

Studienarbeit

# Hardware-Software Co-Design: A Case Study on an accelerated Implementation of RSA

Benedikt Gierlichs

6th June 2005

Supervisor: Prof. Dr.-Ing. Christof Paar



Ruhr-University Bochum  
Chair for  
Communication Security  
Horst Görtz Institute

Supervisor: Dr. Maire McLoone



Queen's University Belfast  
Institute for Electronics,  
Communication and  
Information Technology

## Abstract

This work presents a case study of accelerating embedded processor systems by use of additional, reconfigurable hardware. The well known RSA encryption scheme will be used to demonstrate how to move computationally intensive operators into hardware to gain a significant speed-up.

## Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Cryptography . . . . .	7
1.1.1	Symmetric encryption . . . . .	7
1.1.2	Public key encryption . . . . .	8
<b>2</b>	<b>RSA</b>	<b>8</b>
2.1	Math background . . . . .	9
2.2	Encryption / Decryption algorithm . . . . .	10
2.2.1	Analysis and comments . . . . .	11
<b>3</b>	<b>RSA in Software</b>	<b>12</b>
3.1	Advantages and disadvantages . . . . .	12
3.2	RSA acceleration . . . . .	13
<b>4</b>	<b>Hardware</b>	<b>15</b>
4.1	Choice of hardware module . . . . .	15
4.2	The Montgomery Multiplication Algorithm . . . . .	16
4.2.1	Montgomery Reduction Technique . . . . .	16
4.2.2	Montgomery Multiplication Algorithm . . . . .	17
4.2.3	Montgomery Exponentiation Algorithm . . . . .	17
4.2.4	VHDL implementation of a Montgomery Multiplier . . . . .	19
<b>5</b>	<b>Development Board</b>	<b>23</b>
5.1	Xilinx VirtexII Pro Chip . . . . .	24
5.2	Tools involved: XPS . . . . .	25
<b>6</b>	<b>Wiring FPGA to PowerPC</b>	<b>26</b>
6.1	Starting from “Hello World” . . . . .	26
6.2	The OCM-Controller . . . . .	29
6.3	Block RAM (BRAM) Block . . . . .	30
6.4	Faults in XPS 6.2i . . . . .	32
6.5	Interfacing BlockRAM . . . . .	33
6.6	Further aspects . . . . .	35

---

6.7	The final hardware design . . . . .	38
6.8	VHDL code . . . . .	38
<b>7</b>	<b>Processor Application</b>	<b>39</b>
<b>8</b>	<b>Results</b>	<b>39</b>
<b>A</b>	<b>VHDL code - Finite State Machine</b>	<b>42</b>
<b>B</b>	<b>VHDL code - Montgomery Multiplier</b>	<b>49</b>
B.1	main_mult.vhd . . . . .	49
B.2	main_mult_counter.vhd . . . . .	53
B.3	mult_64Bits.vhd . . . . .	54
B.4	mult_32Bits.vhd . . . . .	56
B.5	mult_16bits.vhd . . . . .	59
<b>C</b>	<b>C source code</b>	<b>60</b>
<b>D</b>	<b>Test vectors</b>	<b>65</b>

## List of Figures

1	Cascading 16x16-bit multipliers . . . . .	20
2	AVNET Xilinx VirtexII Pro Development Board . . . . .	24
3	Xilinx VirtexII Pro architecture [3] . . . . .	24
4	Screenshot - system layout of “Hello World” example by AVNET	27
5	Using DSOCM_V10 and EDK Infrastructure Lib in a PowerPC System. [19] . . . . .	29
6	DSOCM Controller Block Diagramm [19] . . . . .	31
7	Block Ram Block Block Diagramm [15] . . . . .	32
8	DSOCM to Block Ram Block Diagramm [18] . . . . .	33
9	Data Load Timing in single cycle mode [18] . . . . .	36
10	Data Store Timing in single cycle mode [18] . . . . .	37
11	Deterministic Mealy state machine . . . . .	38
12	Model of functional blocks’ co-action . . . . .	39
13	Screenshot - system hardware layout . . . . .	40

**List of Tables**

1	RSA public key encryption - key generation algorithm . . . .	11
2	RSA public key encryption algorithm . . . . .	11
3	RSA public key decryption algorithm . . . . .	11
4	Chinese Remainder Theorem . . . . .	14
5	Montgomery Reduction Technique . . . . .	17
6	Montgomery Multiplication . . . . .	17
7	Montgomery Exponentiation . . . . .	18
8	Montgomery Exponentiation Algorithm . . . . .	18
9	VHDL code example I . . . . .	35
10	VHDL code example II . . . . .	35
11	VHDL code example III . . . . .	36

## 1 Introduction

The security of certain information has been a matter ever since. Means of communication have changed a lot in the last 30 years and the amount of information that is being communicated, especially by electronic means, is growing exponentially. To be able to deal with the vast amounts of data being processed in a secured electronic information system, the system itself has to be a high-performance system.

Nearly all cryptographic algorithms, being the core component of most security systems, are based upon the fact that their complexity (= level of security provided) is superior to present computing power. As - according to Moore's Law [7] - computing power doubles every 1.5 years, the complexity of cryptographic computations needs to grow at least at the same rate to provide a consistent level of security. But this does also mean, that the actual workload of data processing, meaning encryption and decryption, increases. As a consequence, the demanded amount of computing power of a secured information system increases at the same speed as cryptographic complexity and integration level of its hardware components.

One way to deal with these increasing standards is to raise the system's computing power, eg by simply using a CPU with greater clock speed. Another approach to meet the challenge of providing sufficient computing power is to add highly specialised hardware accelerators, satisfying the requirements of cryptographic components involved, to the system.

The first approach seems to be very easy and feasible in general. This is true as long as we are just thinking of banal computer systems. The second approach seems to be far more complex but, as will be explained in detail later, is very significant if we think of systems with limited resources of computing power and/or space, eg PDAs, mobile phones, or onboard units. The main part of this report will explain the procedure of accelerating an embedded cryptographic system by specially designed hardware from the evaluation of the involved algorithms down to designing the accelerator using VHDL <sup>1</sup>.

---

<sup>1</sup>Very large scale integration Hardware Description Language

## 1.1 Cryptography

The word cryptography comes from the Greek *kryptos*, meaning hidden, and *graphia*, meaning writing. Cryptography, thus, literally means the art of secret writing. The art of hiding information therefore is not as modern as one might guess but is known to be some thousand years old.

Cryptography provides, amongst others, means of hiding and recovering information called encryption schemes. In general an encryption scheme consists of a set of encryption and decryption operations each associated with a key, which is supposed to be kept secret.

Upon the relation between the two keys, an encryption scheme can be related to either main category of encryption: symmetric or public key encryption (cf [1]).

### 1.1.1 Symmetric encryption

An encryption scheme is related to symmetric cryptography, when it is computationally easy to discover the second key, knowing one of them. In most practical cases the two keys will be identical, which is illustrated by the word symmetric, the shared key is referred to as *secret key* (cf [1]).

One advantage of schemes related to this category is their performance in terms of throughput, as these schemes mostly use boolean operations, permutations and shift operations on bit or byte level. A disadvantage is, that all parties involved in the communication process have to share a common secret, the *secret key*. This implicates more difficulties, than might be obvious at first sight. First of all it means, that one can only communicate securely with another party, if the two have agreed or shared a symmetric key before. The actual act of sharing or agreeing on a key might be difficult, if you consider both parties on different continents for example, having in mind, that the key must not be disclosed to others during transfer.

A common image to explain the idea of symmetric encryption is a safe. All participating parties own an identic copy of the key to the safe, so every party can open the safe to either put something inside (encryption), or to

get something out (decryption).

### 1.1.2 Public key encryption

An encryption scheme is said to be public key encryption, when it is impossible to compute the second key, knowing one of them. In this context the encryption operation, using the encryption key, can be regarded as a trapdoor one way function, with the decryption key being the trapdoor, that allows easy message recovery. Message recovery without knowledge of the decryption key is computationally infeasible (cf [1]).

A major advantage of a scheme belonging to this category is the fact, that one cannot compute the decryption key knowing only the encryption key. This allows distribution of a party's encryption key over insecure channels, which simplifies the process of key distribution. Therefore the encryption key is referred to as *public key* while the decryption key is called *private key*. One of the disadvantages of public key encryption is its bad performance in terms of throughput. In order to keep the decryption key secure, even though the encryption key is available in public, the encryption scheme needs to be more complex than a symmetric one. This denotes that the operations being performed become more complex and time consuming.

To get an idea of Public Key Encryption, one can imagine a simple mailbox. Anyone can put a letter into the mailbox (public encryption key), but only the owner of the mailbox's key can get the letters out of it (private decryption key).

## 2 RSA

The RSA cryptosystem is by far the most used public key encryption system. Its name is an abbreviation of the names of R. Rivest, A. Shamir and L. Adleman, who published it in 1978 [8]. This section provides a short introduction to the underlying mathematical principles, a detailed look at RSA encryption and decryption operations, followed by a short analysis and some



comments.

## 2.1 Math background

The security of public key encryption schemes relies on number theoretic problems which are assumed to be hard, eg factoring and computing discrete logarithms in finite fields. This introduction to number theory and modular arithmetic or finite fields follows [1], but does not provide a complete overview. The author refers to [1] for further reading.

An integer  $p$  larger than 1 is called a *prime number* if its only divisors are 1 and  $p$ , eg  $p = 2, 3, 5, 7, 11, 13, \dots$ . There exists an infinite set of prime numbers and there are several well known algorithms of generating prime numbers.

Two integers  $a$  and  $b$  are called *relatively prime*, if their greatest common divisor is 1, eg 3 and 4 are relatively prime. Prime numbers play an important role in public key encryption as will be seen later on.

Although most people would say they do not know modular arithmetic or modular reduction they use it in everyday life. Modular reduction means that the set of integer numbers available is limited, the limit is set by the so called modulus, denoted by  $n$ . Modular arithmetic with the modulus being 5 means, that the set of available numbers consists of  $\{0, 1, 2, 3, 4\}$ . Any number bigger than or equal to the modulus has to be reduced by the modulus by subtraction until it equals a number within the set of available numbers, this operation is called *modular reduction*.

*Modular addition* is defined by an ordinary addition followed by a modular reduction operation in order to keep the result within the set of available numbers. Let  $n = 5$ ,  $a = 3$  and  $b = 2$  then  $a + b \equiv 0 \pmod{n}$  since  $a + b = 5$  and  $5 \equiv 0 \pmod{n}$ . People often use *modular reduction* when talking about time as 21:00 is referred to as 9:00, which is nothing else than  $21 \equiv 9 \pmod{12}$ .

In public key cryptography *modular multiplication*, *exponentiation* and *in-*

*version* are the most important operations.

Modular multiplication works exactly the same way as addition: it is an ordinary multiplication followed by a modular reduction operation.  $a \cdot b \equiv 1 \pmod{5}$  since  $a \cdot b = 6$  and  $6 \equiv 1 \pmod{5}$ . Modular exponentiation works slightly different, it can be computed as a series of multiplications followed by a modular reduction operation.  $a^b \pmod{n} = a \cdot a \cdot a \dots \pmod{n}$ . In practice the modular reduction operation will be performed after each multiplication to keep the intermediate results as small as possible in order to save memory and to avoid unnecessary big inputs to the next multiplication.  $a^b \pmod{n} \equiv (((a \cdot a) \pmod{n}) \cdot a \pmod{n} \dots)$ .

The multiplicative inverse of  $a \pmod{n}$  is a number within the set of available integers satisfying  $a \cdot b \equiv 1 \pmod{n}$ . If  $b$  exists, it is unique and denoted by  $b = a^{-1} \pmod{n}$ .  $b$  exists, if  $a$  and  $n$  are relatively prime. In the example,  $a$  is invertible and the multiplicative inverse of  $a$  modulo  $n$  is  $b$ , as  $3 \cdot 2 = 6 \equiv 1 \pmod{5}$ . The well known Extended Euclidean Algorithm can be used to compute the greatest common divisor of  $a$  and  $n$ . If it is 1, the algorithm computes the multiplicative inverse of  $a$  at the same time. The quantity of numbers within the set of numbers defined by the modulus  $p$  for which a multiplicative inverse exists is denoted by  $\phi(p)$ , which will be the quantity of numbers being relatively prime to  $p$ . If  $p$  is prime,  $\phi(p) = p - 1$ , otherwise Euler's  $\phi$ -function can be used to compute  $\phi(p)$ . The author refers to [1] for further reading on groups, rings, and finite fields.

## 2.2 Encryption / Decryption algorithm

In order to set up an RSA encryption scheme, several numbers have to be either randomly chosen or computed. Every party that wants to participate in RSA secured communication has to set up an own scheme according to the steps shown in table 1.

In order to encrypt a message  $m$  for Alice, Bob should follow the steps shown in table 2.

If Alice wants to read the received message, she should decrypt the ciphertext according to the steps in table 3.

