**INTERFACE CONTROL DOCUMENT**

**Hardware Implementation of 4096-Bit RSA Modular Exponentiation**

**APPLIED CRYPTOLAB UCSB**

**DEPARTMENT OF COMPUTER SCIENCE**

**UNIVERSITY OF CALIFORNIA, SANTA BARBARA**

***Version:*** 0.2

***Guided by:*** Prof. Çetin Kaya Koç ([koc@cs.ucsb.edu](mailto:koc@cs.ucsb.edu))

***Written by:*** Qin Zhou ([qinzhou@cs.ucsb.edu](mailto:qinzhou@cs.ucsb.edu))

Balasubramaniam Muthuvelu ([balasubramaniam@umail.ucsb.edu](mailto:balasubramaniam@umail.ucsb.edu))

|  |  |
| --- | --- |
| College Logo | http://www.aw.id.ucsb.edu/logos/images/ucsbwave_144.jpg |

TABLE OF CONTENTS

[VERSION CONTROL HISTORY 2](#_Toc355598467)

[1. INTRODUCTION 4](#_Toc355598468)

[1.1 RSA Algorithm 4](#_Toc355598469)

[1.2 RSA Implementation 4](#_Toc355598470)

[1.3 Montgomery transform based modular exponentiation algorithm 4](#_Toc355598471)

[1.4 MonPro Algorithm 7](#_Toc355598472)

[2 HARDWARE IMPLEMENTATION 11](#_Toc355598473)

[2.1 Platform and Device 11](#_Toc355598474)

[2.2 I/O Pin Assignment 12](#_Toc355598475)

[2.3 Memory (M4K) Usage 12](#_Toc355598477)

[2.4 All Verilog Files 13](#_Toc355598478)

[2.4 Module Description 14](#_Toc355598479)

[2.5 Compilation Summary 15](#_Toc355598480)

[3. ANALYSIS 16](#_Toc355598481)

[3.1 Resource Usage 16](#_Toc355598482)

[3.2 Time Usage 16](#_Toc355598483)

[4. FUTURE WORK 17](#_Toc355598484)

[5. BIBLIOGRAPHY 17](#_Toc355598485)

# VERSION CONTROL HISTORY

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Date | Author of Changes | Approver | Description of Changes |
| 0.1 | 24 April 2013 | Balasubramaniam Muthuvelu | Çetin Kaya Koç | Initial Draft Document |
| 0.2 | 4 May 2013 | Qin Zhou | Çetin Kaya Koç | Refine Document |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# INTRODUCTION

## RSA Algorithm

The RSA algorithm, invented by Rivest, Shamir, and Adleman, is one of the simplest public-key cryptosystems. The parameters are , and , , and . The modulus is the product of two distinct large random primes and such that . The public exponent is a number in the range such that

,

where is the Euler's totient function of , given by

.

The private exponent of is obtained by inverting modulo i.e.

,

using the extended Euclidean algorithm. The encryption operation is performed by computing

,

where is the plaintext such that . The number is the ciphertext from which the plaintext can be computed using

.

The RSA algorithm can be used to send encrypted messages and to produce digital signatures for electronic documents. It provides a procedure for signing a digital document, and verifying whether the signature is indeed authentic. The signing of a digital document is somewhat different from signing a paper document, where the same signature is being produced for all paper documents. A digital signature cannot be a constant; it is a function of the digital document for which it was produced. After the signature (which is just another piece of digital data) of a digital document is obtained, it is attached to the document for anyone wishing to verify the authenticity of the document and the signature. [1]

## RSA Implementation

There are many optimized software and hardware techniques that are available currently, using which RSA algorithm can be implemented. In our case, we are trying to implement a hardware algorithm which would be optimal with respect to time and which can encrypt and decrypt a given message when the modulus size is 4096 bits. We have implemented the algorithm using a Montgomery-transformed exponentiation algorithm. The details of which are provided in the next section.

## Montgomery transform based modular exponentiation algorithm

The 4096-bit numbers are organized as bits, where is the number of words and is the word length, such that . We will take and bits in our implementation.

The input variables are as provided below:

1. Input message *,* signed integer of size 4096 bits. Generally, the messages would be less than 4096 bits in size. In cases where the size is greater than 4096 bits we divide the message into blocks of 4096 bits. We use two’s complement notation to store this signed integer.
2. The exponent which is also of size 4096 bits. Exponent is -bit unsigned integer where . Inputs and are our primary inputs.
3. The modulus operator which is of size 4096 bits. is a 4096 bit signed integer. Also, is assumed to be an odd number such that the least significant bit and the most significant bit of message would be when due to properties of two’s complement notation.
4. The input which is pre-computed from and such that
5. The Montgomery constant defined as
6. The constant defined as

or

The inputs is one word in size (32 bits) while and are signed integers of size 4096 bits. The output of the process is the cipher text of size 4096 bits.

