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)

Written by: Qin Zhou (qinzhou@cs.ucsb.edu)

Balasubramaniam Muthuvelu (balasubramaniam@umail.ucsb.edu)

TABLE OF CONTENTS

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

1. INTRODUCTION

1.1 RSA Algorithm

The RSA algorithm, invented by Rivest, Shamir, and Adleman, is one of the simplest public-key cryptosystems. The parameters are n, p and q, e, and d. The modulus n is the product of two distinct large random primes p and q such that $n = pq$. The public exponent e is a number in the range $1 < e < \phi(n)$ such that

$$\mathrm{GCD}(e,\phi(n)) = 1,$$

where $\phi(n)$ is the Euler's totient function of n, given by

$$\phi(n) = (p-1)(q-1).$$

The private exponent of d is obtained by inverting e modulo $\phi(n)$ i.e.

$$d = e^{-1} \bmod \phi(n),$$

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

$$C = M^e \pmod n,$$

where M is the plaintext such that $0 < M < n$. The number C is the ciphertext from which the plaintext M can be computed using

$$M = C^d \pmod n.$$

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]

1.2 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.

1.3 Montgomery transform based modular exponentiation algorithm

The 4096-bit numbers are organized as sw bits, where s is the number of words and w is the word length, such that $sw = 4096$. We will take $s = 64$ and $w = 64$ bits in our implementation.
The input variables are as provided below:

1) Input message m, 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 e which is also of size 4096 bits. Exponent e is k-bit unsigned integer where $k \leq 4096$. Inputs m and e are our primary inputs.

3) The modulus operator n which is of size 4096 bits. n is a 4096 bit signed integer. Also, n is assumed to be an odd number such that the least significant bit $n_0 = 1$ and the most significant bit of message n would be 0 when $n > 0$ due to properties of two's complement notation.

4) The input $n0'$ which is pre-computed from n and w such that

$$n_0' = -n^{-1} \pmod{2^w}$$

5) The Montgomery constant r defined as

$$r = 2^{sw} \pmod{n}$$

6) The constant t defined as

$$t = r^2 \pmod{n} \text{ or } t = r^{2sw} \pmod{n}$$

The inputs $n0'$ is one word in size (64 bits) while r and t are signed integers of size 4096 bits. The output of the process is the cipher text c of size 4096 bits.

