthesis time, we could make them reconfigurable at runtime. For example, the bitwidth of our prime tester could be adapted to the bitwidth of the numbers to test directly at runtime. The number of pipeline stages and replications of our multiplier could also be modified at runtime to get the optimum throughput depending on how it is used: in standalone mode, by the exponentiator or by the prime tester.

Bibliography

[1] D. Narh Amanor, C. Paar, J. Pelzl, V. Bunimov, and M. Schimmler. Efficient hardware architectures for modular multiplication on FPGAs. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 539–542, Aug. 2005.

[2] Wolfram webpage on Rabin-Miller strong pseudoprime test. http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html.

[3] RSA Laboratories article on RSA security. http://www.rsa.com/rsalabs/node.asp?id=2004.

[4] Nadia Nedjah and Luiza de Macedo Mourelle. A review of modular multiplication methods and respective hardware implementation. *Informatica (Slovenia)*, 30(1):111–129, 2006.

[5] Gaël Hachez and Jean-Jacques Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 293–301, London, UK, 2000. Springer-Verlag.

[6] Bunimov. V, Schimmler. M, and Tolg. B. A complexity-effective version of montgomery's algorithm. In *Workshop on Complexity Effective Designs (WECD02)*, May 2002.

[7] Thomas Blum. Montgomery modular exponentiation on reconfigurable hardware. *Computer Arithmetic, IEEE Symposium on*, 0:70, 1999.

[8] John Fry and Martin Langhammer. Rsa & public key cryptography in FPGAs. *CDC*, 2003. Altera.

[9] Young Sae Kim, Woo Seok Kang, and Jun Rim Choi. Asynchronous implementation of 1024-bit modular processor for rsa cryptosystem. In *ASICs, 2000. AP-ASIC 2000. Proceedings of the Second IEEE Asia Pacific Conference on*, pages 187–190, 2000.

[10] Jing Lu and Wan Qian. Implementing a 1024-bit RSA on FPGA. 2003. CS502 Final Project, Washington University in St. Louis.

[11] R.C.C. Cheung, A. Brown, W. Luk, and P.Y.K. Cheung. A scalable hardware architecture for prime number validation. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pages 177–184, Dec. 2004.

[12] H. Wiklicky, N. Dulay, and M. Huth. Public key cryptography & digital signature. Network Security Lecture Notes.

[13] Ersin Oksuzoglu and Erkay Savas. Parametric, secure and compact implementation of rsa on fpga. *Reconfigurable Computing and FPGAs, International Conference on*, 0:391–396, 2008.

[14] M. Flynn. Area - time - power and design effort: the basic tradeoffs in application specific systems. In *ASAP '05: Proceedings of the 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Wolfram webpage on Lucas test.
http://mathworld.wolfram.com/LucasPseudoprime.html.

[16] Laszlo Hars. Modular inverse algorithms without multiplications for cryptographic applications. *EURASIP J. Embedded Syst.*, 2006(1):2–2, 2006.

[17] Icarus verilog website.
http://www.icarus.com/eda/verilog/.

[18] Myhdl documentation.
http://www.myhdl.org/doc/0.6/manual/index.html.

[19] Verilator manual.
http://www.veripool.org/projects/verilator/wiki/Manual-verilator.

[20] Tobias Becker, Wayne Luk, and Peter Y. Cheung. Parametric design for reconfigurable software-defined radio. In *ARC '09: Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, pages 15–26, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] Gmp library manual.
http://gmplib.org/manual/.

[22] Great Internet Mersenne Prime Search.
http://www.mersenne.org/.

[23] Altaf Abdul Gaffar, Jonathan A. Clarke, and George A. Constantinides. Modeling of glitch effects in fpga based arithmetic circuits. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 349–352, Dec. 2006.

[24] Steven J. E. Wilton, Su-Shin Ang, and Wayne Luk. The impact of pipelining on energy per operation in field-programmable gate arrays. In *FPL*, pages 719–728, 2004.

