**INTERFACE CONTROL DOCUMENT**

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

**APPLIED CRYPTOLAB UCSB**

**DEPARTMENT OF COMPUTER SCIENCE**

**UNIVERSITY OF CALIFORNIA, SANTA BARBARA**

***Version:*** 2.0

***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 4](#_Toc361470432)

[1. INTRODUCTION 5](#_Toc361470433)

[1.1 RSA Algorithm 5](#_Toc361470434)

[1.2 RSA Implementation 6](#_Toc361470435)

[1.3 Montgomery transform based modular exponentiation algorithm 6](#_Toc361470436)

[1.4 MonPro Algorithm 8](#_Toc361470437)

[1.5 Secondary Input Parameters Computation 12](#_Toc361470438)

[1.5.1 Compute 13](#_Toc361470439)

[1.5.2 Compute 13](#_Toc361470440)

[1.5.3 Compute 13](#_Toc361470441)

[2 HARDWARE IMPLEMENTATION 13](#_Toc361470442)

[2.1 Platform and Device 13](#_Toc361470443)

[2.2 I/O Design and Control Signals 14](#_Toc361470444)

[2.3 I/O Pin Assignment 15](#_Toc361470445)

[2.4 Memory (M4K) Usage 15](#_Toc361470447)

[2.5 All Verilog Files 15](#_Toc361470448)

[2.6 Module Description 16](#_Toc361470449)

[2.7 Compilation Summary 17](#_Toc361470450)

[3. ANALYSIS 18](#_Toc361470451)

[3.1 Resource Usage 18](#_Toc361470452)

[3.2 Time Usage 19](#_Toc361470453)

[4. FUTURE WORK 20](#_Toc361470454)

[5. BIBLIOGRAPHY 21](#_Toc361470455)

[6. APPENDIX 21](#_Toc361470456)

[6.1 Appendix A: Pin Assignments 21](#_Toc361470457)

[6.1.1 Input Pins 21](#_Toc361470458)

[6.1.2 Output Pins 23](#_Toc361470459)

[6.2 Appendix B: Simulation Example 25](#_Toc361470460)

[6.2.1 Summary 25](#_Toc361470461)

[6.2.2 Testing Case 25](#_Toc361470462)

# 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 |
| 1.0 | 6 May 2013 | Çetin Kaya Koç | Çetin Kaya Koç | Refine Document |
| 2.0 | 7 Jul 2013 | Qin Zhou | Cetin Kaya Koç | IO and control signal features added; width upgraded from 32bits to 64bits |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 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 (64 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:

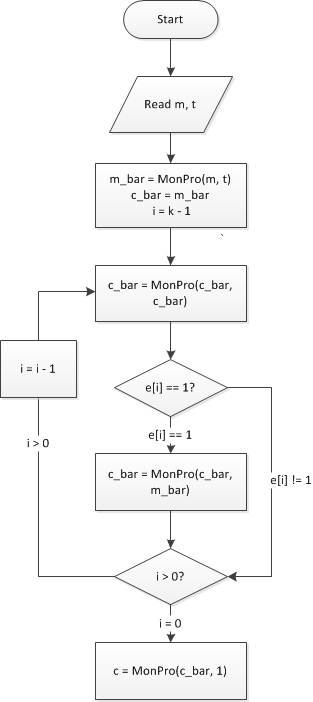


Figure 1: Workflow of 1-bit ModExp Algorithm

The pseudo code is below:

*Find the index of the leftmost in*

*For downto*

*If then*

Name it 1-bit ModExp Algorithm because we check 1 bit of at a time.

## 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 64 words of 64 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 64 words of 64 bit each and has 1 word of 64 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 64-word , and add it to the 64 word partial product (which is now all zero) to obtain the (65)-word temporary result as .