---

Table 1: RSA public key encryption - key generation algorithm

---

- Generate two large random (and distinct) primes  $p$  and  $q$ , each roughly the same size.
  - Compute  $n = pq$  and  $\phi = (p - 1)(q - 1)$ .
  - Select a random integer  $e$ ,  $1 < e < \phi$ , such that  $\gcd(e; \phi) = 1$ .
  - Use the Extended Euclidean Algorithm to compute the unique integer  $d$ ,  $1 < d < \phi$ , such that  $ed \equiv 1 \pmod{\phi}$ .
  - The public key is  $(n, e)$ , the private key is  $d$ . [1]
- 

---

Table 2: RSA public key encryption algorithm

---

- Obtain Alice's authentic public key  $(n, e)$ .
  - Represent the message as an integer  $m$  in the interval  $[0, n - 1]$ .
  - Compute  $c = m^e \pmod{n}$ .
  - Send the ciphertext  $c$  to A. [1]
- 

---

Table 3: RSA public key decryption algorithm

---

- Use the private key  $d$  to recover  $m = c^d \pmod{n}$ .
- 

The RSA encryption scheme works, as  $m = c^d \pmod{n} = m^{e^d} \pmod{n} = m^{ed} \pmod{n} = m^1 \pmod{n}$ . A detailed proof that the scheme works can be found in ([1], p. 286).

### 2.2.1 Analysis and comments

According to ([1], p. 287) the security of RSA relies on the so called RSA problem, that is to recover the message  $m$  of a ciphertext  $c$  knowing only

public information, the modulus  $n$  and the encryption exponent  $e$ . The problem's core turns out to be computing the decryption exponent  $d$  knowing only  $n$  and  $e$ . This problem is well studied and is - assuming the RSA scheme is secure - ought to be computationally equivalent to factoring  $n$ , which is known to be hard, if  $n$  is of sufficient size.

There are two main reasons for RSA's poor performance in terms of throughput. The first one is the fact that large numbers have to be used in order to achieve a satisfying level of security. The other one is the repeated use of the modular reduction operation, which is very costly on general purpose computers. Fortunately there are modular multiplication algorithms that avoid the costly reduction operation. These will be introduced in section 4.

### **3 RSA in Software**

This section deals with a look at RSA's performance when implemented in software only. Advantages, disadvantages and workarounds for the latter ones will be shown.

RSA's performance when implemented in software depends on the processor being used. As mentioned before, the modular reduction operation is very costly and as general purpose computers normally do not have special hardware to meet this task, the performance of RSA in this case mostly depends on the processors performance on the reduction operation.

#### **3.1 Advantages and disadvantages**

Implementation of RSA always involves the need to find a way how to deal with multi precision integers. Recent sizes for RSA parameters are 1024 bits and above. One advantage of a software-only implementation is, that a change of the size of parameters involves only little changes to the code, if the possible need for this change was well considered during implementation phase, so that the size of operands can be changed easily.

Another advantage of a pure software solution is simply the fact, that it is

very easy to change RSA parameters as this only involves editing a few lines of code. The need for this might arise by a disclosed decryption key for example.

When RSA is implemented in software only, there is always the risk of disclosure of valuable information, especially meaning the decryption key, as the software will not be run in specially secured environments in general. Reading the memory of the system the implementation is running on is one approach to gather valuable information. Another approach is to decompile the binary RSA program file and to read the decryption key out of the source code. There are of course means to counteract these approaches, which reach from secure software environments to access controlled and safely designed computer centres.

The main disadvantage of a RSA implementation in software has been mentioned earlier already. The repeated use of the costly modular reduction operation after each multiplication while performing the modular exponentiation, which is the main RSA operation, makes up the bottleneck. Ways to cope with the costly operation will be discussed in the next section.

### **3.2 RSA acceleration**

The problem of the costly modular reduction operation needed in RSA exponentiations has been introduced and discussed in previous sections. This section will introduce three approaches to counteract the deceleration.

The first approach applies the idea of designing special purpose hardware. Trial division of a number by the modulus is one way of performing modular reduction and a piece of hardware especially designed to perform multi precision modular division will perform far better than a general purpose processor. Kaihara and Takagi propose a hardware algorithm capable of performing the modular division operation in  $O(n)$  cycles, where  $n$  is the bitlength of the operands [2].

The second approach is well known as the *Chinese Remainder Theorem* (see table 4).

---

Table 4: Chinese Remainder Theorem

---

If the integers  $n_1, n_2 \dots n_k$  are pairwise relatively prime, then the system of simultaneous congruences

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\dots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

has a unique solution modulo  $n = n_1 * n_2 \dots n_k$  ([1], p. 68).

---

The general idea of applying the Chinese Remainder Theorem to RSA is to solve a system of simultaneous congruences with operands of small size instead of finding the solution of one equation with large operands.

The Chinese Remainder Theorem can be applied to RSA exponentiation, if the numbers  $p$  and  $q$ , used to compute the modulus  $n$ , are known. This will most likely be the case for the party that set up the RSA encryption scheme only, hence in practice the Chinese Remainder Theorem will only be applied to RSA decryption.

Recall that the message  $m$  can be obtained by computing  $m = c^d \pmod{n}$ . As  $n = p \cdot q$ , while  $p$  and  $q$  are prime, the Chinese Remainder Theorem can be applied, leading to the following system of congruences:

$$\begin{aligned} m_p &\equiv c_p^{d_p} \pmod{p}, \text{ where } c_p \equiv c \pmod{p} \text{ and } d_p \equiv d \pmod{p-1} \\ m_q &\equiv c_q^{d_q} \pmod{q}, \text{ where } c_q \equiv c \pmod{q} \text{ and } d_q \equiv d \pmod{q-1} \end{aligned}$$

Finally  $m$  can be computed as  $m \equiv ((m_q - m_p) \cdot (p^{-1} \pmod{q})) \pmod{q}$ .

Due to the fact, that the decryption exponents  $d_p$  and  $d_q$  are half of the size of  $d$ , the decryption exponentiation is accelerated. Furthermore, the solutions to the two congruences can be computed independently and in parallel, which means a further speed-up.

Although the Chinese Remainder Theorem can only be applied to the RSA decryption operation it plays an important role in practical implementations of RSA.

The third approach is of a different kind as it does not aim to speed up the modular reduction operation. Instead it avoids the explicit reduction operation at all. The core technique involved in this approach is called *Montgomery Reduction* [6], which replaces the trial division by the modulus with a series of additions and divisions by a power of 2. Applications of the Montgomery Reduction technique in multiplication and exponentiation algorithms will be introduced in the following sections. For the concern of this report the so called Montgomery Multiplication Algorithm is very important. It combines multi-precision multiplication and Montgomery Reduction to compute the reduction of the product of two integers, as will be shown later on.

In practice, the first approach of the three that were introduced is negligible, as mostly a combination of the latter ones is used as follows. First, the Chinese Remainder Theorem is applied to reduce the size of the operands. Each congruence of the system is then computed using the Montgomery Multiplication Algorithm. Finally, the overall result is computed.

## 4 Hardware

It is common sense that a piece of hardware designed to provide just one special function will perform much better than a general purpose processor. In order to accelerate an implementation of the RSA encryption scheme by special purpose hardware the first step will be the choice of the functional module to be realised in hardware.

### 4.1 Choice of hardware module

As it has been mentioned in previous sections, the main operation within the RSA encryption scheme is the modular exponentiation. The process of choosing the hardware module should therefore start with an analysis of this function. Goal of the analysis is to find the main performance bottleneck in order to specify the hardware module such way that it will result in maximal

gain in performance.

An efficient algorithm performing modular exponentiation is the repeated square-and-multiply algorithm ([1], p. 614). It computes the modular exponentiation in a series of modular multiplications. As previously shown, modular multiplication consists of a multiplication operation followed by a modular reduction operation, which has been proved to be the bottleneck. In the previous section three approaches to counteract the deceleration have been introduced. Combined usage of the Chinese Remainder Theorem and the Montgomery Multiplication Algorithm seems an adequate base to start from.

The Chinese Remainder Theorem has been introduced already and will not be discussed furtheron.

## 4.2 The Montgomery Multiplication Algorithm

The *Montgomery Multiplication Algorithm* makes use of the so called Montgomery Reduction Technique, therefore it will be introduced in the beginning of this section, followed by an explanation of the Multiplication Algorithm itself. Next, its application in the *Montgomery Exponentiation Algorithm* will be discussed. Finally, the actually used VHDL implementation will be looked at.

### 4.2.1 Montgomery Reduction Technique

”Montgomery reduction is a technique which allows efficient implementation of modular multiplication without explicitly carrying out the classical reduction step” ([1], p. 600).

But before the multiplication is being looked at, the reduction will be explained (see table 5).

The benefit of using this reduction technique does not become obvious at this stage, but will, when Montgomery Exponentiation is explained.



Table 5: Montgomery Reduction Technique

---

The Montgomery Reduction of  $T$  modulo  $m$  with respect to  $R$  is denoted as  $T \cdot R^{-1} \pmod{m}$ , where  $m$  is a positive integer,  $R$  is an integer  $> m$  with  $\gcd(R, m) = 1$ , and  $T$  is an integer satisfying  $0 \leq T < m \cdot R$ .

Input :  $T, R, m$

Output:  $T \cdot R^{-1} \pmod{m}$

---

### 4.2.2 Montgomery Multiplication Algorithm

Montgomery Multiplication combines Montgomery Reduction with multi-precision multiplication to compute the Montgomery Reduction of the product of two integers as shown in table 6 (cf [1], p. 602).

Table 6: Montgomery Multiplication

---

Input:  $m = (m_{n-1} \dots m_1 m_0)_b$ ,  $x = (x_{n-1} \dots x_1 x_0)_b$ ,  $y = (y_{n-1} \dots y_1 y_0)_b$  with  $0 \leq x, y < m$ ,  $R = b^n$  with  $\gcd(m, b) = 1$ , and  $m' = -m^{-1} \pmod{b}$ .  
 Output:  $x \cdot y \cdot R^{-1} \pmod{m}$ .

---

Still, the benefit of using the Montgomery technique is innoticeable, but it will become obvious once multiple multiplications are considered, as in the next subsection.

### 4.2.3 Montgomery Exponentiation Algorithm

The Montgomery Exponentiation (see table 7) unites all necessary steps to perform modular exponentiation, eg  $m \equiv c^d \pmod{n}$  in RSA, using the Montgomery Multiplication Algorithm. These are

- mapping the operands into the Montgomery Domain <sup>2</sup>

---

<sup>2</sup>By multiplying with a specially formed number, it is assured that the multiplication's result stays within the Montgomery Domain which means in practice that the factor  $R$  will only appear in its first power.

- performing the exponentiation operation using the repeated square-and-multiply algorithm
- using the Montgomery Multiplication Algorithm to perform the multiplication operation
- mapping the result back into the ordinary integer domain.

---

 Table 7: Montgomery Exponentiation
 

---

Input: modulus  $m$  with bitlength  $l$ ,  $R = 2^l$ ,  $m' = -m^{-1} \pmod{2}$ , exponent  $e$  in binary representation of length  $t$ , base  $x$  satisfying  $1 \leq x < m$ .  
 Note that the base  $x$  always satisfies  $1 \leq x < m$  in a RSA encryption scheme. Output:  $x^e \pmod{m}$

---

Table 8 shows the Montgomery Exponentiation Algorithm in detail.

---

 Table 8: Montgomery Exponentiation Algorithm
 

---

- $\tilde{x} = \text{MontMult}(x, R^2 \pmod{m})$ ,  $A = R \pmod{m}$
  - For  $i$  from  $t$  to 0 do:
    - $A = \text{MontMult}(A, A)$
    - If  $e_i = 1$  then  $A = \text{MontMult}(A, \tilde{x})$
  - $A = \text{MontMult}(A, 1)$
  - return( $A$ )
- 

Now the benefit of using the Montgomery Technique for multiple multiplications becomes obvious. Consider the following computation, first done without mapping of the operands, then with mapping:

$$w \cdot x \cdot y \cdot z \pmod{m}$$

1. compute  $w \cdot x \pmod{m} = w \cdot x \cdot R^{-1} \pmod{m}$

2. compute  $y \cdot z \pmod{m} = y \cdot z \cdot R^{-1} \pmod{m}$
3. compute  $w \cdot x \cdot y \cdot z \pmod{m} = w \cdot x \cdot y \cdot z \cdot R^{-2} \pmod{m}$

As it can be seen the power of the factor  $R$  will decrease by each multiplication performed. If the operands are mapped into Montgomery Domain before multiplication, the product will be an element of the Montgomery Domain as well ( $R$  in its first power), as shown below. Once the multiplication is done, the result can be mapped to ordinary integer domain by a Montgomery Multiplication by 1.