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

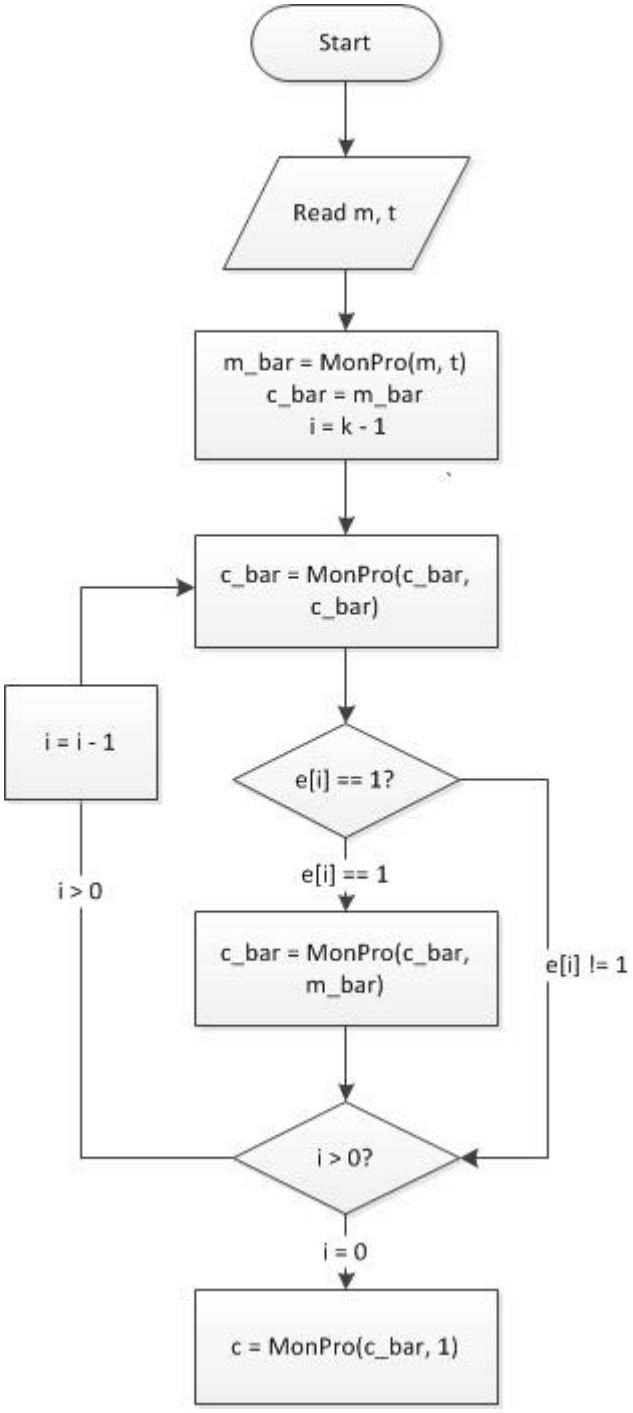

Figure 1: Workflow of 1-bit ModExp Algorithm

The pseudo code is below:

Find the index $k < 4096$ *of the leftmost* 1 *in e*

$\overline{m} = MonPro(m,t)$

$\overline{c} = r$

For $i = k - 1$ *downto* 0

$\overline{c} = MonPro(\overline{c},\overline{c})$

If $e_i = 1$ *then* $\overline{c} = MonPro(\overline{c},\overline{m})$

$c = MonPro(\overline{c},1)$

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

1.4 MonPro Algorithm

The MonPro algorithm implements the equation: $\mathrm{z} = (\mathrm{x} \cdot \mathrm{y} \cdot \mathrm{r}) \bmod \mathrm{n}$. It takes two inputs: x and y and computes the output z such that each variable holds an s-word (total sw bit) signed integer. In this case x and y are 64 words of 64 bit each. The initial value of z is initialized to zero. Hence,

$$\mathrm{x} = \mathrm{x}_{63}\mathrm{x}_{62}\ldots\mathrm{x}_1\mathrm{x}_0$$

$$\mathrm{y} = \mathrm{y}_{63}\mathrm{y}_{62}\ldots\mathrm{y}_1\mathrm{y}_0$$

$$\mathrm{z} = \mathrm{z}_{63}\mathrm{z}_{62}\ldots\mathrm{z}_1\mathrm{z}_0$$

The secondary inputs n_0', n, r and t are also assumed to be available. We have

$$\mathrm{n} = \mathrm{n}_{63}\mathrm{n}_{62}\ldots\mathrm{n}_1\mathrm{n}_0$$

$$\mathrm{n}_0' = \mathrm{n}_0',$$

where n has 64 words of 64 bit each and n_0' 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 x, namely x_0 and multiply by the 64-word y, and add it to the 64 word partial product z (which is now all zero) to obtain the (65)-word temporary result v as $\mathrm{x}_0 \cdot \mathrm{y} + \mathrm{z} = \mathrm{v}$.