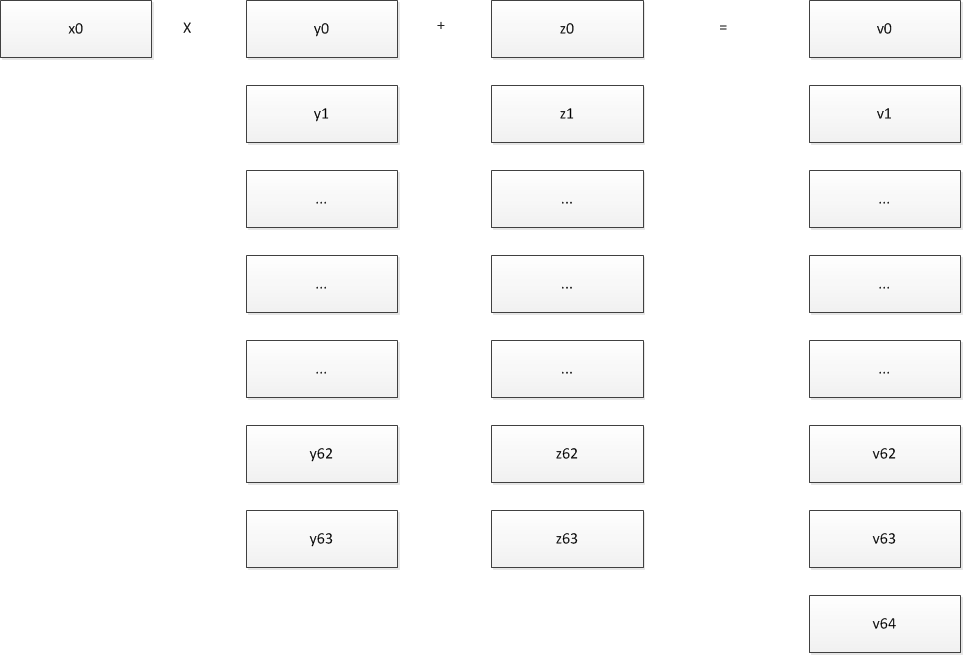


Figure 2: MonPro Step 1:

*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 64-word , and add it to the 65 word temporary value (from step 1) to obtain the new partial product which is the 65 word .

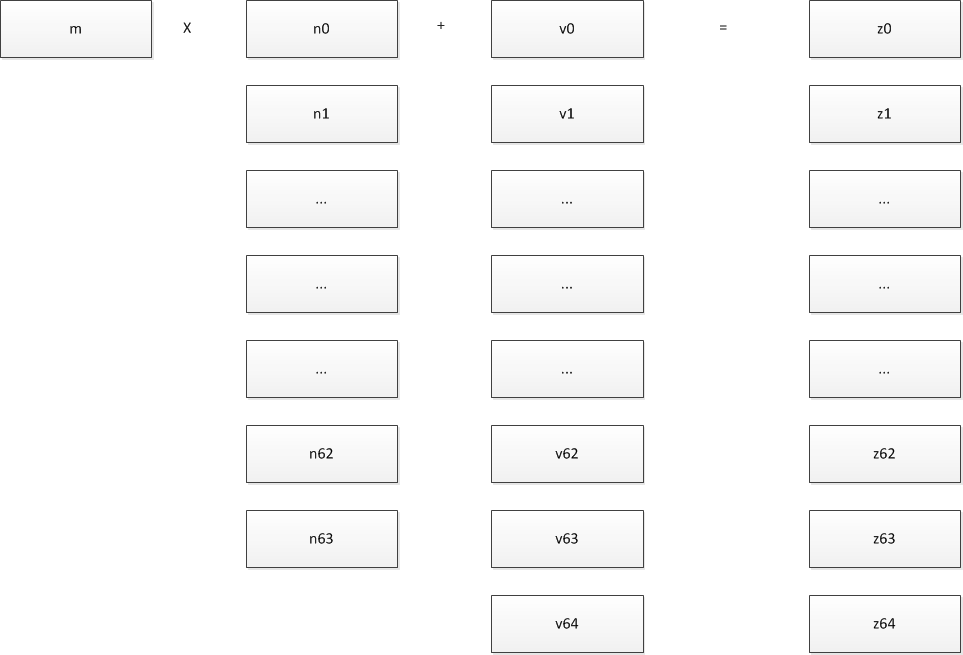


Figure 4: MonPro Step 3:

*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 64 word partial product.