The 1-bit ModExp Algorithm is the core of our implementation. The flowchart described below:

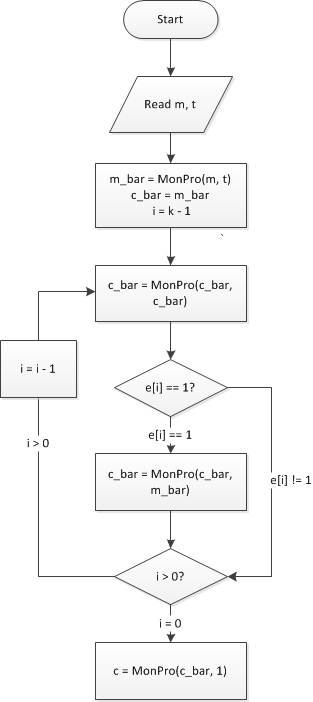


Figure 1: Workflow of 1-bit ModExp Algorithm

The pseudo code is below:

*Find the index of the leftmost in*

*For downto*

*If then*

## MonPro Algorithm

The MonPro algorithm implements the equation: . It takes two inputs: and and computes the output such that each variable holds an -word (total bit) signed integer. In this case and are 128 words of 32 bit each. The initial value of is initialized to zero. Hence,

The secondary inputs ,, and are also assumed to be available. We have

,

where has 128 words of 32 bit each and has 1 word of 32 bit.

The steps of the MonPro algorithm are as explained below:

*Step 1:* We take the LSW (least significant word) of , namely and multiply by the 128-word , and add it to the 128 word partial product (which is now all zero) to obtain the (129)-word temporary result as .

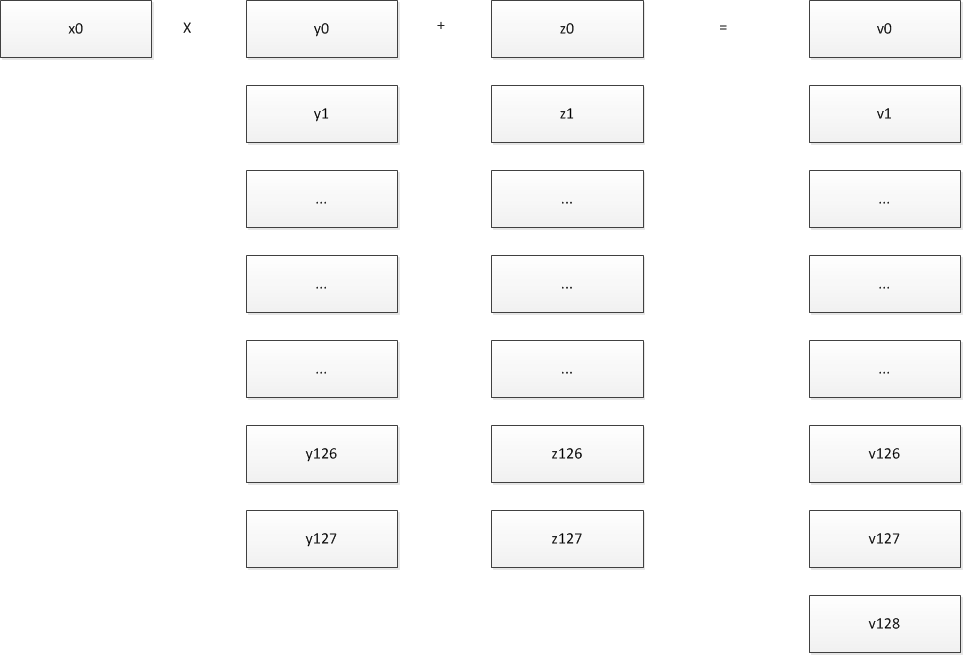


Figure 2: MonPro Step 1:

*Step 1a:* The computation in Step 1 is accomplished using a Multiply-Add block that multiplies two 1-word numbers ( and ), adds the previous higher word (), and adds another 1-word number (), producing a 2-word number (,); the lower word () is assigned to value (), while the higher word () is used for next Multiply-Add step, as follows:

…

Where the initial value of .

*Step 2*: We then take the LSW of , namely and multiply by the 1-word modulo and obtain the 1-word integer as .