[25] Interactive implementation of the Rabin-Miller primality test.
http://herbert.gandraxa.com/herbert/mrp.asp.

Appendix A

Specifications

A.1 Montgomery multiplier

A.1.1 Function

The Montgomery multiplier performs modular multiplication with odd modulus for any bitwidth of the inputs. It supports any number of pipeline stages and replications of the main processing element, the only condition being:

$$p.r \leqslant n$$

with n the bitwidth, p the pipeline depth and r the number of replications. Its adders and subtractors can also be pipelined.

The Montgomery multiplier module is implemented in the `mmult_full.v` Verilog file.

A.1.2 Inputs/Outputs

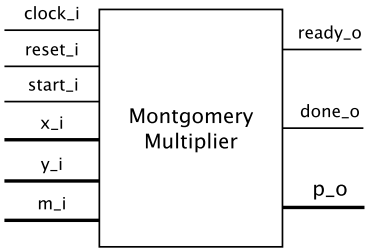


Figure A.1: Montgomery Multiplier interface

The inputs/outputs of the Montgomery multiplier are shown in figure A.1. They are:

- `clock_i`: input for the clock.
- `reset_i`: when this signal is asserted, the multiplier is reset.
- `start_i`: when this signal is asserted, the computation starts with the given inputs for the operands and the modulus.
- `x_i`: first operand of the multiplication.
- `y_i`: second operand of the multiplication.
- `m_i`: modulus.
- `ready_o`: this signal is asserted when the multiplier is ready to receive new inputs.

- `done_o`: this signal is asserted when the current multiplication has finished.
- `p_o`: result of the multiplication.

A.1.3 Parameters

The module has the following parameters:

- `WIDTH`: bitwidth of the multiplier.
- `NB_BLOCKS`: number of pipeline stages.
- `WIDTH_ITER`: bitwidth of the number of iterations performed by a cell: $log_2\left(\frac{WIDTH}{NB\_BLOCKS}\right) + 1$.
- `FINAL_SUB`: determine if the final subtraction is done (in this case this parameter has to be set to 1) or not (set to 0).
- `ADD_SUB_STAGES`: depth of the adders/subtractors pipeline.
- `WIDTH_STAGES`: bitwidth of the number of adders/subtractors pipeline stages: $log_2\left(ADD\_SUB\_STAGES\right) + 1$.
- `NB_REP`: number of replications.

A.1.4 Timing diagram

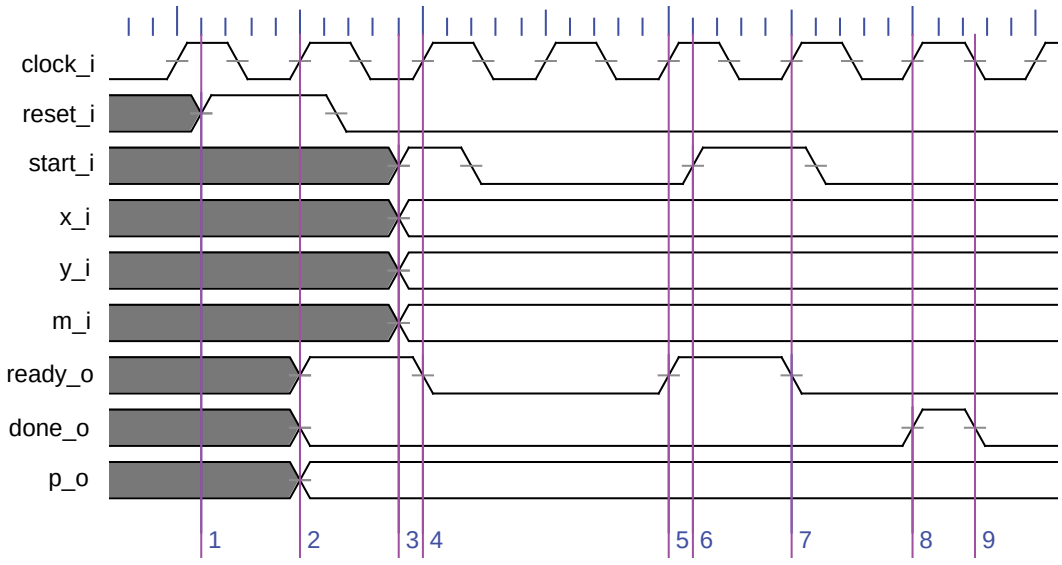


Figure A.2: Multiplier timing diagram

The timing diagram of figure A.2 shows how the multiplier should be used:

- the `reset_i` signal is set (1)
- the `reset_i` signal is taken into account (2): the `ready_o`, `done_o` and `p_o` signals are initialised
- the inputs are given to the multiplier and the `start_i` signal is asserted (3)
- the multiplier begins the computation of the first multiplication (4). The `ready_o` signal is set to 0. The `start_i` signal has to be deasserted before the multiplier becomes ready again.

- When the multiplier is ready to receive new inputs (5), the inputs for the second multiplication can be given and the `start_i` signal asserted (6).
- The `start_i` signal is taken into account (7) and the second multiplication starts. The `start_i` signal has to be deasserted.
- In (8) the first multiplication is finished. The `done_o` signal is asserted during one clock and `p_o` contains the result of the multiplication. This result is valid from (8) to (9).

A.2 Exponentiator

A.2.1 Function

The exponentiator performs modular exponentiation for any bitwidth of the inputs. It supports any number of pipeline stages and replications for its multiplier under the condition:

$$p.r \leqslant n$$

with n the bitwidth, p the pipeline depth and r the number of replications. However, using a pipeline depth greater than 2 is suboptimal as the pipeline of the multiplier cannot be totally filled.

The exponentiator module is implemented in the `mexp.v` Verilog file.

A.2.2 Inputs/Outputs

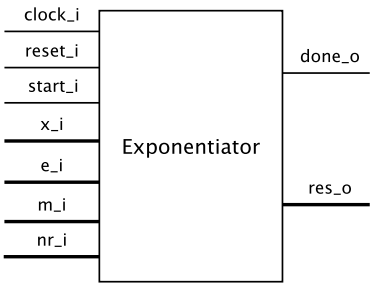


Figure A.3: Exponentiator interface

The inputs/outputs of the exponentiator are shown in figure A.3. They are:

- `clock_i`: input for the clock.
- `reset_i`: when this signal is asserted, the exponentiator is reset.
- `start_i`: when this signal is asserted, the computation starts with the given inputs for the operand, the exponent and the modulus.
- `x_i`: operand of the exponentiation.
- `e_i`: exponent.
- `m_i`: modulus.
- `nr_i`: Montgomery constant (2^{2n} mod m_i, n bitwidth)
- `done_o`: this signal is asserted when the exponentiation is finished.
- `res_o`: result of the exponentiation.

A.2.3 Parameters

The module has the following parameters:

- `WIDTH`: bitwidth of the multiplier.
- `WIDTH_ITER`: bitwidth of the number of iterations performed by the exponentiator: $log_2(WIDTH) + 1$.
- `PIPELINE_STAGES`: number of pipeline stages of the multiplier.
- `LOG_PIPELINE_STAGES`: bitwidth of the number of pipeline stages of the multiplier.
- `FINAL_SUB`: determine if the final subtraction of the multiplier is done (in this case this parameter has to be set to 1) or not (set to 0).
- `ADD_SUB_STAGES`: depth of the multiplier's adders/subtractors pipeline.
- `WIDTH_STAGES`: bitwidth of the number of adders/subtractors pipeline stages: $log_2(ADD\_SUB\_STAGES) + 1$.
- `NB_REP`: number of replications in the multiplier.