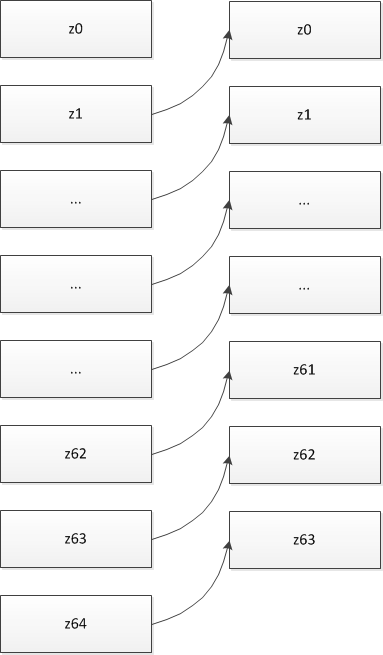


Figure 5: MonPro Step 4

*Step 5*: In the next step the next word of , namely is taken and multiplied with the 64 word and added to the partial product to obtain the new temporary result .

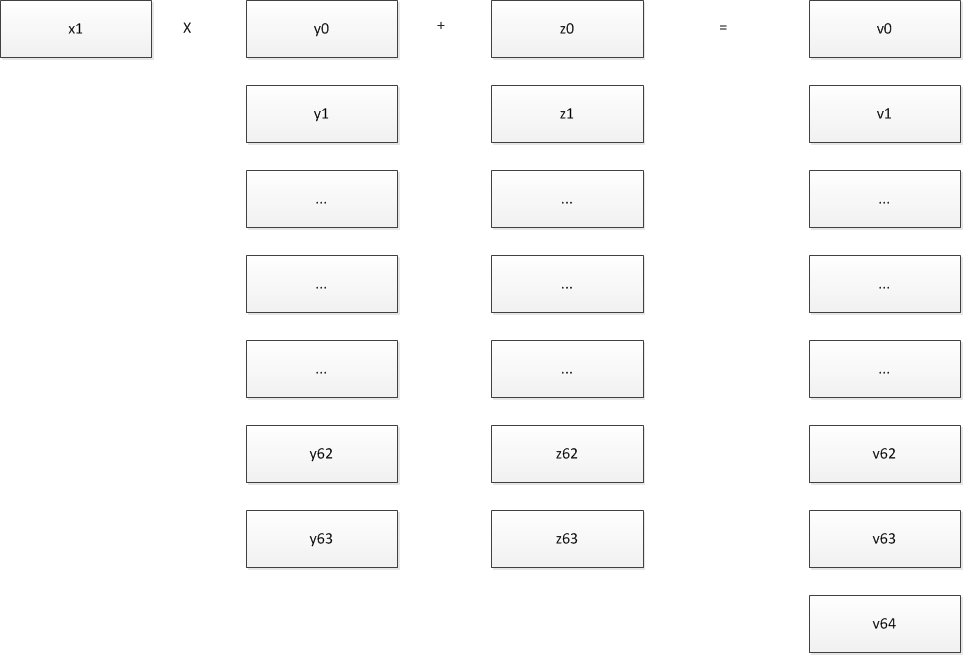


Figure 6: MonPro Step 5:

*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 65 word partial product and shifting the value of by one word (since ) to obtain the new 64 word value.

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

## Secondary Input Parameters Computation

In Section 1.3 we illustrate all parameters for RSA computation, among which , , are derived from so that we name them secondary input parameters.

In order to compute secondary input parameters inside the FPGA, several algorithms were developed. We then introduce the algorithm to compute , and .

### Compute

is a one-word integer which is equal to . In order to compute , we use the algorithm given below which computes for a given odd .

***function*** *ModInverse(, ) { is odd }*

*for to*

*if*

*then*

*else*

*return*

Compute ModInverse(, ), negate the result then we get . [1]

### Compute

equals to , which is the 2’s complement of in bits.

### Compute

equals to , which can be computed by aligning the most significant (nonzero) bit of to the ()th position, and then performing successive multi-precision (-word) subtractions until all most significant positions in the resulting number until to the position () are zero. [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 Design and Control Signals

Because the parameters are 4096 bits which are very large numbers, in order to fit our RSA implementation to the target device successfully, we finalized our I/O design in this way:

The parameters and are read from input pins. There are pins for each parameter (64 \* 2 = 128 pins). We input bits to and each cycle. After cycles bits and are read in.