C:\Users\Jason\Desktop\monpro2.png

Figure 3: MonPro Step 2:

*Step 3*: We take the 1-word and multiply by the 128-word , and add it to the 129 word temporary value (from step 1) to obtain the new partial product which is the 129 word .

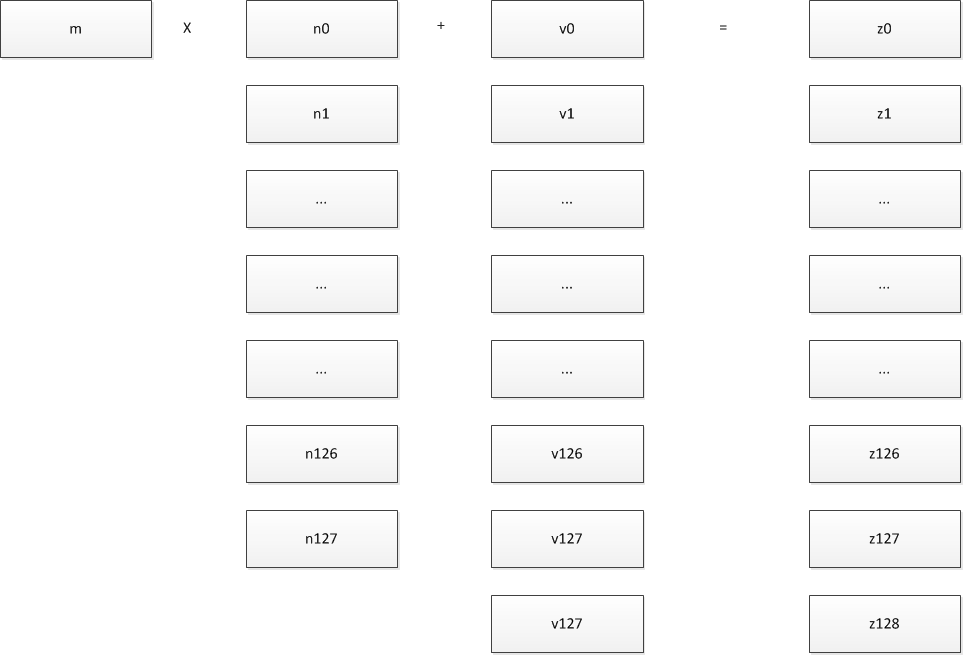


Figure 4: MonPro Step 3:

*Step 3a*: The computation in step 3 is accomplished using a Multiply-Add block that multiplies two 1-word numbers ( and ), adds the previous higher word (), adds another 1-word number (), producing a 2-word number (,); assigning to , while keeping the higher word () for the next Multiply-Add step, as follows:

…

Such that the initial value .

*Step 4*: The resulting partial product has its LSW set to zero, due to the Montgomery property, and therefore, we shift to obtain the new 128 word partial product.

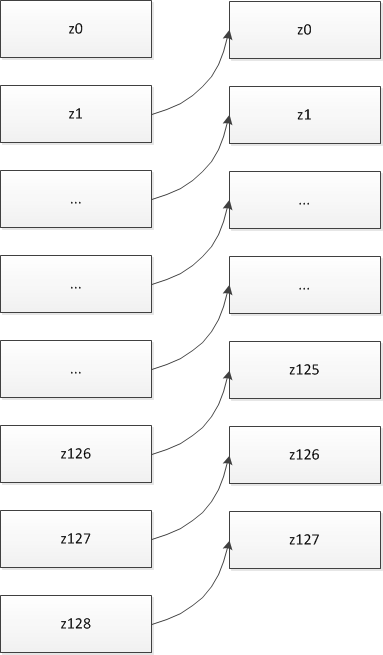


Figure 5: MonPro Step 4

*Step 5*: In the next step the next word of , namely is taken and multiplied with the 128 word and added to the partial product to obtain the new temporary result .



Figure 6: MonPro Step 5:

*Step 6*: This is followed up by computing the new m value

and then the multiplication of by , and then the addition of the result to to obtain the new 129 word partial product and shifting the value of by one word (since ) to obtain the new 128 word value.

*Step 7*: Proceeding in this way, we multiply all values by the multiplicand and reduce it modulo for . [2]

# HARDWARE IMPLEMENTATION

## Platform and Device

Altera Cyclone II EP2C50 is our target device (EP2C50F484C6 is the exact device number). To generate the FPGA image for this device, we use Quartus II 12.1sp1 Web Edition to do Verilog programming and compiling. After that we use ModelSim-Altera 10.1b (Quartus II 12.1sp1) Starter Edition to simulate and verify the code logic.