A.2.4 Timing diagrams

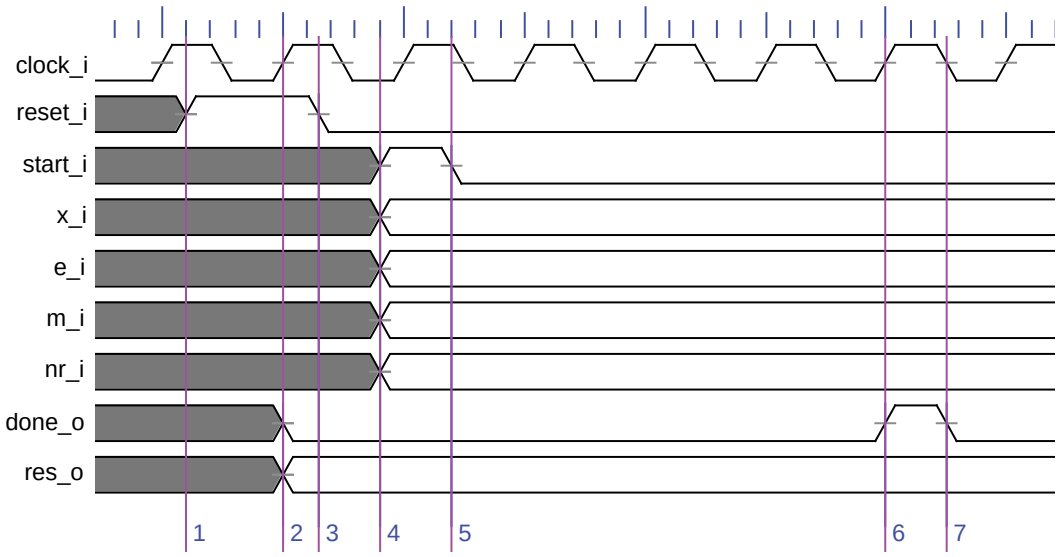


Figure A.4: Exponentiator timing diagram

The timing diagram of figure A.4 shows how the exponentiator should be used:

- the `reset_i` signal is set (1)
- the `reset_i` signal is taken into account (2): the `done_o` and `res_o` signals are initialised
- the `reset_i` signal is deasserted
- the inputs are given to the exponentiator and the `start_i` signal is asserted (4)
- the exponentiator begins its computation. The `start_i` signal is deasserted (5).
- The exponentiation is finished (6). The `done_o` signal is asserted during one clock and `res_o` contains the result of the exponentiation. This result is valid until a new exponentiation is started.

A.3 Prime tester

A.3.1 Function

The prime tester performs a deterministic Rabin-Miller primality test for any bitwidth of the number under test. It supports any number of pipeline stages and replications for the multiplier used under the condition:

$$p.r \leqslant n$$

with n the bitwidth, p the pipeline depth and r the number of replications. However, using a pipeline depth greater than 2 is suboptimal as the pipeline of the multiplier is not totally filled on average.
A ROM stores the values of the first primes used for the test.

The prime tester module is implemented in the `prime_tester.v` Verilog file.

A.3.2 Inputs/Outputs

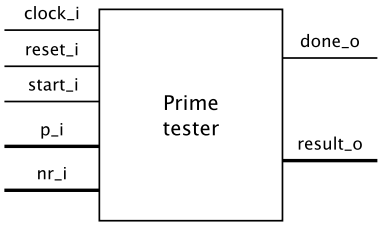


Figure A.5: Prime tester interface

The inputs/outputs of the prime tester are shown in figure A.5. They are:

- `clock_i`: input for the clock.
- `reset_i`: when this signal is asserted, the prime tester is reset.
- `start_i`: when this signal is asserted, the test of the given number starts.
- `p_i`: number under test.
- `nr_i`: Montgomery constant (2^{2n} mod p_i, n bitwidth)
- `done_o`: this signal is asserted when the test is finished.
- `result_o`: result of the test: 0 if the number is composite, 1 if it is probably prime.

A.3.3 Parameters

The parameters of this module are:

- `WIDTH`: bitwidth of the multiplier.
- `WIDTH_ITER`: bitwidth of the number of iterations performed by the exponentiator: $log_2(WIDTH) + 1$.
- `NB_PRIMES`: number of primes used for the test.
- `PRIMES_DATA_WIDTH`: data width of the prime ROM.
- `PRIMES_ADDR_WIDTH`: address width of the prime ROM.