, , and are initialized by memory initialization files (M4K mif files), which means they are hard coded in the FPGA. The reason is if we input by input pins and change then , and should all be changed but it is impossible to fit the computation of into the FPGA due to lack of logic elements.

We also have several control signals to control the process of computation. Reset button resets the FPGA to initial state when pressed; StartInput button tells FPGA to start reading and through which we can synchronize the input process; StartCompute button tells FPGA to start the computation, which should be pressed after the input process is finished; GetResult button tells FPGA to output the result.

In order to know the exact state of current computation, we set 4 pins to output the lower state of the FPGA, and set 5 pins to output the upper state of the FPGA. The upper state is the larger state such as INIT\_STATE, WAIT\_COMPUTE, and which may include multiple lower states. Section 2.4 describes the states within all modules.

Finally we have pins for result output using the same way as the input.

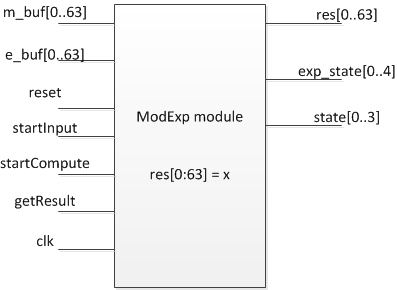


Figure 7: ModExp Module’s Input and Output

## I/O Pin Assignment

For ModExp module, we have 133 input pins (clock, reset, startInput, startCompute, getResult, m\_buf[0:63] and e\_buf[0:63]), and 73 output pins (state[0:3], exp\_state[0:4] and res\_out[0:63]). The I/O pin assignment is in Appendix A.



## 2.4 Memory (M4K) Usage

, , and are transferred through memory blocks, which are called M4Ks in Quartus II.

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 | Memory Initialization Filename | Size (bits) | As Input or Output? |
| n | M4K | SINGLE\_PORT | nMem.mif | 4096 | Input |
| r | M4K | SINGLE\_PORT | rMem.mif | 4096 | Input |
| t | M4K | SINGLE\_PORT | tMem.mif | 4096 | Input |
| nprime0 | M4K | SINGLE\_PORT | nprime0Mem.mif | 64 | Input |

Table 1: Memory Block Usage

## 2.5 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: n\_mem.v, nprime0\_mem.v, r\_mem.v, t\_mem.v;

4 files in total.

1. mul\_add.v

Multiplication and addition module which computes ; all 5 elements are DATA\_WIDTH (64) bits.

1. ModExp.v

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

## 2.6 Module Description

1. mul\_add module

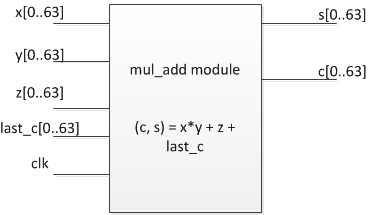


Figure 7: Mul\_add Module Diagram

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

Basically 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.

INIT\_STATE: the beginning state of the module. The reset button will return the FPGA to this state. Change to LOAD\_M\_E state when startInput button is pressed;

LOAD\_M\_E: load , from input pins, go to LOAD\_N;

LOAD\_N: initialize , , and from M4K memory, go to WAIT\_COMPUTE;

WAIT\_COMPUTE: wait the startCompute button, if pressed go to CALC\_M\_BAR;

CALC\_M\_BAR: computes and , go to GET\_K\_E;

GET\_K\_E: initializes the leftmost nonzero bit of , which is k;

BIGLOOP: implements the for loop: i = k downto 0; then go to CALC\_C\_BAR\_1;

CALC\_C\_BAR\_M\_BAR: computes , which will be used by BIGLOOP when ;

CALC\_C\_BAR\_1: computes ; go to COMPLETE;

COMPLETE: wait the getResult button, if pressed go to OUTPUT\_RESULT;

OUTPUT\_RESULT: dumps the result to result memory; go to the TERMINAL state;

TERMINAL: the end state.