1. mapping operands of first multiplication into Montgomery Domain:  
 $\tilde{w} = w \cdot R \pmod{m}$ ,  $\tilde{y} = x \cdot R \pmod{m}$
2. compute  $\tilde{w} \cdot \tilde{x} \pmod{m} = w \cdot R \cdot x \cdot R \cdot R^{-1} \pmod{m} = w \cdot x \cdot R \pmod{m}$
3. mapping operands of second multiplication into Montgomery domain:  
 $\tilde{y} = y \cdot R \pmod{m}$ ,  $\tilde{z} = z \cdot R \pmod{m}$
4. compute  $\tilde{y} \cdot \tilde{z} \pmod{m} = y \cdot R \cdot z \cdot R \cdot R^{-1} \pmod{m} = y \cdot z \cdot R \pmod{m}$
5. compute  $(\tilde{w} \cdot \tilde{x}) \cdot (\tilde{y} \cdot \tilde{z}) \pmod{m} = (w \cdot x \cdot R) \cdot (y \cdot z \cdot R) \cdot R^{-1} \pmod{m}$   
 $= w \cdot x \cdot y \cdot z \cdot R \pmod{m}$
6. mapping the result back to ordinary integer domain:  
 $w \cdot x \cdot y \cdot z \cdot R \cdot 1 \pmod{m} = w \cdot x \cdot y \cdot z \cdot R \cdot R^{-1} \pmod{m} = w \cdot x \cdot y \cdot z \pmod{m}$

#### 4.2.4 VHDL implementation of a Montgomery Multiplier

The implementation of the Montgomery Multiplier used during this project is part of C. McIvor's Ph.D. thesis [3] which presents implementations for 128-bit and 256-bit multiplication. As these would not fit on the employed FPGA chip and for reasons of simplicity, a modified 64-bit multiplier implementation is set up.

The following paragraph corresponds to ([3], pp. 71-73). In order to perform multiplication of two 64-bit numbers, a 64x64-bit multiplier is created using several 16x16-bit unsigned multipliers. Figure 1 shows how to use 16x16-bit multipliers to develop multipliers for larger bitlengths.

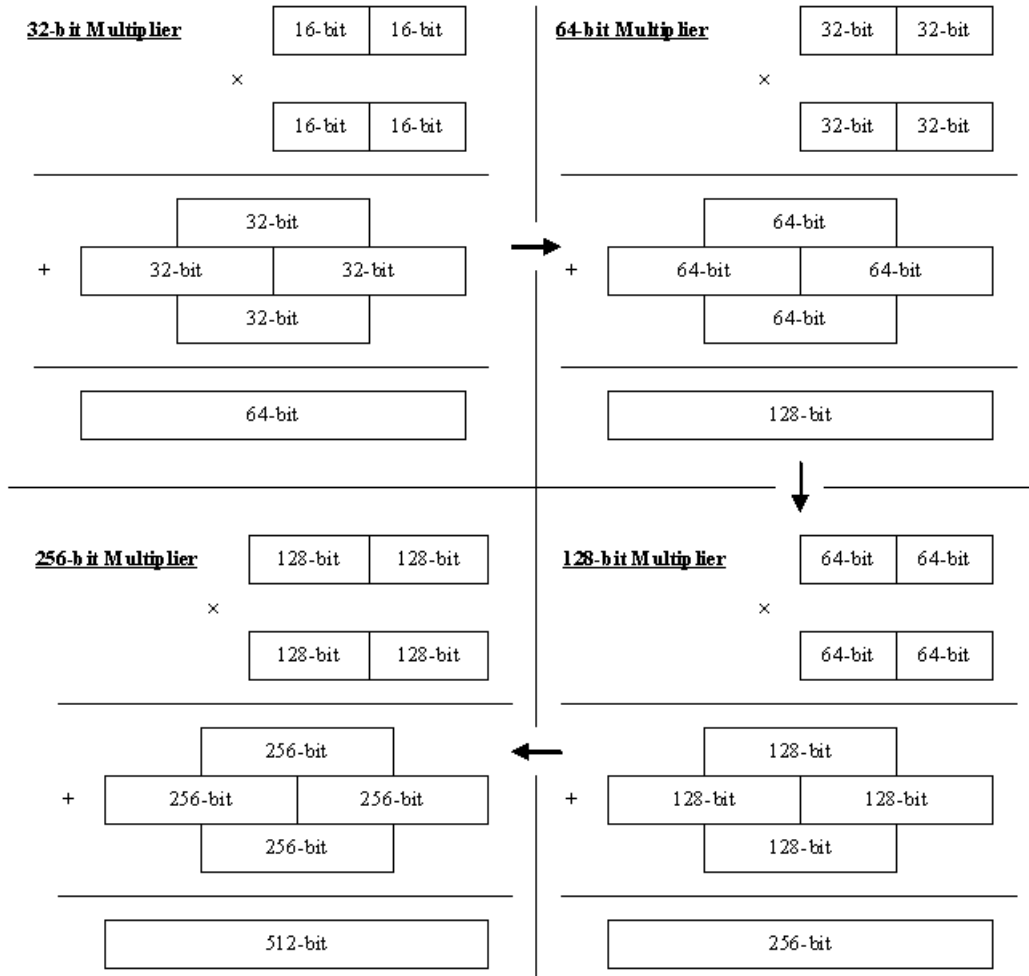


Figure 1: Cascading 16x16-bit multipliers

The larger multipliers are created in a systematic way. The 32x32-bit multiplier, for example, uses 4 16x16-bit multipliers to compute partial products, which are then added to obtain the 64-bit product. This process can be repeated to develop larger multipliers, as shown in figure 1. The multiplier of the desired bitlength can then be used to implement the Montgomery Mul-

tiplier.

The present VHDL implementation of a 64-bit Montgomery Multiplier consists of 5 VHDL-files:

`mult_16bits.vhd`, `mult_32bits.vhd` and `mult_64bits.vhd` are used to create a 64x64-bit multiplier, following the process explained above. `mult_16bits.vhd` implements a 16x16-bit multiplier, `mult_32bits.vhd` instantiates four of these to implement a 32x32bit multiplier and `mult_64bits.vhd` finally instantiates four of them to implement a 64x64-bit multiplier.

`main_mult_counter.vhd` implements a counter that, once started by an external signal, will start at  $'00001'_2$  and end at  $'10100'_2$ , resetting itself to the start value. The actual counter state is used at several locations to coordinate the multiplication process.

`main_mult.vhd` instantiates one counter and one 64-bit multiplier. Along with a clock and a reset signal, which are passed to its components, it reads 5 more input and writes two more output signals. These are as follows:

`load_data` is passed through to the counter, starting it as the signal goes low

`a_operand` one of the factors ( $a$ ) to compute the product (64bits)

`b_operand` one of the factors ( $b$ ) to compute the product (64bits)

`n_modulus` the modulus ( $m$ ) (64bits)

`nprime`  $m' = -m^{-1} \pmod{b}$  as explained in 4.2.2 (64bits)

`mod_ab` the Montgomery Reduction of `a_operand` · `b_operand` (64bits)

`ready` output signal to indicate that the multiplication process has finished.

Once the `load_data` signal is set to low externally, the whole process begins, as the counter is being started. During the first six clock cycles the 64-bit multiplier is used to compute `a_operand` · `b_operand`. The next six clock cycles use the 64-bit multiplier to compute the product of `n_prime` and the 32 least significant bits of the previous computation. In the next period of six cycles the result of the previous computation is multiplied by `n_modulus`. Overall, these three multiplications are done in 18 clock cycles.

During the 19th clock cycle, the 64 most significant bits of the sum of the results of multiplication one and three are stored to be written to the output in the next cycle. If the stored value is smaller than `n_modulus`, it will be written to `mod_ab` immediately, otherwise it will be reduced by `n_modulus` before that. In addition to this, `ready` is set to high in cycle 20, too.

step	clock cycle(s)	computation
I	1 - 6	<code>a_operand · b_operand</code>
II	7 - 12	<code>n_prime · 32 LSB of previous result</code>
III	13 - 18	<code>n_modulus · result of previous result</code>
IV	19	64 MSB of I + III are stored
V	20	<code>mod_ab = result of IV (mod m)</code>

## 5 AVNET Xilinx VirtexII Pro Development Board

The AVNET Xilinx VirtexII Pro Development Board (Figure 2) used in this project has the following features (cf [4], p. 4):

- Xilinx VirtexII Pro Chip FPGA: XC2VP7-FF896 and 1 PowerPC 405 Microprocessor
- High-speed Serial Communication
  - Eight SMA connectors (TX/RX pairs for two Rocket I/Os)
  - Board configurable loop-back for two Rocket I/Os
  - Pads for four additional Rocket I/Os
- Board I/O Connectors
  - Two 140-pin general purpose I/O expansion connectors (AvBus)
  - Up to 30 LVDS pairs
  - 50 Pin 0.1'' Header
- Memory: Micron DDR SDRAM (64MB)
- Communication: RS-232 serial ports
- Power
  - 22.5 Watt AC/DC +5.0V power supply
  - Texas Instruments 3.3V 6A Module
  - National Linear regulators
- Configuration
  - Two Xilinx XC18V04-VQ44 PROMs
  - Parallel IV Cable support for JTAG
  - Fly-wire support for Parallel-III and MultiLINK

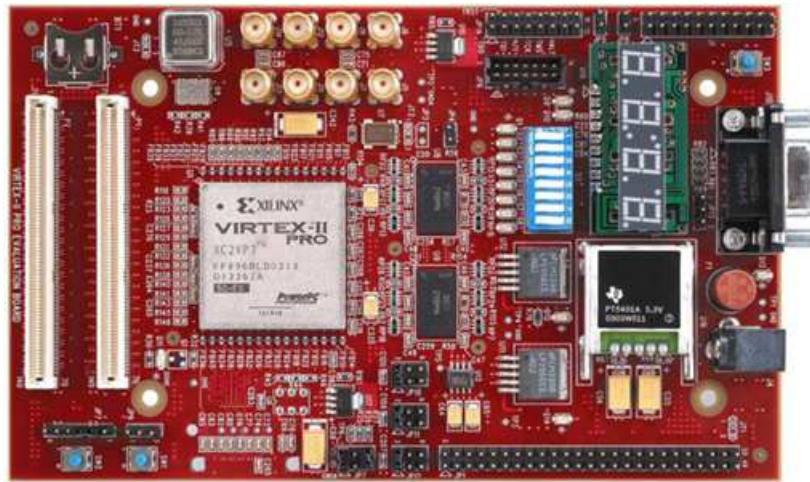


Figure 2: AVNET Xilinx VirtexII Pro Development Board

This project does only make use of the Xilinx VirtexII Pro chip, which will be introduced in the next section. The other components will not be looked at in detail.

## 5.1 Xilinx VirtexII Pro Chip

Figure 3 provides an overview of the VirtexII Pro architecture. The main

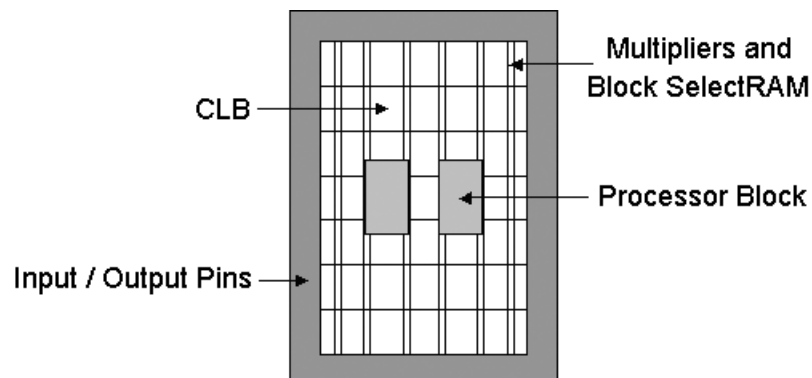


Figure 3: Xilinx VirtexII Pro architecture [3]

processing units are Configurable Logic Blocks (CLBs), which are spread



over the face of the device in an rectangular shape. One CLB consists of four slices, which each contain two Look-Up Tables (LUTs), two flip-flops and fast carry look-ahead logic. Between each two columns of CLBs, there is a column of dedicated 18x18-bit multiplier blocks and Block Select Ram. The outside of the CLB array is surrounded by input/output pins. A Virtex Pro device may contain up to four PowerPC Processors, the one used during this project contains one PowerPC Processor.

## 5.2 Tools involved: XPS

The entire process of wiring dedicated hardware on to the PowerPC Processor and using it as an accelerator was completed using only the Xilinx Platform Studio (XPS, Version 6.2i). It provides all necessary tools for the creation of a basic system, development of user cores in VHDL and development of software in C language as well as all necessary compilers and tools to finally programm the FPGA and transferring the compiled C programm into the memory. The worklow will mostly be like this:

- create a basic system using the base system wizard (Processor, Busses, JTAG)
- import user cores into the system
- connect user core pins to the system
- develop C application to use with processor
- compile hardware code into a bitstream for programming the CLBs
- compile C source code
- update bitstream, so it contains the compiled application code for programming memory units
- download bitstream into the Xilinx chip

## 6 Wiring FPGA to PowerPC: a Roadmap

This section describes how the user core is connected to the PowerPC in order to allow interaction. An important goal is to design this connection as being efficient and easy to use. Different methods of connecting FPGA and PowerPC will be introduced and discussed in regards of this design goal. Next, all necessary steps to use the method meeting this design goal are explained in detail as well as their realization in practice. Finally an overview of the entire system is given and its general function is explained. The VHDL code of the user core can be found in the appendix.

### 6.1 Starting from “Hello World”

The process of creating the connection between the FPGA and the PowerPC starts with a “Hello World” example, provided by AVNET in their example projects shipped together with the board. This example system is ought to be as simple as possible, meaning to use as little components as possible, and consists of the following:

**PowerPC 405** is a RISC microprocessor [18].

**Processor Local Bus** is a high-performance synchronous bus designed for connection of processors to high-performance peripheral devices ([9], [10]).