Altera Cyclone II EP2C50 has 50,528 logic elements, 129 M4K RAM blocks (4Kbits plus 512 parity bits per block), 594,432 Total RAM bits, 86 embedded multipliers, 4 PLLs, and 450 maximum user I/O pins. The clock frequency is 100 MHz. Within this limitation, we successfully build an FPGA image that can compute 4096-bit modular exponentiation. [3]

## I/O Pin Assignment

For ModExp module, we have 2 input pins (reset and clock), and 32 output pins (32-bit result value). All of them are in I/O Bank 1. The I/O pin assignment is as follow:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| I/O Name | Pin Number | Type | I/O Name | Pin Number | Type |
| clk | M1 | Input | res\_out[15] | P6 | Output |
| reset | M2 | Input | res\_out[16] | U3 | Output |
| res\_out[0] | R1 | Output | res\_out[17] | T3 | Output |
| res\_out[1] | W2 | Output | res\_out[18] | R2 | Output |
| res\_out[2] | W1 | Output | res\_out[19] | V1 | Output |
| res\_out[3] | N5 | Output | res\_out[20] | T1 | Output |
| res\_out[4] | V2 | Output | res\_out[21] | R4 | Output |
| res\_out[5] | U1 | Output | res\_out[22] | P2 | Output |
| res\_out[6] | U2 | Output | res\_out[23] | P4 | Output |
| res\_out[7] | N1 | Output | res\_out[24] | P3 | Output |
| res\_out[8] | M6 | Output | res\_out[25] | P5 | Output |
| res\_out[9] | P1 | Output | res\_out[26] | N3 | Output |
| res\_out[10] | N4 | Output | res\_out[27] | R6 | Output |
| res\_out[11] | Y1 | Output | res\_out[28] | R5 | Output |
| res\_out[12] | N6 | Output | res\_out[29] | M5 | Output |
| res\_out[13] | T5 | Output | res\_out[30] | N2 | Output |
| res\_out[14] | T6 | Output | res\_out[31] | T2 | Output |

Table 1: I/O Pin Assignment



## 2.3 Memory (M4K) Usage

All data input and output are actually transferred through memory blocks, which are called M4Ks in Quartus II. I/O pins are triggers rather than data transfer ports.

For Memory blocks we can use In-System Memory Content Editor to view and update memories and constants with the JTAG port connection. [4]

The memory block usage is as follow:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Memory Block Instance Name | Type | Mode | Clock Mode | Memory Initialization Filename | Size (bits) | As Input or Output? |
| m | M4K | True Dual Port | Dual Clocks | mMem.mif | 4096 | Input |
| e | M4K | True Dual Port | Dual Clocks | eMem.mif | 4096 | Input |
| n | M4K | True Dual Port | Dual Clocks | nMem.mif | 4096 | Input |
| r | M4K | True Dual Port | Dual Clocks | rMem.mif | 4096 | Input |
| t | M4K | True Dual Port | Dual Clocks | tMem.mif | 4096 | Input |
| nprime0 | M4K | True Dual Port | Dual Clocks | nprime0Mem.mif | 32 | Input |
| res | M4K | True Dual Port | Dual Clocks | None | 4096 | Output |

Table 2: Memory Block Usage

## 2.4 All Verilog Files

1. \_parameter.v

Defines the word size (DATA\_WIDTH), the address size (ADDR\_WIDTH) and number of words (TOTAL\_ADDR).

1. memory files

Defines each M4K block and memory initialization file.

Files are: m\_mem.v, e\_mem.v, n\_mem.v, nprime0\_mem.v, r\_mem.v, t\_mem.v, res\_mem.v;

7 files in total.

1. mul\_add.v

Multiplication and addition module which computes ; all 5 elements are WIDTH bits.

1. ModExp.v

Core module which reads in , computes , and output c to the result memory.

1. ModExp\_tb.v

Test bench file which tests ModExp module.

## 2.4 Module Description

1. mul\_add module

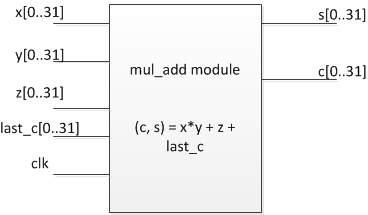


Figure 7: Mul\_add Module Diagram

Mul\_add module is the simplest component in our code, which multiplies and adds four elements and output two words result.