## 2.7 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 | 42,341 / 50,528 (80 %) |
| Total Registers | 27962 |
| Total Pins | 206 / 294 (12 %) |
| Total Virtual Pins | 0 |
| Total Memory Bits | 16,448 / 594,432 (3 %) |
| Embedded Multiplier 9-bit Elements | 32 / 172 (19 %) |
| Total PLLs | 0 / 4 (0 %) |
| FPGA POF Image Size | 2,049 KB |
| FPGA SOF Image Size | 1,201 KB |

Table 3 Compilation Summary

# 3. ANALYSIS

## Resource Usage

1. Logic Element Usage

There are three types of logic elements: the combinational with no register logic elements are 14,379; Register only logic elements are 8,884; and Combinational with a register logic elements are 19,078.

1. M4K Usage

If we do not configure JTAG In-System Memory Content Editor, we use four M4Ks, which are exactly our four inputs.

If JTAG In-System Memory Content Editor configured, 10 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 suggests 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 , ; (1)*

*Initialize , , , ; (2)*

*Find the index of the leftmost in (3)*

*(4)*

*(5)*

*For downto (6)*

*(7)*

*If then (8)*

*(9)*

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

Assume the length of is and the Hamming weight of e is .

Step 1 and 2 needs cycles;   
Step 2 needs cycles;   
Step 3 needs cycles;   
Step 4 needs cycles;   
Step 6 needs ) cycles;

Step 7 needs cycles;

Step 5-7 needs cycles;   
Step 8 needs cycles.

Overall, ModExp needs

cycles.

Cycle length is the inverse of the frequency . The total time in seconds is given as .

Note that for a fixed size modulus of size 4096 bits, is given as 64.

The table below enumerates the total time in terms the parameters, : The frequency, : The length of the exponent, and : The Hamming weight of the exponent.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | L=512, H=256 | L=1024, H=256 | L=1024, H=512 | L=2048, H=512 | L=2048, H=1024 |
| F = 25 MHz | 0.511 | 0.851 | 1.021 | 1.699 | 2.039 |
| F = 50 MHz | 0.256 | 0.425 | 0.510 | 0.850 | 1.019 |
| F = 75 MHz | 0.170 | 0.284 | 0.340 | 0.566 | 0.680 |

# FUTURE WORK

We will build side-channel counter measures on this 4096-bit ModExp module.

# BIBLIOGRAPHY

[1] Ç. K. Koç. *High-Speed RSA Implementation*. TR 201, RSA Labs, 73 pages, November 1994.