**On Chip Peripheral Bus** is one element of IBM’s CoreConnect architecture and is a general-purpose synchronous bus designed for easy connection of on-chip peripheral devices [11].

**PLB 2 OPB Bridge** The On-Chip Peripheral Bus (OPB) to Processor Local Bus (PLB) Bridge module translates OPB transactions into PLB transactions. It functions as a slave on the OPB side and as a master on the PLB side [12].

**OPB UARTLite** is a full duplex UART interfacing to the OPB [13].

**PLB BRAM IF Controller** is the interface between the PLB and the bram\_block peripheral [14].

**BRAM Block** is a parameterizable memory module that attaches to a variety of BRAM Interface Controllers [15].

**Proc Sys Rst** allows the customer to tailor the design to suit their application by setting certain parameters to enable/disable features [16].

**JTAGPPC Controller** allows the PowerPC in a Virtex-II Pro to be connected to the JTAG chain of the FPGA [17].

This project uses the PowerPC 405 and the OPB UART Lite components directly. All other components involved are either necessary to connect those two or to get the entire system up und running.

Figure 4 provides an overview of the system that is used. The PowerPC

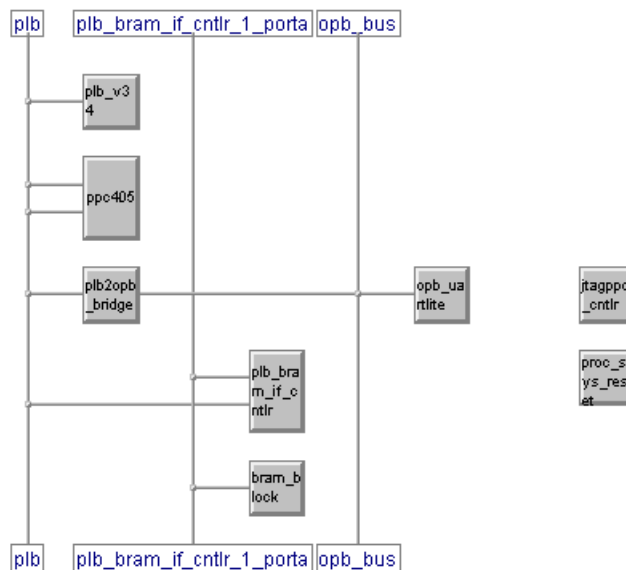


Figure 4: Screenshot - system layout of “Hello World” example by AVNET

405 is used, to send the string ”Hello World” to the UART Lite component, which cannot be connected to the PowerPC 405 straight ahead. Two busses

can be connected to the PPC: the PLB, used in this example, and the On Chip Memory (OCM) Bus. The UART component can only be connected to the OPB, which can be interfaced through the PLB. So the PLB is connected to the PowerPC 405, the PLB2OPB bridge is used to connect PLB and OPB and finally the UART component is connected to the OPB. The UART sends the string “Hello World” to the RS232 socket on the board.

The BRAM Block is used to store the compiled C application so that the PowerPC 405 can read and execute the instructions later on. To connect the BRAM Block to the system, the PLB BRAM IF Controller is needed.

All other components will not be looked at in detail as they have to be part of every system in order to be able to communicate with the FPGA and to reset the system.

In the last paragraph, all three available bus systems of the Virtex device, PLB, OPB, and OCM bus, have been mentioned. This paragraph will provide a short analysis of these and lead to a choice of the bus system, that will be used to connect the user core to the system.

The Processor Local Bus is 64-bit wide, but 32-bit wide slave compatible, and has separate address, write and read data path units that can be arbitrated in only 3 cycles. It can be used by up to 16 masters and 16 slaves and is connected directly to the PowerPC 405 [10]. Therefore it seems to be designed to be used as a quickly accessible and high-bandwidth capable bus. The On-chip Peripheral Bus is 32-bits wide and can be used by up to 16 masters and an unlimited number of clients (up to hardware resources). As it cannot be connected to the PowerPC 405, it needs to be combined with a processor-connected bus. This will mostly be done using the PLB2OPB bridge, as in the “Hello World” example. Using that bridge, the OPB becomes a slave of the PLB and the bridge becomes a master of the OPB. Therefore it seems, that this bus is ought to connect low bandwidth components to the system, that do not need quick processor access.

The On Chip Memory bus consists of the Data Side On Chip Memory Bus (DSOCM) and the Instruction Side On Chip Memory Bus (ISOCM). OCM in general can be used to provide Memory access to the processor. In the present

case it is rather necessary to provide memory to store pure data than processor instructions, that is why only the DSOCM will be looked at in detail. "A typical usage is to connect PowerPC405 and DSBRAM\_IF\_CNTRL modules (both can be found in the EDK Infrastructure Library) to the DSOCM\_V10 Bus" [19]. The OCM can only be used to access On Chip Memory, therefore no arbitration is needed at all. Furthermore, the connected BRAM Block can be accessed by the processor as well as by the FPGA [18]. This seems to meet the requirements very well, so the DSOCM and the anticipated Block RAM will be used for the communication between processor and user core.

## 6.2 Using the OCM-Controller: DSOCM

Figure 5 provides an overview of a usage proposal by Xilinx, that is obeyed in this project. In order to access the Block RAM that is located in the FPGA chip from the processor using the DSOCM bus, three components have to be added to the system: the DSOCM bus, an interface between DSOCM bus and BRAM Block, and the BRAM Block itself.

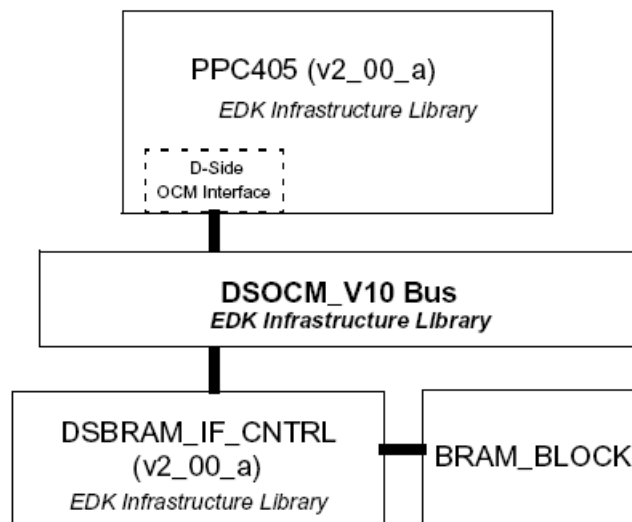


Figure 5: Using DSOCM\_V10 and EDK Infrastructure Lib in a PowerPC System. [19]

The bus itself is fairly simple, as it only multiplexes all present signals. The PowerPC 405 OCM Controller serves as a dedicated interface between the block RAMs in the FPGA and OCM signals available on the embedded PPC405 core. It is capable of addressing up to 16MB of DSBRAM and reads and writes through separate 32-bit read and write busses. A load instruction from the processor will result in the address being passed through to the associated BRAM Block by the controller, on a store instruction, the address and the data to be stored will be passed on. Several controller attributes (observable in Figure 6), can be set in specific registers using system software, details can be found in [18]. Figure 6 shows a block diagram of the DSOCM controller and essential signals:

*Input* signals:

BRAMDSOCMCLK clocks the DCM controller

BRAMDSOCMRDDBUS [0:31] 32-bit read data

*Output* signals:

DSOCMBRAMABUS [8:29] 22-bit read or write address from DSCOM controller to BRAM block; for a write operation, the appropriate write enable signals need to be set (see DSOCMBRAMBYTEWRITE).

DSOCMBRAMWRDBUS [0:31] 32-bit data to be written

DSOCMBRAMBYTEWRITE [0:3] 4 byte write enable signals allowing byte-wide write operations

DSOCMBRAMEN needs to be enabled to read from or write to block RAM

DSOCMBUSY optional signal to provide information of block RAM status to the processor

### 6.3 Block RAM (BRAM) Block

The BRAM block is a customizable memory module that can be attached to a variety of BRAM Block controllers. In this project DSBRAM\_IF\_CNTRL

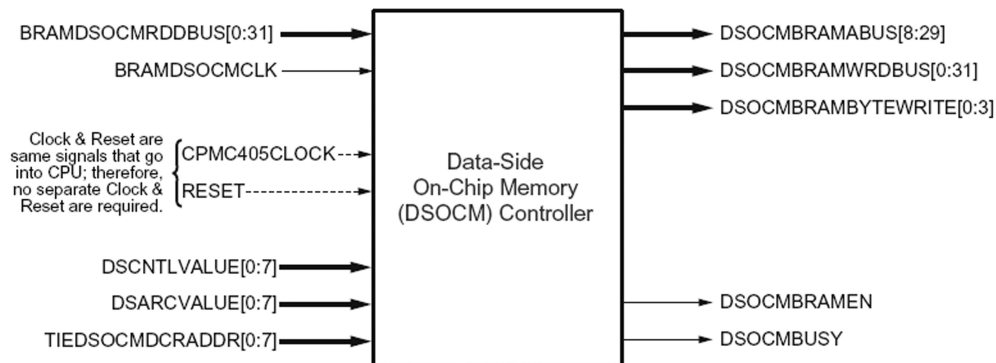


Figure 6: DSOCM Controller Block Diagramm [19]

is used. Memory of the specified size within the range of 2KB up to 128KB will be provided using 4 to 64 of the dual-ported Block Ram cells spread over the FPGA chip. Each of the two ports can be connected to independent BRAM Block controllers, providing great flexibility, as both ports can be used simultaneously. Several features of the Block RAM block may be customized using system software. For customization of Block RAM blocks, further reading of [15] is recommended.

Figure 7 shows a block diagram of a dual-ported BRAM block, where the signals are defined as follows:

*Input signals*

BRAM\_Rst\_A BRAM BRAM Reset, active high

BRAM\_Clk\_A BRAM BRAM clock

BRAM\_EN\_A BRAM BRAM enable, active high

BRAM\_WEN\_A BRAM [0:3] BRAM write enable, 4 bit for enabling byte-wide writing

BRAM\_Addr\_A [0:C\_PORT\_AWIDTH-1] BRAM address, bitwidth depends on amount of memory

BRAM\_Din\_A [0:C\_PORT\_DWIDTH-1] BRAM data *output*, referenced from point of view of the controller, bitwidth defaults to 32

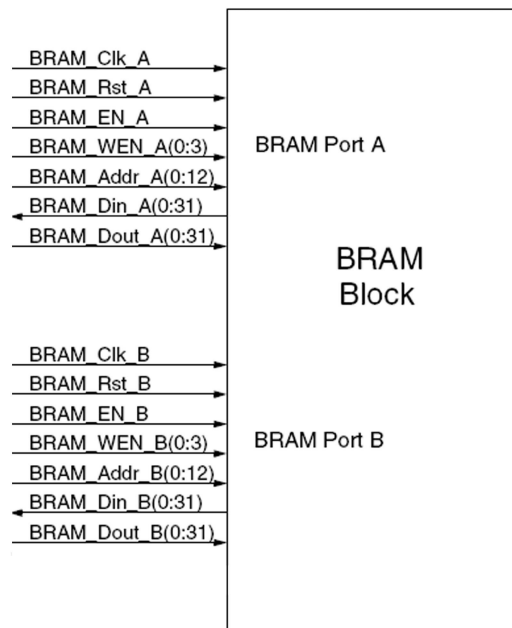


Figure 7: Block Ram Block Block Diagramm [15]

#### Output signals

BRAM\_Dout\_A [0:C\_PORT\_DWIDTH-1] BRAM data *input*, referenced from point of view of the controller, bitwidth defaults to 32

The signals on the B Port are defined accordingly, further details can be found in [15]. Figure 8 shows a DSOCM to BRAM block interface example, where one controller is connected to Port A of the BRAM block.