- `PIPELINE_STAGES`: number of pipeline stages of the multiplier.
- `LOG_PIPELINE_STAGES`: bitwidth of the number of pipeline stages of the multiplier.
- `ADD_SUB_STAGES`: depth of the multiplier's adders/subtractors pipeline.
- `WIDTH_STAGES`: bitwidth of the number of adders/subtractors pipeline stages: $log_2(ADD\_SUB\_STAGES) + 1$.
- `NB_REP`: number of replications in the multiplier.

A.3.4 Timing diagrams

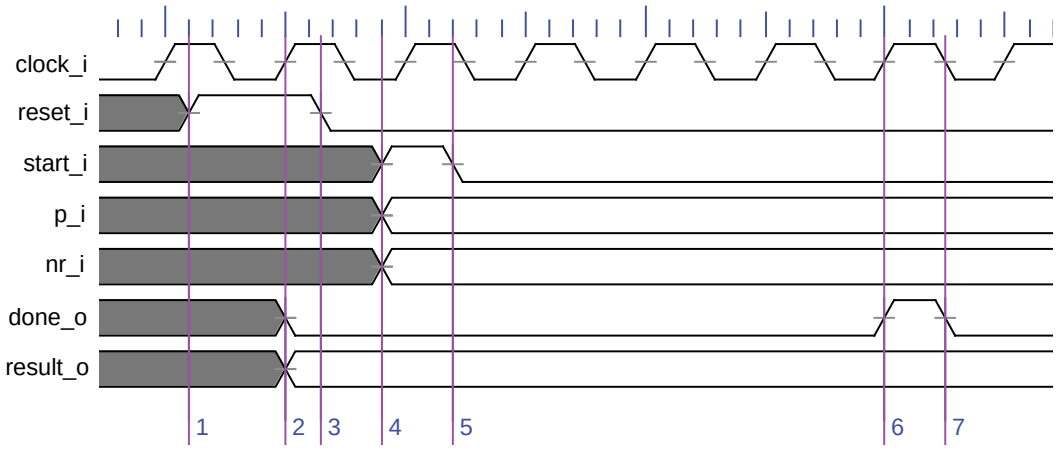


Figure A.6: Prime tester timing diagram

The timing diagram of figure A.6 shows how the prime tester should be used:

- the `reset_i` signal is set (1)
- the `reset_i` signal is taken into account (2): the `done_o` and `result_o` signals are initialised
- the `reset_i` signal is deasserted (3)
- the number under test is given to the prime tester and the `start_i` signal is asserted (4)
- the prime tester begins the test. The `start_i` signal is deasserted (5).
- The prime test is finished (6). The `done_o` signal is asserted during one clock and `result_o` contains the result of the test: 0 if the number is composite, 1 if it is probably prime. This result is valid until a new prime test is started.

Appendix B

Software manual

B.1 Features

The "Multiplier parameters finder" has two main features:

1. Find the optimal number of replications and pipeline stages of the multiplier according to our model.
2. Run syntheses around the values of the r and p given by our model in order to possibly find a better optimum.

B.2 User interface

Figure B.1 represents the different components of the user interface:

- Zone (1) gathers the different parameters of the model used for the optimisation.
- The user clicks on the button "Optimize" (2) after having completed all the fields of zone (1) in order to launch the optimisation.
- The results of the optimisation are given in zone (3).
- After having obtained these results, the user can perform real syntheses to confirm them. The range of p and r in zone (4) are automatically filled according to the results given by the model. However, the user can change them as desired. Note that syntheses can also be run independently of the model's results. In that case, only the `iterations` field of zone (1) matters. It corresponds to the bitwidth of the multiplier .
- Zone (5) enables the user to choose the folder where the Verilog files of the multipliers are located and the synthesis tool used. The software only supports `xst` from the ISE synthesis toolsuite.
- Clicking on the button "Run" (6) launches the syntheses. Information about the process are given in the area (8).
- Once the syntheses are complete, the optimal values of r and p in the synthesis range are printed in zone (7) together with the corresponding values of the logic/register area, maximum frequency and throughput.