[2] Ç. K. Koç. 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.*

# APPENDIX

## Appendix A: Pin Assignments

### Input Pins

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | **Pin #** | **I/O Bank** | **Name** | **Pin #** | **I/O Bank** |
| clk | M1 | 1 | m\_buf[0] | E8 | 3 |
| e\_buf[0] | D14 | 4 | m\_buf[10] | M22 | 6 |
| e\_buf[10] | E9 | 3 | m\_buf[11] | J5 | 2 |
| e\_buf[11] | D9 | 3 | m\_buf[12] | H1 | 2 |
| e\_buf[12] | A14 | 4 | m\_buf[13] | J2 | 2 |
| e\_buf[13] | C13 | 4 | m\_buf[14] | H2 | 2 |
| e\_buf[14] | A11 | 3 | m\_buf[15] | W7 | 8 |
| e\_buf[15] | F10 | 3 | m\_buf[16] | J3 | 2 |
| e\_buf[16] | A3 | 3 | m\_buf[17] | G1 | 2 |
| e\_buf[17] | A5 | 3 | m\_buf[18] | J4 | 2 |
| e\_buf[18] | B8 | 3 | m\_buf[19] | G2 | 2 |
| e\_buf[19] | D6 | 2 | m\_buf[1] | H3 | 2 |
| e\_buf[1] | B16 | 4 | m\_buf[20] | AA7 | 8 |
| e\_buf[20] | B3 | 3 | m\_buf[21] | Y7 | 8 |
| e\_buf[21] | E7 | 3 | m\_buf[22] | J6 | 2 |
| e\_buf[22] | B5 | 3 | m\_buf[23] | W5 | 1 |
| e\_buf[23] | D3 | 2 | m\_buf[24] | Y2 | 1 |
| e\_buf[24] | B4 | 3 | m\_buf[25] | AB3 | 8 |
| e\_buf[25] | A6 | 3 | m\_buf[26] | U4 | 1 |
| e\_buf[26] | C7 | 3 | m\_buf[27] | W3 | 1 |
| e\_buf[27] | E3 | 2 | m\_buf[28] | Y6 | 8 |
| e\_buf[28] | A8 | 3 | m\_buf[29] | Y3 | 1 |
| e\_buf[29] | B6 | 3 | m\_buf[2] | B10 | 3 |
| e\_buf[2] | E11 | 3 | m\_buf[30] | Y5 | 8 |
| e\_buf[30] | A7 | 3 | m\_buf[31] | AB4 | 8 |
| e\_buf[31] | D5 | 2 | m\_buf[32] | AA3 | 8 |
| e\_buf[32] | B9 | 3 | m\_buf[33] | W4 | 1 |
| e\_buf[33] | B15 | 4 | m\_buf[34] | Y4 | 1 |
| e\_buf[34] | B13 | 4 | m\_buf[35] | U8 | 8 |
| e\_buf[35] | B14 | 4 | m\_buf[36] | V4 | 1 |
| e\_buf[36] | A4 | 3 | m\_buf[37] | AB5 | 8 |
| e\_buf[37] | F8 | 3 | m\_buf[38] | AB6 | 8 |
| e\_buf[38] | A9 | 3 | m\_buf[39] | AA4 | 8 |
| e\_buf[39] | D7 | 3 | m\_buf[3] | F2 | 2 |
| e\_buf[3] | A16 | 4 | m\_buf[40] | AA6 | 8 |
| e\_buf[40] | C10 | 3 | m\_buf[41] | B12 | 4 |
| e\_buf[41] | A15 | 4 | m\_buf[42] | A12 | 4 |
| e\_buf[42] | A13 | 4 | m\_buf[43] | AA5 | 8 |
| e\_buf[43] | A10 | 3 | m\_buf[44] | AB7 | 8 |
| e\_buf[44] | F11 | 3 | m\_buf[45] | W12 | 7 |
| e\_buf[45] | D11 | 3 | m\_buf[46] | V12 | 7 |
| e\_buf[46] | B11 | 3 | m\_buf[47] | W9 | 8 |
| e\_buf[47] | F12 | 4 | m\_buf[48] | Y9 | 8 |
| e\_buf[48] | H6 | 2 | m\_buf[49] | U9 | 8 |
| e\_buf[49] | E1 | 2 | m\_buf[4] | V8 | 8 |
| e\_buf[4] | F14 | 4 | m\_buf[50] | W11 | 8 |
| e\_buf[50] | E4 | 2 | m\_buf[51] | W8 | 8 |
| e\_buf[51] | F3 | 2 | m\_buf[52] | AA10 | 8 |
| e\_buf[52] | C1 | 2 | m\_buf[53] | V9 | 8 |
| e\_buf[53] | E2 | 2 | m\_buf[54] | AB9 | 8 |
| e\_buf[54] | G6 | 2 | m\_buf[55] | Y10 | 8 |
| e\_buf[55] | H5 | 2 | m\_buf[56] | AA12 | 7 |
| e\_buf[56] | C2 | 2 | m\_buf[57] | AB12 | 7 |
| e\_buf[57] | D1 | 2 | m\_buf[58] | AA9 | 8 |
| e\_buf[58] | G5 | 2 | m\_buf[59] | V11 | 8 |
| e\_buf[59] | G3 | 2 | m\_buf[5] | AA8 | 8 |
| e\_buf[5] | C9 | 3 | m\_buf[60] | AB8 | 8 |
| e\_buf[60] | D2 | 2 | m\_buf[61] | AB11 | 8 |
| e\_buf[61] | H4 | 2 | m\_buf[62] | U10 | 8 |
| e\_buf[62] | F4 | 2 | m\_buf[63] | AB10 | 8 |
| e\_buf[63] | F1 | 2 | m\_buf[6] | J1 | 2 |
| e\_buf[6] | B7 | 3 | m\_buf[7] | AB13 | 7 |
| e\_buf[7] | D4 | 2 | m\_buf[8] | AA11 | 8 |
| e\_buf[8] | D8 | 3 | m\_buf[9] | M21 | 6 |
| e\_buf[9] | F9 | 3 | reset | M2 | 1 |
| getResult | F13 | 4 | startCompute | L21 | 5 |
|  |  |  | startInput | L22 | 5 |