## 6.4 Faults in XPS 6.2i

Several difficulties were experienced while using the DSOCM during the project. Analysis of the VHDL code created by Xilinx Platform Studio showed, that the software did not work as expected. Even though the software did not report any errors, some of the DSOCM wrapper's signals were not properly connected to the PowerPC 405, which prevented the system of working as supposed (BRAMDSOCMRDDBUS  $\Rightarrow$  open, DSARCVALUE  $\Rightarrow$  open, DSCNTLVALUE  $\Rightarrow$  open). These signals had to be connected by hand editing



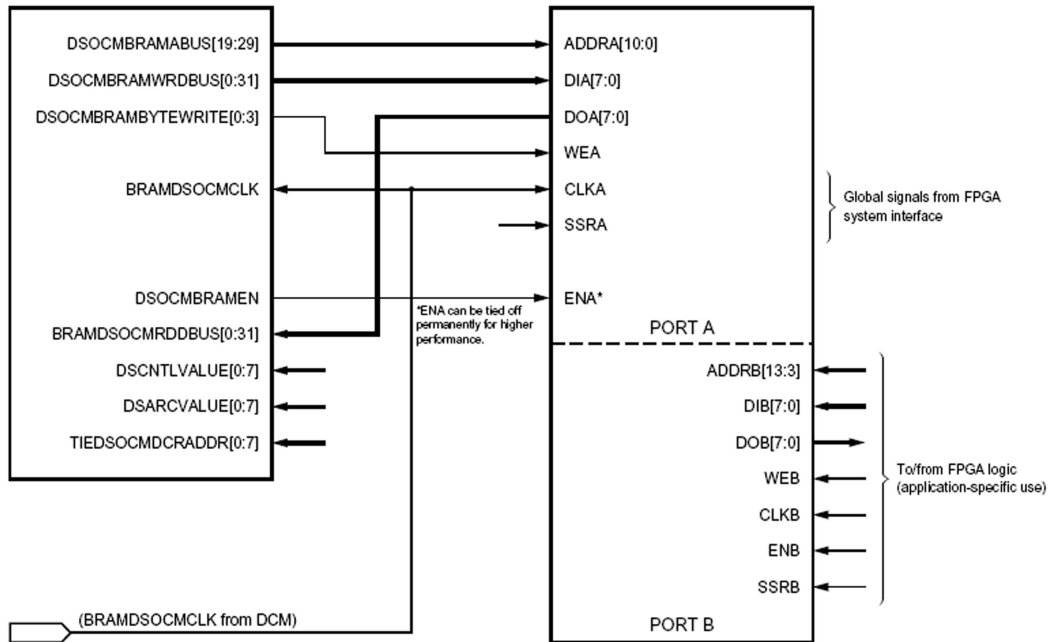


Figure 8: DSOCM to Block Ram Block Diagramm [18]

system master definition files. Furthermore the PowerPC 405 hardware version had to be changed to 2.00a in order to be able to assign it as a master of the DSOCM bus. Sometimes the project file had to be closed and re-opened to make the software realize changes in system definition files.

## 6.5 Interfacing BlockRAM on the B-Side

In order to access the BRAM block from the FPGA fabric using Port B, the user core needs to contain a BRAM Block controller. Its main function is to provide and handle the seven signals described in section 6.3 (clock, reset, enable, write\_enable, address, data\_in, data\_out), that are necessary to control and access the BRAM block.

In this project the BRAM block is only used as a means of data exchange between processor and FPGA fabric or to be more specific, the BRAM block is used to transfer the input data to the multiplier and the result back to the processor. These read and write operations on the BRAM block are highly

predictable, as the sequence of communication is fixed.

Therefore it seems to be adequate, to implement the controller as a Finite State Machine. Because every transition of the state machine is predictable it is deterministic and as the output depends on the present state and the input, the state machine is classified as a Mealy machine [5].

For reasons of simplicity, the following permanent signal assignments were made:

`BRAM_Rst_B <= rst;` connects the BRAM block reset signal to the system wide reset

`BRAM_Clk_B <= clk;` connects the BRAM block clock signal to the system clock (100MHz default)

`BRAM_EN_B <= '1';` enables the BRAM block for operation permanently

The activities of the controller will be limited to either loading data, writing data or ideling, thus, the state machine has to comprise several different states, each associated with one of the three activities. For the purpose of using the Montgomery Multiplier, it is necessary to pass four values from the processor to the multiplier: `factorA`, `factorB`, `modulus`, `R`. After the multiplier has completed the computation, the value `result` needs to be passed from the FPGA to the processor.

To perform a read operation, a valid address must be assigned to `BRAM_Addr_B`. The 32-bit value stored at that address can then be fetched from `BRAM_Din_B`, see code example I. In order to write a value into the BRAM block, a valid address must be assigned to `BRAM_Addr_B` as well. Furthermore, the value to be written must be present at `BRAM_Dout_B` and the `BRAM_WEN` bit(s) must be set to `high`.

As already mentioned in section 4.2.4, the implemented Montgomery Multiplier uses 64-bit operands. These have to be transferred to and from the BRAM block in two separate steps, because the DSOCM read and write busses are only 32-bit wide. VHDL code example II shows how to write a 64-bit value from a 64-bit signal `a` into the BRAM block starting at address `0x31000100`.

Table 9: VHDL code example I

read	write
BRAM_Addr_b <= X"31000100"; a <= BRAM_Din_B;	BRAM_Addr_b <= X"31000100"; BRAM_Dout_B <= result; BRAM_WEN_B <= "1111";

Table 10: VHDL code example II

cycle	command
1	BRAM_Addr_B <= X"31000100"; BRAM_Dout_B <= a(31 downto 0); BRAM_WEN_B <= "1111";
2	BRAM_Addr_B <= X"31000120"; BRAM_Dout_B <= a(63 downto 32); BRAM_WEN_B <= "1111";

## 6.6 Further aspects: timing, memory addressing, latch vs. register, etc.

This section provides further technical details: timing models and memory address mappings used during the project. Furthermore, details on the implementation style are provided.

Figure 9 shows the data load timing for DSOCM in single-cycle mode, which is used in this project. It means that the controller on processor-side and BRAM block run at the same clock speed. It can be observed, that a read operation needs at least two BRAM block clock cycles to complete. If `BRAM_Addr_B` is assigned with a valid address at the beginning of cycle  $n$ , the value stored at that memory address will be present at `BRAM_Din_B` in the beginning of cycle  $n+1$ . Therefore address- and data -bus commands for loading data from a BRAM block will be offset by one cycle, as follows. VHDL code example III shows how to read a 64-bit value from the BRAM block starting at address `0x31000100` into a 64-bit signal `a`.

Figure 10 shows the data store timing for DSOCM in single cycle mode.

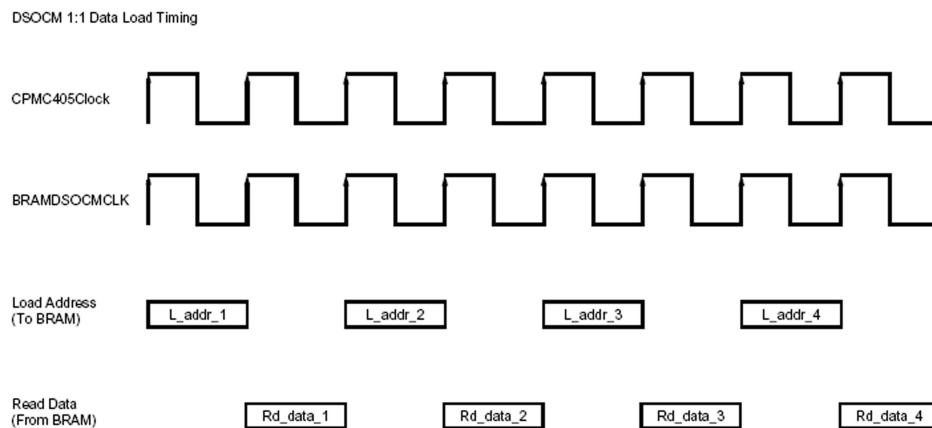


Figure 9: Data Load Timing in single cycle mode [18]

Table 11: VHDL code example III

cycle	command	comment
1	<code>BRAM_Addr_B &lt;= X"31000100";</code>	
2	<code>BRAM_Addr_B &lt;= X"31000120";</code> <code>BRAM_Dout_B &lt;= a(31 downto 0);</code>	reads data from address, set in cycle 1
3	<code>BRAM_Dout_B &lt;= a(63 downto 32);</code>	reads data from address, set in cycle 2

Note that a data store operation can be completed in two BRAM block clock cycles. If `BRAM_Addr_B` is assigned with a valid address at the beginning of cycle  $n$ , the 32-bit value present at `BRAM_Dout_B` at the beginning of cycle  $n$  will be stored in the BRAM block in cycle  $n+1$ . The next store operation must not start before the beginning of cycle  $n+2$ .

During this project, the amount of block RAM to be used was set to 8KB. This is far more memory than actually needed to exchange the 64-bit operands, but it would be well dimensioned, if it comes to 1024-bit operands. The 8KB were mapped into processor address space from `0x31000000` to `0x31001fff` and used according to the following tableau:

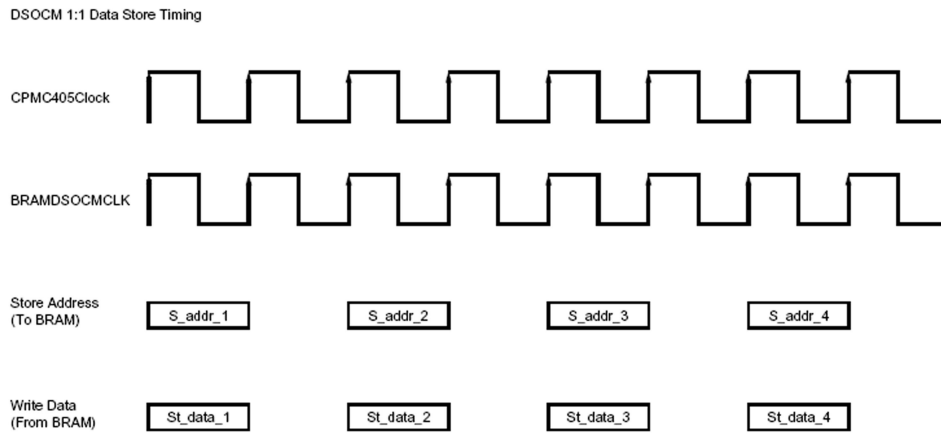


Figure 10: Data Store Timing in single cycle mode [18]

address	value	address	value
0000	A1	0020	A2
0040	B1	0060	B2
0080	MOD1	00A0	MOD2
00C0	PRIME1	00E0	PRIME2
0100	RESULT1	0120	RESULT2
03E0	CTRL		

Note that all memory addresses are prefixed by `0x3100`. The 64-bit values A, B, MOD, PRIME and RESULT are gained by a merge of A1, A2 ... RESULT1, RESULT2 32-bit values, respectively.

The 32-bit word `CTRL` is used to communicate processor and FPGA state to each other.

<code>CTRL = 0x00000000</code>	processor active, FPGA inactive
<code>CTRL = 0x00000001</code>	processor inactive, FPGA multiplying
<code>CTRL = 0x00000002</code>	processor active, FPGA done

Another important matter of fact is, that VHDL signal values should only be changed by clocked processes in order to actually use clocked registers rather than latches, which would be used if the process was not clocked.

## 6.7 The final hardware design

In this section the final hardware implementation is introduced. To ease understanding certain design decisions, several graphics are used.

Figure 11 introduces the concept of the deterministic Mealy state machine as it is implemented. Figure 12 shows the three main functional blocks of

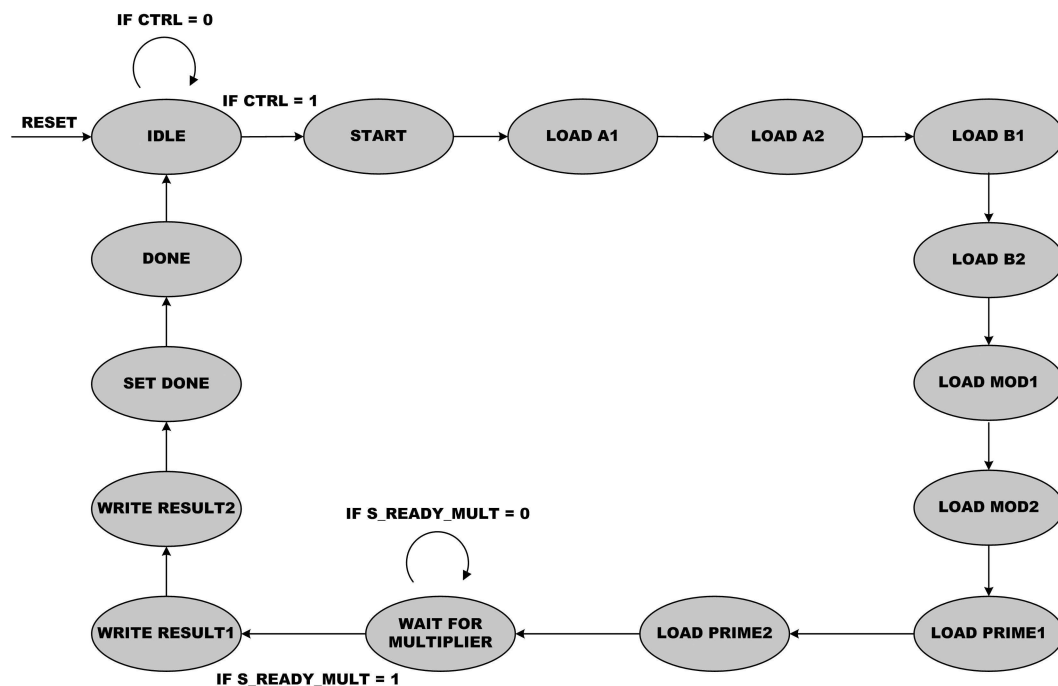


Figure 11: Deterministic Mealy state machine

the implementation, their ports, internal signals, systemwide signals, and indicates their co-act-ion.

Figure 13 gives an overview of the entire hardware system implemented in this project.

## 6.8 VHDL code

The VHDL code implementing the controller Finite State Machine is printed in appendix A. Appendix B contains the VHDL code implementing the 64-bit Montgomery Multiplier [3].