1. MonPro module

MonPro implements this equation: . In our implementation, MonPro module is a finite state machine which has 8 different states. We add extra states to steps of Section 1.4 because there exists that need minor steps in finite state machine.

State 0 computes , which corresponds to step 1a in Section 1.4;

State 1 computes the highest word of : ;

State 2 computes , which corresponds to step 2 in Section 1.4;

State 3 computes , which corresponds to step 3a in Section 1.4;

State 4 computes the second highest word of : ;

State 5 computes the highest word of : ;

State 6 dumps the result;

State 7 is the terminate state.

1. ModExp module

ModExp follows Figure 1: Workflow of 1-bit ModExp Algorithm, and explicitly uses MonPro module code. It is a finite state machine which has 8 states.

State 00 initializes all registers;

State 0 computes and ;

State 1 initializes the leftmost nonzero bit of , which is k;

State 2 implements the for loop: i = k downto 0;

State 3 computes , which will be used by State 2 when ;

State 4 computes ;

State 5 dumps the result to result memory;

State 6 is the terminate state.

## 2.5 Compilation Summary

|  |  |
| --- | --- |
| Entry Name | Entry Content |
| Quartus II 32-bit Version | 12.1 Build 243 01/31/2013 SP 1 SJ Web Edition |
| Top-level Entity Name | ModExp |
| Family | Cyclone II |
| Device | EP2C50F484C6 |
| Timing Models | Final |
| Total Logic Elements | 40,343 / 50,528 (80 %) |
| Total Registers | 25692 |
| Total Pins | 34 / 294 (12 %) |
| Total Virtual Pins | 0 |
| Total Memory Bits | 28,704 / 594,432 (5 %) |
| Embedded Multiplier 9-bit Elements | 8 / 172 (5 %) |
| Total PLLs | 0 / 4 (0 %) |
| FPGA POF Image Size | 2,049 KB |
| FPGA SOF Image Size | 1,199 KB |

Table 3 Compilation Summary

# 3. ANALYSIS

## Resource Usage

1. Logic Element Usage

We use 40343 logic elements which occupies 80% of all logic elements. However, combinational logic elements with no register are 14651 which is only 29% of all logic elements. Other element usage is because the data is too large. We have six 4096-bit integers and one 32-bit integer as input and one 4096-bit integer as output. Thus 8209 logic elements are register only, and 17483 logic elements are combinational with a register.

1. M4K Usage

If we do not configure JTAG In-System Memory Content Editor, we use eight M4K, which are exactly our seven inputs and one output.

If JTAG In-System Memory Content Editor configured, 14 M4K blocks are needed, since JTAG needs 6 extra M4K blocks as connections.

1. Maximum Clock Frequency

Using TimeQuest Timing Analyzer Wizard [5], Quartus II sugguests that the maximum frequency .

## Time Usage

The maximum clock frequency is . Then we can estimate the overall time usage of the ModExp module. Suppose word size is , word number of a 4096-bit integer is .

1. mul\_add module needs 2 cycles: one cycle to dump input registers into mul\_add module; another cycle to dump output registers to caller module. Thus we need 2 cycles to compute -bit mul\_add operation.
2. MonPro module follows the pseudo-code below

*(1)*

*(2)*

*(3)*

*(4)*

*output (5)*

Step 2 needs cycles; Step 3 needs 2 cycles; Step 4 needs cycles; Step 1 is a for loop, which means Step 1-4 needs cycles; Step 5 needs cycles;

Overall, MonPro needs cycles.

1. ModExp module follows this pseudo-code below

*Initialize 7 registers; (1)*

*Find the index of the leftmost in (2)*

*(3)*

*(4)*

*For downto (5)*

*(6)*

*If then (7)*

*(8)*

The time usage of ModExp is closely related to the value of . Suppose the length of is .

Step 1 needs cycles; Step 2 needs cycles; Step 3 needs cycles; Step 4 needs cycles; Step 6 needs cycles; Suppose half of bits of are 1, then Step 7 needs cycles; Step 5-7 needs cycles; Step 8 needs cycles.

Overall, ModExp needs

steps.

Suppose ,

The estimated total time usage is

# FUTURE WORK

Build side-channel attacks on this 4096-bit ModExp FPGA board.

# BIBLIOGRAPHY

[1] C. K. Koc. RSA Hardware Implementation

[2] C. K. Koc. ModExp 4096 Definitions & Algorithms

[3] Altera. Corp. Cyclone II Device Handbook, Volume 1

[4] Altera. Corp. In-System Modification of Memory and Constants

[5] Altera. Corp. Using Timing Analysis in the Quartus II Software