### 6.1.2 Output Pins

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | **Pin #** | **I/O Bank** | **Name** | **Pin #** | **I/O Bank** |
| res\_out[0] | R1 | 1 | res\_out[38] | AA16 | 7 |
| res\_out[10] | N4 | 1 | res\_out[39] | V14 | 7 |
| res\_out[11] | Y1 | 1 | res\_out[3] | N5 | 1 |
| res\_out[12] | N6 | 1 | res\_out[40] | M18 | 6 |
| res\_out[13] | T5 | 1 | res\_out[41] | Y16 | 7 |
| res\_out[14] | T6 | 1 | res\_out[42] | AA14 | 7 |
| res\_out[15] | P6 | 1 | res\_out[43] | AB17 | 7 |
| res\_out[16] | U3 | 1 | res\_out[44] | N22 | 6 |
| res\_out[17] | T3 | 1 | res\_out[45] | Y14 | 7 |
| res\_out[18] | R2 | 1 | res\_out[46] | AA15 | 7 |
| res\_out[19] | V1 | 1 | res\_out[47] | W15 | 7 |
| res\_out[1] | W2 | 1 | res\_out[48] | U14 | 7 |
| res\_out[20] | T1 | 1 | res\_out[49] | AA17 | 7 |
| res\_out[21] | R4 | 1 | res\_out[4] | V2 | 1 |
| res\_out[22] | P2 | 1 | res\_out[50] | AB16 | 7 |
| res\_out[23] | P4 | 1 | res\_out[51] | W21 | 6 |
| res\_out[24] | P3 | 1 | res\_out[52] | U13 | 7 |
| res\_out[25] | P5 | 1 | res\_out[53] | W22 | 6 |
| res\_out[26] | N3 | 1 | res\_out[54] | AA18 | 7 |
| res\_out[27] | R6 | 1 | res\_out[55] | P18 | 6 |
| res\_out[28] | R5 | 1 | res\_out[56] | AB15 | 7 |
| res\_out[29] | M5 | 1 | res\_out[57] | N21 | 6 |
| res\_out[2] | W1 | 1 | res\_out[58] | AA13 | 7 |
| res\_out[30] | N2 | 1 | res\_out[59] | M19 | 6 |
| res\_out[31] | T2 | 1 | res\_out[5] | U1 | 1 |
| res\_out[32] | P19 | 6 | res\_out[60] | AB18 | 7 |
| res\_out[33] | L19 | 5 | res\_out[61] | R20 | 6 |
| res\_out[34] | AB14 | 7 | res\_out[62] | P20 | 6 |
| res\_out[35] | L18 | 5 | res\_out[63] | U20 | 6 |
| res\_out[36] | Y13 | 7 | res\_out[6] | U2 | 1 |
| res\_out[37] | W14 | 7 | res\_out[7] | N1 | 1 |
| exp\_state[0] | V22 | 6 | res\_out[8] | M6 | 1 |
| exp\_state[1] | U18 | 6 | res\_out[9] | P1 | 1 |
| exp\_state[2] | E14 | 4 | state[0] | P22 | 6 |
| exp\_state[3] | W18 | 6 | state[1] | K22 | 5 |
| exp\_state[4] | Y17 | 7 | state[2] | K18 | 5 |
|  |  |  | state[3] | E15 | 4 |

## Appendix B: Simulation Example

### 6.2.1 Summary

In this appendix we provide a complete example to verify the implementation: first we create , and some control signals in test bench file, and create , , , in memory initialization file; second we simulate the code ModExp.v in ModelSim-Altera.

In this simulation waveform we can see that the logic is correct.