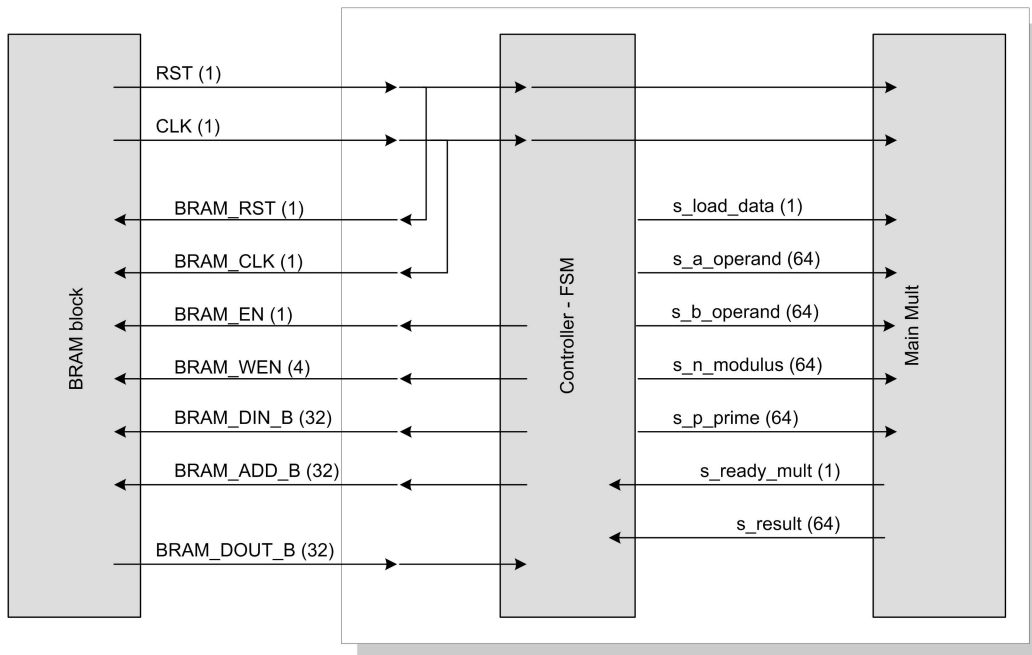


Figure 12: Model of functional blocks' co-action

## 7 Processor Application

The main part of this report deals with the implementation of a hardware multiplier and its interface to the PowerPC 405, although the report is titled *Hardware-Software CoDesign: a case study on an accelerated implementation of RSA*. This is due to the fact that, as explained in sections 3 and 4, only the part that makes up the bottleneck in RSA encryption is put into hardware. The *RSA Framework* remains in software, is written in C language, and cross-compiled for the target processor within the Xilinx Platform Studio Software. The source code developed in this project is printed in appendix C.

## 8 Results

The following is a performance estimation of the hardware accelerator. The 64-bit Montgomery multiplier computes the result within 20 clock cy-

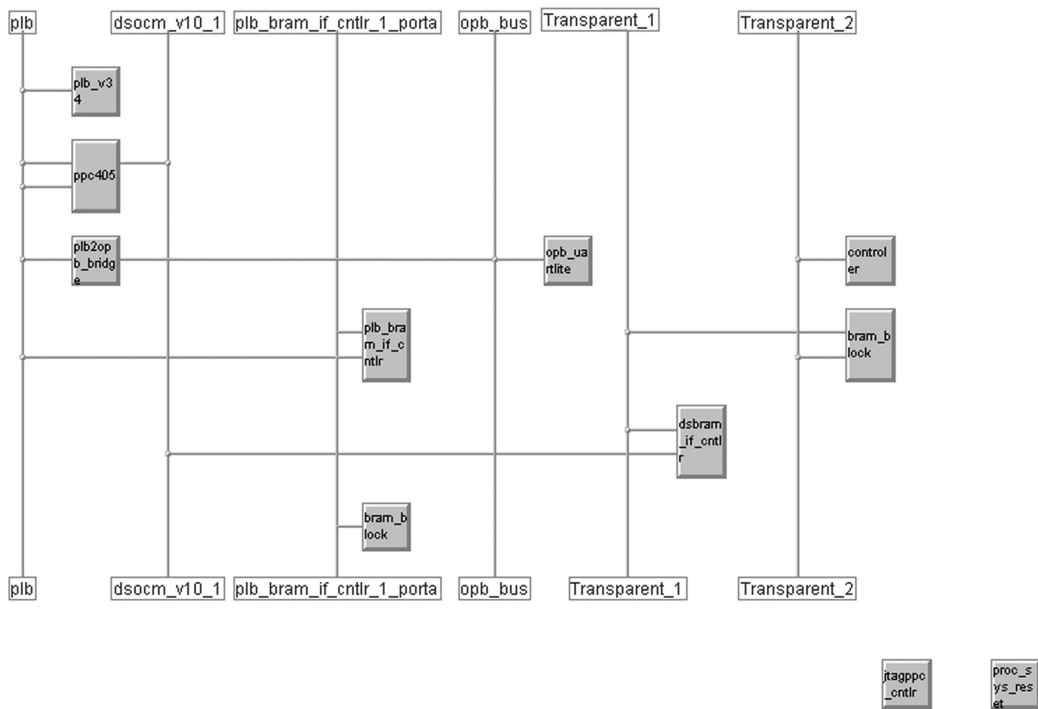


Figure 13: Screenshot - system hardware layout

cles, where the clock speed is not limited up to 100 MHz (default board clock speed).

The designed interface needs ten clock cycles to read all operands into the BRAM block and another four clock cycles, to write the result into the BRAM block after multiplication.

In total, the entire interfaced Montgomery multiplier needs 34 clock cycles to complete one multiplication operation, which equals 340ns per multiplication or 2.94 million multiplications per second at 100MHz clock speed.

Assuming that the 64-bit RSA encryption exponent consists of as many ones as zeros, 96 64-bit multiplications have to be computed. This can be done in 3264 clock cycles which equals 32.64  $\mu$ s per exponentiation or 30637 exponentiations per second at 100 MHz.

Note that additional time for processor operations has been neglected.



In section 3.1 it was mentioned, that the main disadvantage of a pure software implementation of RSA is the costly modular reduction operation which decelerates the performance dramatically.

In the course of this report one way to counteract this fact has been presented and shown to be efficient. By moving computationally intensive operators into hardware the performance of an application can normally be speeded up by several hundred percent. In this project not only the modular reduction operation is moved into hardware, but the entire modular multiplication operation. This is done due to the fact that the Montgomery Multiplier is the best hardware architecture known, so far, to perform modular multiplication and that moving the entire multiplication instead of only the reduction operation results in a further speed-up.

The Montgomery Multiplier and the interface that is needed to wire it to the main system, created by the Xilinx Platform Studio software, are designed in VHDL and implemented on a Xilinx VirtexII pro FPGA. The *RSA Framework* application remains in software and is run on the Power PC 405 microprocessor which is located within the same FPGA chip.

## A VHDL Code implemeting the controller Finite State Machine

```
-----  
-- Controller for user cores connected to OCM via B-side of Block RAM  
-- Benedikt Gierlichs, Ruhr-Universität Bochum  
-- Institute for electronics, communications, and information technology  
-- Queen's University Belfast  
-----
```

```
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_unsigned.all;
```

```
-----  
-- Port Declarations  
-----
```

```
-- Definition of Generics:
```

```
-- C_MEMSIZE           -- Memory Size  
-- C_PORT_DWIDTH      -- Data bus width  
-- C_PORT_AWIDTH      -- Address bus width  
-- C_NUM_WE           -- Number of write enables  
-- C_FAMILY           -- Target FPGA architecture  
--
```

```
-- Definition of Ports:
```

```
-- BRAM Port B  
-- BRAM_Rst           -- BRAM Port B reset  
-- BRAM_Clk           -- BRAM Port B clock  
-- BRAM_EN            -- BRAM Port B enable  
-- BRAM_WEN           -- BRAM Port B write enable  
-- BRAM_Addr          -- BRAM Port B address  
-- BRAM_Din           -- BRAM Port B data in  
-- BRAM_Dout          -- BRAM Port B data out  
-- clk                -- System Clock  
-- rst                -- Reset  
-----
```

```
entity controler is
```

```
generic
```

```
(  
  C_MEMSIZE      : integer := 2048;  
  C_PORT_DWIDTH  : integer := 32;  
  C_PORT_AWIDTH  : integer := 32;  
  C_NUM_WE       : integer := 4;  
  C_FAMILY       : string  := "virtex2"  
);
```

```
port
```

## A VHDL CODE - FINITE STATE MACHINE

---

```
(
  BRAM_Rst_B  : out std_logic;
  BRAM_Clk_B  : out std_logic;
  BRAM_EN_B   : out std_logic;
  BRAM_WEN_B  : out std_logic_vector(0 to C_NUM_WE-1);
  BRAM_Addr_B : out std_logic_vector(0 to C_PORT_AWIDTH-1);
  BRAM_Din_B  : in  std_logic_vector(0 to C_PORT_DWIDTH-1);
  BRAM_Dout_B : out std_logic_vector(0 to C_PORT_DWIDTH-1);
  clk         : in  std_logic;
  rst         : in  std_logic
);

end entity controler;

architecture dfl of controler is

-----
-- Signals
-----

signal s_a_operand :std_logic_vector(63 downto 0);
signal s_b_operand :std_logic_vector(63 downto 0);
signal s_n_modulus :std_logic_vector(63 downto 0);
signal s_n_prime   :std_logic_vector(63 downto 0);
signal s_load_data :std_logic;
signal s_ready_mult :std_logic;
signal s_result    :std_logic_vector(63 downto 0);
signal clock2     :std_logic;
signal cnt4clk    :std_logic_vector(3 downto 0);
-----

-- FSM stuff
-----

type fsm_state is (IDLE, START, LOAD_A1, LOAD_A2, LOAD_B1, LOAD_B2, LOAD_MOD1,
  LOAD_MOD2, LOAD_PRIME1, LOAD_PRIME2, WAIT_FOR_USER_CORE,
  WRITE_1, WRITE_2, SET_DONE, DONE);

signal state, next_state : fsm_state;
-----

-----
--Component Declaration
-----

component main_mult is

port
```

## A VHDL CODE - FINITE STATE MACHINE

---

```
(  
clk      :in std_logic;  
reset   :in std_logic;  
load_data :in std_logic;  
a_operand :in std_logic_vector(63 downto 0);  
b_operand :in std_logic_vector(63 downto 0);  
n_modulus :in std_logic_vector(63 downto 0);  
n_prime  :in std_logic_vector(63 downto 0);  
ready   :out std_logic;  
mod_ab  :out std_logic_vector(63 downto 0)  
);
```

```
end component main_mult;
```

```
-----  
  
begin
```

```
BRAM_Rst_B <= rst;  
BRAM_Clk_B <= clk;  
BRAM_EN_B  <= '1';
```

```
-----  
--Component Instantiation  
-----
```

```
Main_Mult_1:main_mult  port map(clk => clk,  
                                reset => rst,  
                                load_data => s_load_data,  
                                a_operand => s_a_operand,  
                                b_operand => s_b_operand,  
                                n_modulus => s_n_modulus,  
                                n_prime  => s_n_prime,  
                                ready   => s_ready_mult,  
                                mod_ab  => s_result);
```

```
-----  
-- FSM init  
-----
```

```
state_change: process(clk, rst)  
begin  
IF ( rst = '1') THEN  
state <= IDLE;  
ELSIF (clk'event and clk = '1') THEN  
state <= next_state;  
END IF;  
end process;
```

## A VHDL CODE - FINITE STATE MACHINE

---

```
-----  
state_compute : process(state, BRAM_Din_B, s_result, s_ready_mult)
```

```
begin
```

```
-- Default values--
```

```
next_state <= IDLE;
```

```
BRAM_WEN_B <= "0000";
```

```
    BRAM_Addr_B <= X"310003E0";
```

```
BRAM_Dout_B <= X"00000000";
```

```
-- /Default values--
```

```
case state is
```

```
when IDLE =>
```

```
IF ( BRAM_Din_B(31) = '1' ) THEN
```

```
    BRAM_WEN_B <= "1111"; -- this sets control
```

```
    next_state <= START; -- bit back to 0
```

```
END IF;
```

```
-----  
-- to read from block RAM the address must be set one cycle
```

```
-- before the actual read operation, writing to block RAM is a
```

```
-- one cycle operation  
-----
```

```
when START =>
```

```
BRAM_Addr_B <= X"31000000";
```

```
next_state <= Load_A1;
```

```
when LOAD_A1 =>
```

```
BRAM_Addr_B <= X"31000020";
```

```
next_state <= LOAD_A2;
```

```
when LOAD_A2 =>
```

```
BRAM_Addr_B <= X"31000040";
```

```
next_state <= LOAD_B1;
```

```
when LOAD_B1 =>
```

```
BRAM_Addr_B <= X"31000060";
```

```
next_state <= LOAD_B2;
```

```
when LOAD_B2 =>
```

```
BRAM_Addr_B <= X"31000080";
```

```
next_state <= LOAD_MOD1;
```

```
when LOAD_MOD1 =>
```