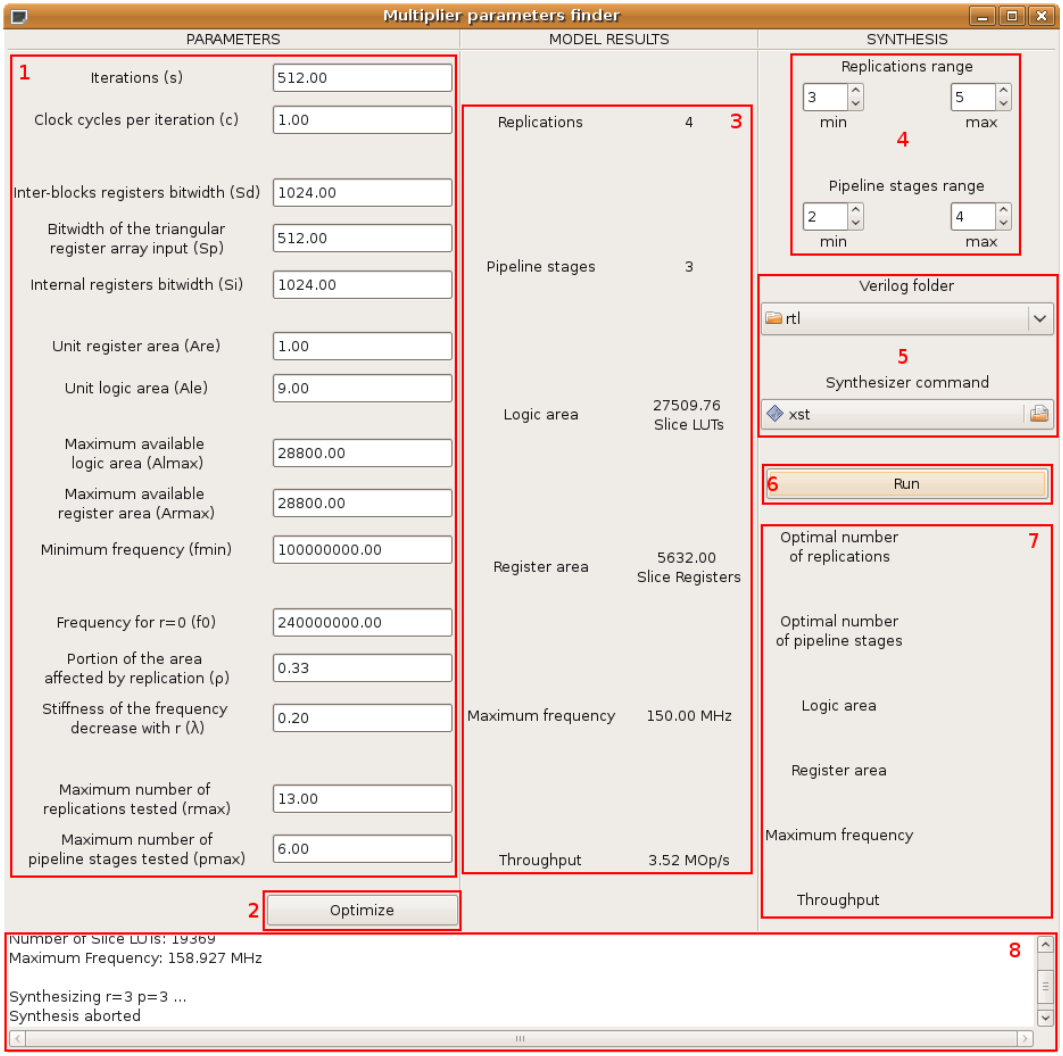


Figure B.1: Components of the user interface