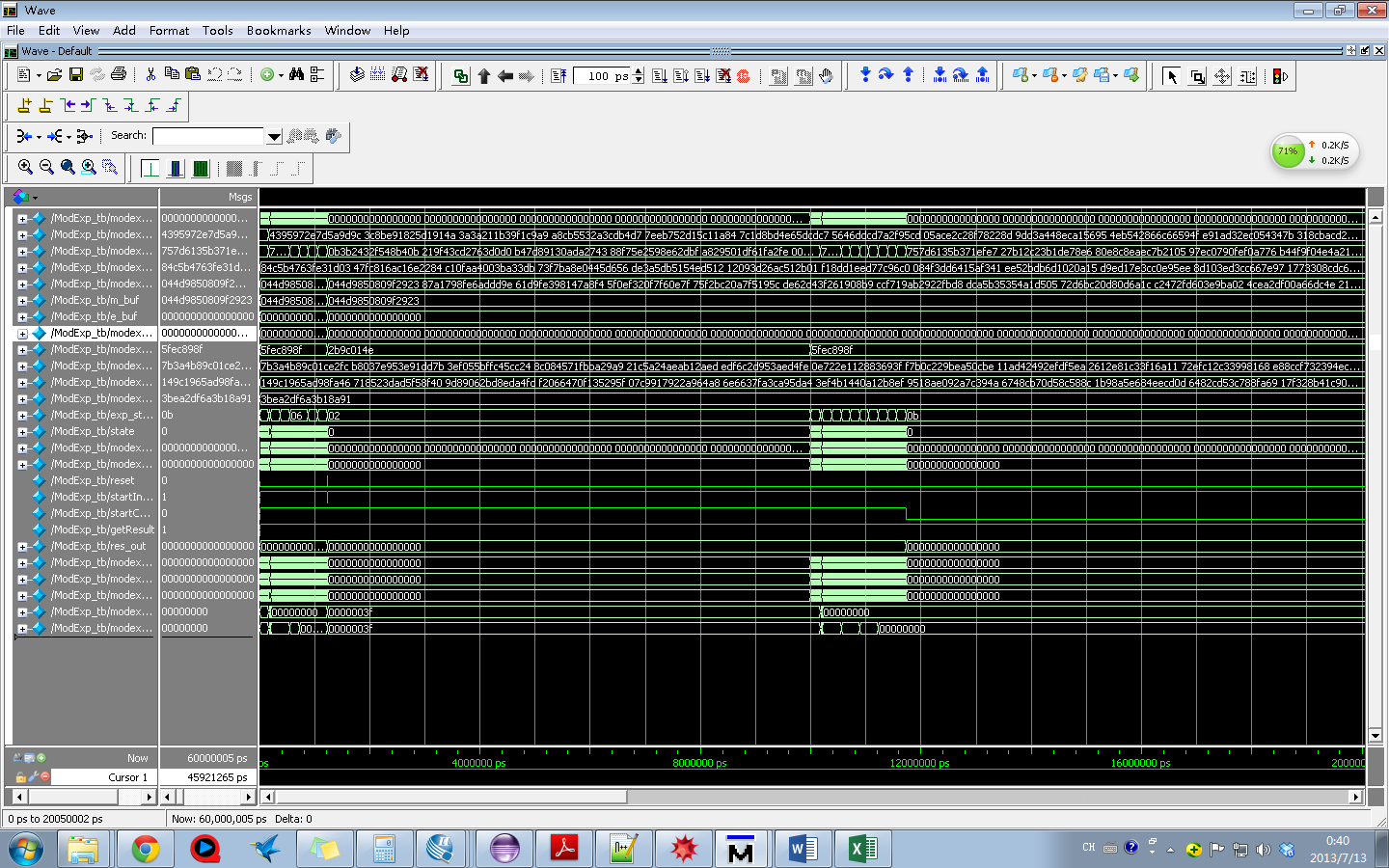


Figure 8: Snapshot of Simulation

### 6.2.2 Testing Case

m = 

e1 = 0x5 e2 = 0xf

n = 

r = 

t = 

nprime0 = 0x3bea2df6a3b18a91

result1 = m ^ e1 mod n = 

result2 = m^e2 mod n = 