## A VHDL CODE - FINITE STATE MACHINE

---

```
BRAM_Addr_B <= X"310000A0";
next_state <= LOAD_MOD2;

when LOAD_MOD2 =>
BRAM_Addr_B <= X"310000C0";
next_state <= LOAD_PRIME1;

when LOAD_PRIME1 =>
BRAM_Addr_B <= X"310000E0";
next_state <= LOAD_PRIME2;

when LOAD_PRIME2 =>
-- address was set in last cycle --
next_state <= WAIT_FOR_USER_CORE;

when WAIT_FOR_USER_CORE =>
IF (s_ready_mult = '1') THEN
next_state <= WRITE_1;
ELSE
next_state <= WAIT_FOR_USER_CORE;
END IF;

when WRITE_1 =>
BRAM_Addr_b <= X"31000100";
BRAM_WEN_B <= "1111";
BRAM_Dout_B <= s_result(63 downto 32);
next_state <= WRITE_2;

when WRITE_2 =>
BRAM_Addr_b <= X"31000120";
BRAM_WEN_B <= "1111";
BRAM_Dout_B <= s_result(31 downto 0);
next_state <= SET_DONE;

when SET_DONE =>
BRAM_WEN_B <= "1111";
BRAM_Dout_B <= X"00000002";
next_state <= DONE;

when DONE =>
next_state <= IDLE;

when others =>
next_state <= IDLE;

end case;

end process;
```

---

## A VHDL CODE - FINITE STATE MACHINE

---

```
store_values : process(clk, rst)
begin

if (rst = '1') then
s_load_data <= '0';
s_a_operand <= (others => '0');
s_b_operand <= (others => '0');
s_n_modulus <= (others => '0');
s_n_prime <= (others => '0');

elseif( clk'event and clk = '1') then

if ( state = START ) THEN
s_load_data <= '1';

ELSIF ( state = LOAD_A1 ) THEN
s_a_operand(63 downto 32) <= BRAM_Din_B;

ELSIF ( state = LOAD_A2 ) THEN
s_a_operand(31 downto 0) <= BRAM_Din_B;

ELSIF ( state = LOAD_B1 ) THEN
s_b_operand(63 downto 32) <= BRAM_Din_B;

ELSIF ( state = LOAD_B2 ) THEN
s_b_operand(31 downto 0) <= BRAM_Din_B;

ELSIF ( state = LOAD_MOD1 ) THEN
s_n_modulus (63 downto 32) <= BRAM_Din_B;

ELSIF ( state = LOAD_MOD2 ) THEN
s_n_modulus (31 downto 0) <= BRAM_Din_B;

ELSIF ( state = LOAD_PRIME1 ) THEN
s_n_prime(63 downto 32) <= BRAM_Din_B;

ELSIF ( state = LOAD_PRIME2 ) THEN
s_n_prime (31 downto 0) <= BRAM_Din_B;
s_load_data <= '0';

END IF;
END IF;
end process;
```

```
-----

Clocking:process(clk,rst)
begin
```

## A VHDL CODE - FINITE STATE MACHINE

---

```
    if rst = '1' then
        cnt4clk <= "0000";
        clock2 <= '0';

    elsif clk'event and clk = '1' then

        if cnt4clk = "0000" THEN
            clock2 <= '0';
            cnt4clk <= "0001";

            elsif cnt4clk = "0010" THEN
clock2 <= '1';
cnt4clk <= "0011";

            elsif cnt4clk = "0011" THEN
cnt4clk <= "0000";

            else cnt4clk <= cnt4clk + 1;

        end if;
    end if;
end process Clocking;

end architecture;
```



## B VHDL code implementing the 64-bit Montgomery Multiplier

### B.1 main\_mult.vhd

```
--Main Mult-- --64 Bit--
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
```

```
-----
entity main_mult is
```

```
port
(
clk :in std_logic;
reset :in std_logic;
load_data :in std_logic;
a_operand :in std_logic_vector(63 downto 0);
b_operand :in std_logic_vector(63 downto 0);
n_modulus :in std_logic_vector(63 downto 0);
n_prime :in std_logic_vector(63 downto 0);
ready :out std_logic;
mod_ab :out std_logic_vector(63 downto 0)
);
```

```
constant width :integer := 64;
```

```
end main_mult;
```

```
-----
architecture main_mult of main_mult is
```

```
-----
component main_mult_counter is
```

```
port
(
clk :in std_logic;
reset :in std_logic;
load_data :in std_logic;
main_mult_cnt :out std_logic_vector(4 downto 0)
);
```

## B.1 main\_mult.vhd B VHDL CODE - MONTGOMERY MULTIPLIER

```
end component main_mult_counter;
```

```
-----  
component mult_64bits is
```

```
port  
(  
  clk      :in std_logic;  
  reset   :in std_logic;  
  a       :in std_logic_vector(width-1 downto 0);  
  b       :in std_logic_vector(width-1 downto 0);  
  p       :out std_logic_vector((2*width)-1 downto 0)  
);
```

```
end component mult_64bits;
```

```
-----  
signal main_mult_cnt      :std_logic_vector(4 downto 0);  
signal a1_mult            :std_logic_vector(width-1 downto 0);  
signal b1_mult            :std_logic_vector(width-1 downto 0);  
signal p1_mult            :std_logic_vector((2*width)-1 downto 0);  
signal p1_mult_sig        :std_logic_vector((2*width)-1 downto 0);  
signal u_sig              :std_logic_vector(width downto 0);
```

```
begin
```

```
-----  
Main_Mult_Counter1:main_mult_counter
```

```
port map  
(  
  clk,  
  reset,  
  load_data,  
  main_mult_cnt  
);
```

```
-----  
Mult_64Bits1:mult_64bits
```

```
port map  
(  
  clk,  
  reset,  
  a1_mult,  
  b1_mult,
```

## B.1 main\_mult.vhd B VHDL CODE - MONTGOMERY MULTIPLIER

```
p1_mult
);

-----

Mult_Operations:process(clk, reset)

begin

    if (reset = '1') then
        a1_mult <= (others => '0');
        b1_mult <= (others => '0');
        p1_mult_sig <= (others => '0');

    elsif (clk'event and clk = '1') then

        case main_mult_cnt is

            when "00001" =>
a1_mult <= a_operand;
                b1_mult <= b_operand;

            when "00111" =>--mult_time + 1--
p1_mult_sig <= p1_mult;
a1_mult <= p1_mult(width-1 downto 0);
b1_mult <= n_prime;

            when "01101" => --mult_time + 1-- --%1--
a1_mult <= p1_mult(width-1 downto 0);
b1_mult <= n_modulus;

            when others =>
                null;

        end case;

    end if;

end process Mult_Operations;

-----

Main_Operations:process(clk, reset)

variable u_var :std_logic_vector(2*width downto 0);
variable mod_ab_var :std_logic_vector(width downto 0);

begin

    if (reset = '1') then
```

## B.1 main\_mult.vhd B VHDL CODE - MONTGOMERY MULTIPLIER

```
u_sig <= (others => '0');
mod_ab <= (others => '0');

elsif (clk'event and clk = '1') then

    case main_mult_cnt is

        when "10011" =>--%1 + mult_time +1--
            u_var := ('0' & p1_mult_sig) + ('0' & p1_mult);
            u_sig <= u_var(2*width downto width);

        when "10100" =>--+1--
            if (u_sig >= n_modulus) then
                mod_ab_var := u_sig - n_modulus;
                mod_ab <= mod_ab_var(width-1 downto 0);
            else
                mod_ab <= u_sig(width-1 downto 0);

                end if;

            when others =>
                null;

    end case;

end if;

end process Main_Operations;

-----

Ready_Signal:process(clk,reset)

begin

    if (reset = '1') then
        ready <= '0';

    elsif (clk'event and clk = '1') then

        if (main_mult_cnt = "10100") then
            ready <= '1';
        else
            ready <= '0';
        end if;
    end if;

end process Ready_Signal;

-----
```

```
end main_mult;
```

## B.2 *main\_mult\_counter.vhd*

```
--Main Mult Counter--
```

```
library ieee;
```

```
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_unsigned.all;
```

```
-----  
entity main_mult_counter is
```

```
port(clk          :in std_logic;  
      reset       :in std_logic;  
      load_data   :in std_logic;  
      main_mult_cnt :out std_logic_vector(4 downto 0));
```

```
end main_mult_counter;
```

```
-----  
architecture main_mult_counter of main_mult_counter is
```

```
-----  
    signal main_mult_cnt_sig :std_logic_vector(4 downto 0) := (others => '0');
```

```
-----  
begin
```

```
-----  
    Count:process(clk,reset)
```

```
    begin
```

```
        if reset = '1' then
```

```
            main_mult_cnt_sig <= (others => '0');
```

```
        elsif clk'event and clk = '1' then
```

```
            if load_data = '1' then
```

```
                main_mult_cnt_sig <= (others => '0');
```

### B.3 mult\_64Bits.vhd B VHDL CODE - MONTGOMERY MULTIPLIER

```
        else

            if main_mult_cnt_sig = "10100" then

                main_mult_cnt_sig <= "00001";

            else

                main_mult_cnt_sig <= main_mult_cnt_sig + 1;

            end if;

        end if;

    end if;

end process Count;

-----

    main_mult_cnt <= main_mult_cnt_sig;

-----

end main_mult_counter;
```

### B.3 mult\_64Bits.vhd

```
--Mult_64Bits--

library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

-----

entity mult_64bits is

    port(clk      :in std_logic;
          reset   :in std_logic;
          a       :in std_logic_vector(63 downto 0);
          b       :in std_logic_vector(63 downto 0);
          p       :out std_logic_vector(127 downto 0));

    subtype mult_input_32bits is std_logic_vector(31 downto 0);
    type mult_input_32bits_array is array(1 downto 0) of mult_input_32bits;
```

### B.3 mult\_64Bits.vhd B VHDL CODE - MONTGOMERY MULTIPLIER

```
    subtype mult_output_32bits is std_logic_vector(63 downto 0);
    type mult_output_32bits_array is array(1 downto 0) of mult_output_32bits;

end mult_64bits;
```

```
-----

architecture mult_64bits of mult_64bits is
```

```
-----

    component mult_32bits is
        port(clk      :in std_logic;
              reset   :in std_logic;
              a       :in std_logic_vector(31 downto 0);
              b       :in std_logic_vector(31 downto 0);
              p       :out std_logic_vector(63 downto 0));
    end component mult_32bits;
```

```
-----

    signal a_mult   :mult_input_32bits_array;
    signal b_mult   :mult_input_32bits_array;
    signal p_mult1  :mult_output_32bits_array;
    signal p_mult2  :mult_output_32bits_array;
    signal p_sig    :std_logic_vector(96 downto 0);
```

```
-----

begin
```

```
-----

    a_mult(1) <= a(63 downto 32);

    a_mult(0) <= a(31 downto 0);

    b_mult(1) <= b(63 downto 32);

    b_mult(0) <= b(31 downto 0);
```

```
-----

    Partial_Product_Generate1:for i in 1 downto 0 generate
```

```
        mult_32bits_1:mult_32bits    port map(clk,
                                                reset,
                                                a_mult(i),
                                                b_mult(i),
                                                p_mult1(i));
```

## B.4 mult\_32Bits.vhd B VHDL CODE - MONTGOMERY MULTIPLIER

```
end generate Partial_Product_Generate1;

-----

Partial_Product_Generate2:for i in 1 downto 0 generate

    mult_32bits_2:mult_32bits    port map(clk,
                                        reset,
                                        a_mult(1),
                                        b_mult(i),
                                        p_mult2(i));

end generate Partial_Product_Generate2;

-----

p_sig(31 downto 0) <= (others => '0');

-----

Add_Partial_Products:process(clk,reset)

begin

    if reset = '1' then

        p_sig(96 downto 32) <= (others => '0');

        p <= (others => '0');

    elsif clk'event and clk = '1' then

        p_sig(96 downto 32) <= ('0' & p_mult1(1)) + ('0' & p_mult2(0));

        p <= p_sig + (p_mult2(1) & p_mult1(0));

    end if;

end process Add_Partial_Products;

-----

end mult_64bits;
```

## B.4 mult\_32Bits.vhd

```
--Mult_32Bits--
```

```
library ieee;
```



## B.4 mult\_32Bits.vhd B VHDL CODE - MONTGOMERY MULTIPLIER

```
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

-----

entity mult_32bits is

    port(clk      :in std_logic;
          reset   :in std_logic;
          a       :in std_logic_vector(31 downto 0);
          b       :in std_logic_vector(31 downto 0);
          p       :out std_logic_vector(63 downto 0));

    subtype mult_input_16bits is std_logic_vector(15 downto 0);
    type mult_input_16bits_array is array(1 downto 0) of mult_input_16bits;

    subtype mult_output_16bits is std_logic_vector(31 downto 0);
    type mult_output_16bits_array is array(1 downto 0) of mult_output_16bits;

end mult_32bits;

-----

architecture mult_32bits of mult_32bits is

    component mult_16bits is
        port (clk      :in std_logic;
              reset   :in std_logic;
              a       :in std_logic_vector(15 downto 0);
              b       :in std_logic_vector(15 downto 0);
              p       :out std_logic_vector(31 downto 0));
    end component mult_16bits;

    signal a_mult      :mult_input_16bits_array;
    signal b_mult      :mult_input_16bits_array;
    signal p_mult1     :mult_output_16bits_array;
    signal p_mult2     :mult_output_16bits_array;
    signal p_sig       :std_logic_vector(48 downto 0);

    -----

begin

    -----
```

## B.4 mult\_32Bits.vhd B VHDL CODE - MONTGOMERY MULTIPLIER

```
a_mult(1) <= a(31 downto 16);
```

```
a_mult(0) <= a(15 downto 0);
```

```
b_mult(1) <= b(31 downto 16);
```

```
b_mult(0) <= b(15 downto 0);
```

```
-----  
Partial_Product_Generate1:for i in 1 downto 0 generate
```

```
    mult_16bits_1:mult_16bits    port map(clk,  
                                           reset,  
                                           a_mult(0),  
                                           b_mult(i),  
                                           p_mult1(i));
```

```
end generate Partial_Product_Generate1;
```

```
-----  
Partial_Product_Generate2:for i in 1 downto 0 generate
```

```
    mult_16bits_2:mult_16bits    port map(clk,  
                                           reset,  
                                           a_mult(1),  
                                           b_mult(i),  
                                           p_mult2(i));
```

```
end generate Partial_Product_Generate2;
```

```
-----  
p_sig(15 downto 0) <= (others => '0');
```

```
-----  
Add_Partial_Products:process(clk,reset)
```

```
begin
```

```
    if reset = '1' then
```

```
        p_sig(48 downto 16) <= (others => '0');
```

```
        p <= (others => '0');
```

```
    elsif clk'event and clk = '1' then
```

## B.5 mult\_16bits.vhd B VHDL CODE - MONTGOMERY MULTIPLIER

```
        p_sig(48 downto 16) <= ('0' & p_mult1(1)) + ('0' & p_mult2(0));

        p <= p_sig + (p_mult2(1) & p_mult1(0));

    end if;

end process Add_Partial_Products;

-----

end mult_32bits;
```

### **B.5 mult\_16bits.vhd**

--Mult\_16Bits--

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
```

-----

```
entity mult_16bits is
```

```
    port (clk    :in std_logic;
          reset  :in std_logic;
          a      :in std_logic_vector(15 downto 0);
          b      :in std_logic_vector(15 downto 0);
          p      :out std_logic_vector(31 downto 0));
```

```
end mult_16bits;
```

-----

```
architecture mult_16bits of mult_16Bits is
```

```
begin
```

-----

```
    DFF:process(clk, reset)
```

```
    begin
```

```
        if reset = '1' then
```

```
            p(31 downto 0) <= (others => '0');
```

```
        elsif clk'event and clk = '1' then

            p(31 downto 0) <= a * b;

        end if;

    end process DFF;

-----

end mult_16bits;
```

## C C source code

```
#include <xuartlite_1.h>
#include <xparameters.h>
#include <xexception_1.h>
```

```
-----
-- General definitions
-----
#define size_of_bignum 64

-----
-- Definition of the structure we're going to use
-----
typedef struct
{
    int word[size_of_bignum];
} bignum;

-----
-- Definition of the structure we're going to use
-----
char test_a_bit();
short get_bitlength();
void mul();

int main(void) {

    bignum base;
    bignum exponent;
    bignum modulus;
```

```
bignum prime;
bignum result;
bignum temp;
bignum K;

int temptemp;
short bitpos;

temptemp = 1;
bitpos = 0;

base.word[0] = 0xD431;
base.word[1] = 0x0;

exponent.word[0] = 0x25318523;
exponent.word[1] = 0x0;

modulus.word[0] = 0x8000013B;
modulus.word[1] = 0x0;

prime.word[0] = 0x8D00D00D;
prime.word[1] = 0x0D979124;

K.word[0] = 0x2D7EBD3D;
K.word[1] = 0x0;

result.word[0] = 0x0;
result.word[1] = 0x0;

temp.word[0] = 0x0;
temp.word[1] = 0x0;

print("\033[H\033[J"); //clears the screen
xil_printf("base = %x %x \t", base.word[1], base.word[0]);
xil_printf("exponent = %x %x \t", exponent.word[1], exponent.word[0]);
xil_printf("modulus = %x %x \t", modulus.word[1], modulus.word[0]);
print("\r\n");
xil_printf("prime = %x %x \t", prime.word[1], prime.word[0]);
xil_printf("K = %x %x \t", K.word[1], K.word[0]);
print("\r\n");

// base into montgom domain...
mul( &K.word[1], &K.word[0], &base.word[1], &base.word[0],
    &modulus.word[1], &modulus.word[0], &prime.word[1], &prime.word[0],
    &temp.word[1], &temp.word[0]);
xil_printf("base after transfer to montgomery domain = %x %x \t",
    temp.word[1], temp.word[0]);

// exponentiation done using square and multiply
for ( bitpos = get_bitlength(&exponent) -1; bitpos > -1; bitpos -- )
```

```

{
mul( &temp.word[1], &temp.word[0], &temp.word[1], &temp.word[0],
    &modulus.word[1], &modulus.word[0], &prime.word[1],
    &prime.word[0], &temp.word[1], &temp.word[0]);

if ( test_a_bit( &exponent.word[bitpos/32], bitpos ) == 1 )
{
mul( &temp.word[1], &temp.word[0], &base.word[1],
    &base.word[0], &modulus.word[1], &modulus.word[0],
    &prime.word[1], &prime.word[0], &temp.word[1],
    &temp.word[0]);
}
}
mul( &temp.word[1], &temp.word[0], &temptemp, &temptemp, &modulus.word[1],
    &modulus.word[0], &prime.word[1], &prime.word[0], &temp.word[1],
    &temp.word[0]);
print("\r\n\r\n");
xil_printf("result = %x %x \t", temp.word[1], temp.word[0]);
}

```

```

char test_a_bit(int *temp, int bit)
{
if ( *temp & (1<<bit%32) )
{
return 1;
}
else
{
return 0;
}
}

```

```

short get_bitlength(bignum *number)
{
short bitpos;
short intpos = 64;
char found = 0;

// this loop hops through the interger numbers the bignum consists of
while ( !found && (intpos>= 0) )
{
intpos = intpos - 1;
bitpos = 31;

//now the inner loop searches the single integer numbers for the first
//non-zero bit
while ( !found && bitpos > -1 )
{
if ( test_a_bit(&(*number).word[intpos], bitpos) )

```

```
{
found = 1;
}
else
{
bitpos = bitpos - 1;
}
}

// IMPORTANT: this function returns the number of bits!!!!
// It does NOT return the position of the first non-zero bit!!
// That is eg:
// 4 = 100 in binary, the number of bits is 3 while the first non-zero bit
// is at position 2, as we start counting at the LSB.
//
////////////////////////////////////
return (intpos*32 + bitpos + 1);
}

-----
--
-----

void mul(int *factorah, int *factoral, int *factorbh, int *factorbl, int *modh,
int *modl, int *primeh, int *primel, int *resulth, int *resultl)
{
int *ptr_to_a1;
int *ptr_to_a2;
int *ptr_to_b1;
int *ptr_to_b2;
int *ptr_to_mod1;
int *ptr_to_mod2;
int *ptr_to_prime1;
int *ptr_to_prime2;
int *ptr_to_output1;
int *ptr_to_output2;
int *ptr_to_ctrl;

ptr_to_a1 = 0x31000000;
*ptr_to_a1 = 0x00000000;
ptr_to_a2 = 0x31000020;
*ptr_to_a2 = 0x00000000;

ptr_to_b1 = 0x31000040;
*ptr_to_b1 = 0x00000000;
ptr_to_b2 = 0x31000060;
*ptr_to_b2 = 0x00000000;
```

```
ptr_to_mod1 = 0x31000080;
*ptr_to_mod1 = 0x00000000;
ptr_to_mod2 = 0x310000A0;
*ptr_to_mod2 = 0x00000000;

ptr_to_prime1 = 0x310000C0;
*ptr_to_prime1 = 0x00000000;
ptr_to_prime2 = 0x310000E0;
*ptr_to_prime2 = 0x00000000;

ptr_to_output1 = 0x31000100;
*ptr_to_output1 = 0x00000000;
ptr_to_output2 = 0x31000120;
*ptr_to_output2 = 0x00000000;

ptr_to_ctrl = 0x310003E0;
*ptr_to_ctrl = 0x00000000;

// xil_printf("a = %x %x \t", *ptr_to_a1, *ptr_to_a2);
// xil_printf("b = %x %x \t", *ptr_to_b1, *ptr_to_b2);
// xil_printf("mod = %x %x \t", *ptr_to_mod1, *ptr_to_mod2);
// xil_printf("prime = %x %x \t", *ptr_to_prime1, *ptr_to_prime2);
// xil_printf("output = %x %x \t", *ptr_to_output1, *ptr_to_output2);
// xil_printf("ctrl = %d \r\n\n", *ptr_to_ctrl);

*ptr_to_a1 = *factorah;
*ptr_to_a2 = *factoral;

*ptr_to_b1 = *factorbh;
*ptr_to_b2 = *factorbl;

*ptr_to_mod1 = *modh;
*ptr_to_mod2 = *modl;

*ptr_to_prime1 = *primeh;
*ptr_to_prime2 = *primel;

// start the hardware processing unit
*ptr_to_ctrl = 0x00000001;
print("");

while ( *ptr_to_ctrl != 0x00000002 )
{
// DO NOTHING!
}
*ptr_to_ctrl = 0x00000000;
print("");
```



```
// xil_printf("a = %x %x \t", *ptr_to_a1, *ptr_to_a2);
// xil_printf("b = %x %x \t", *ptr_to_b1, *ptr_to_b2);
// xil_printf("mod = %x %x \t", *ptr_to_mod1, *ptr_to_mod2);
// xil_printf("prime = %x %x \t", *ptr_to_prime1, *ptr_to_prime2);
// xil_printf("output = %x %x \t", *ptr_to_output1, *ptr_to_output2);
// xil_printf("ctrl = %d \r\n\n", *ptr_to_ctrl);
// print("done");

*resulth = *ptr_to_output1;
*resultl = *ptr_to_output2;

}
```

## D Test vectors

Set1:

```
r      = 100000000000000000
n      = 32ED94BEAFAD89AD
r-1   = 1F232C33B2A120A3
nprime = 9C84B039A1513DDB
a      = 7E8C0146A7158418
b      = 1D7F356E6E2F27F6
monpro = 23CFFA7944CC169D
```

Set 2:

```
n      = 88b53d612dd5b053
r      = 100000000000000000
r-1   = 43c11cf82e823d94
nprime = 7ee08ae63a0bec25
a      = 76bb25365f319426
b      = 571f169d04a5735b
monpro = 565b25c421c077cd
```

## References

- [1] Menezes, van Oorshot, Vanstone (1997), Handbook of applied cryptography, CRC Press
- [2] Marcelo E. Kaihara and Naofumi Takagi, A VLSI Algorithm for Modular Multiplication/Division, <http://www.dec.usc.es/arith16/papers/paper-180.pdf>
- [3] McIvor, Ciaran (2005), Algorithms and Silicon Architectures for Public-key Cryptography
- [4] Avenet Inc. (2003), Xilinx Virtex-II Pro Evaluation Kit, Xilinx Virtex-II Pro Eval Kit - User's Guide 102103F.pdf
- [5] Mealy, G. H. (1955), A Method for Synthesizing Sequential Circuits, Bell System Tech. J. vol 34, pp. 1045 - 1079
- [6] Montgomery, P.L. (1985), Modular Multiplication without Trial Division, Math. Computation, Vol. 44, pp. 519-521
- [7] Moore, Gordon E. (1965), Cramming more components onto integrated circuits, Electronics, Volume 38, Number 8
- [8] R.L. Rivest, A. Shamir, and L. Adleman (1978), A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM 21,2, pp. 120-126
- [9] Xilinx (2004), Processor IP Reference Guide, [proc\\_ip\\_ref\\_guide.pdf](#)
- [10] Xilinx (2004), Processor Local Bus PLB v3.4, [plb\\_v34.pdf](#)
- [11] Xilinx (2004), OPB Usage in Xilinx FPGAs, [proc\\_ip\\_ref\\_guide.pdf](#)
- [12] Xilinx (2004), OPB to PLB Bridge (v1.00b), [proc\\_ip\\_ref\\_guide.pdf](#)
- [13] Xilinx (2004), OPB UART Lite, [opb\\_uartlite.pdf](#)
- [14] Xilinx (2004), PLB Block RAM (BRAM) Interface Controller, [plb\\_bram\\_if\\_ctrl.pdf](#)

- [15] Xilinx (2004), Block RAM (BRAM) Block, bram\_block.pdf
- [16] Xilinx (2004), Processor System Reset Module, proc\_sys\_reset.pdf
- [17] Xilinx (2004), JTAGPPC Controller, jtagppc\_cntlr.pdf
- [18] Xilinx (2004), PowerPC 405 Processor Block Reference Guide, ppc405block\_ref\_guide.pdf
- [19] Xilinx (2004), Data Side OCM Bus V1.0, dsocm\_v10.pdf