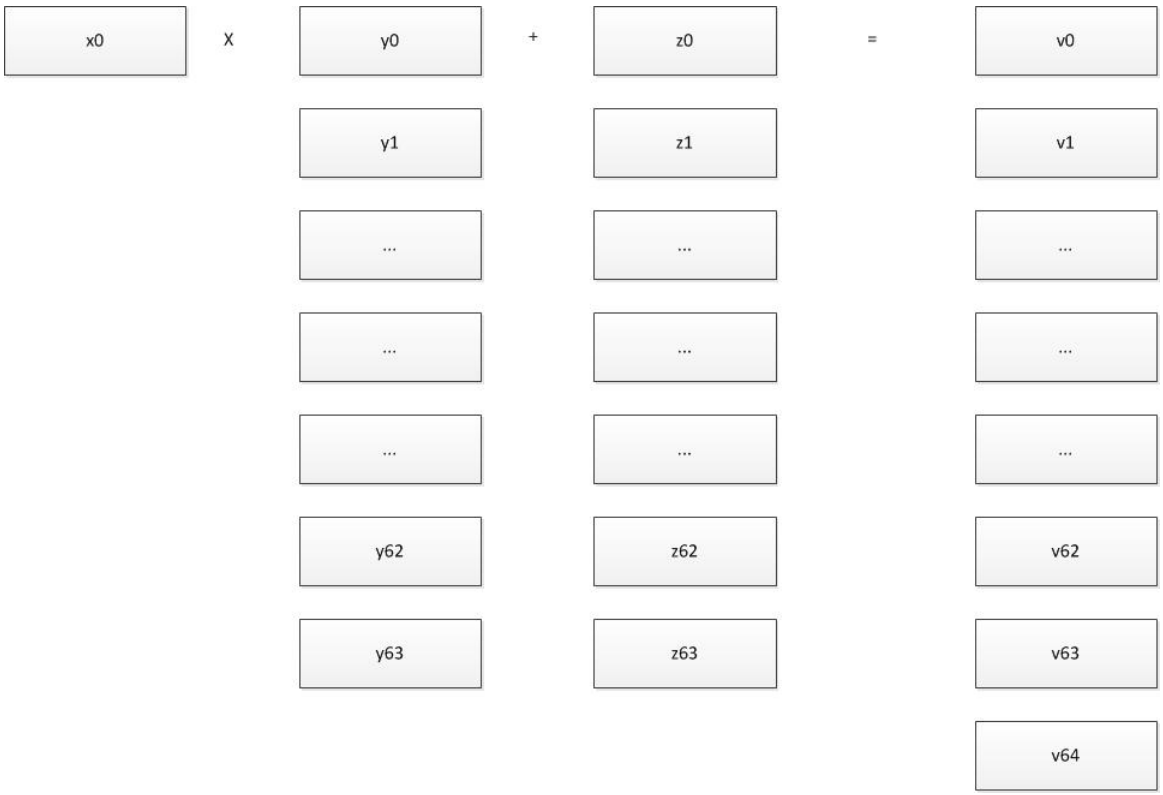

Figure 2: MonPro Step 1: $x_0 \cdot y + z = v$

Step 1a: The computation in Step 1 is accomplished using a Multiply-Add block that multiplies two 1-word numbers (x_0 and y_0), adds the previous higher word (C_0), and adds another 1-word number (z_0), producing a 2-word number (C_1, S_0); the lower word (S_0) is assigned to value (v_0), while the higher word (C_1) is used for next Multiply-Add step, as follows:

$$(C_1,S_0) = x_0 \cdot y_0 + C_0 + z_0$$

$$v_0 = S_0$$

$$(C_2,S_1) = x_0 \cdot y_1 + C_1 + z_1$$

$$v_1 = S_1$$

$$(C_3,S_2) = x_0 \cdot y_2 + C_2 + z_2$$

$$v_2 = S_2$$

...

Where the initial value of $C_0 = 0$.

Step 2: We then take the LSW of v, namely v_0 and multiply by the 1-word n_0' modulo 2^w and obtain the 1-word integer m as $m = n_0' \cdot v_0 (\mathrm{mod}\ 2^w)$.

Figure 3: MonPro Step 2: $m = n_0' \cdot v_0 (\mathrm{mod}\ 2^w)$

Step 3: We take the 1-word m and multiply by the 64-word n, and add it to the 65 word temporary value v (from step 1) to obtain the new partial product which is the 65 word z.

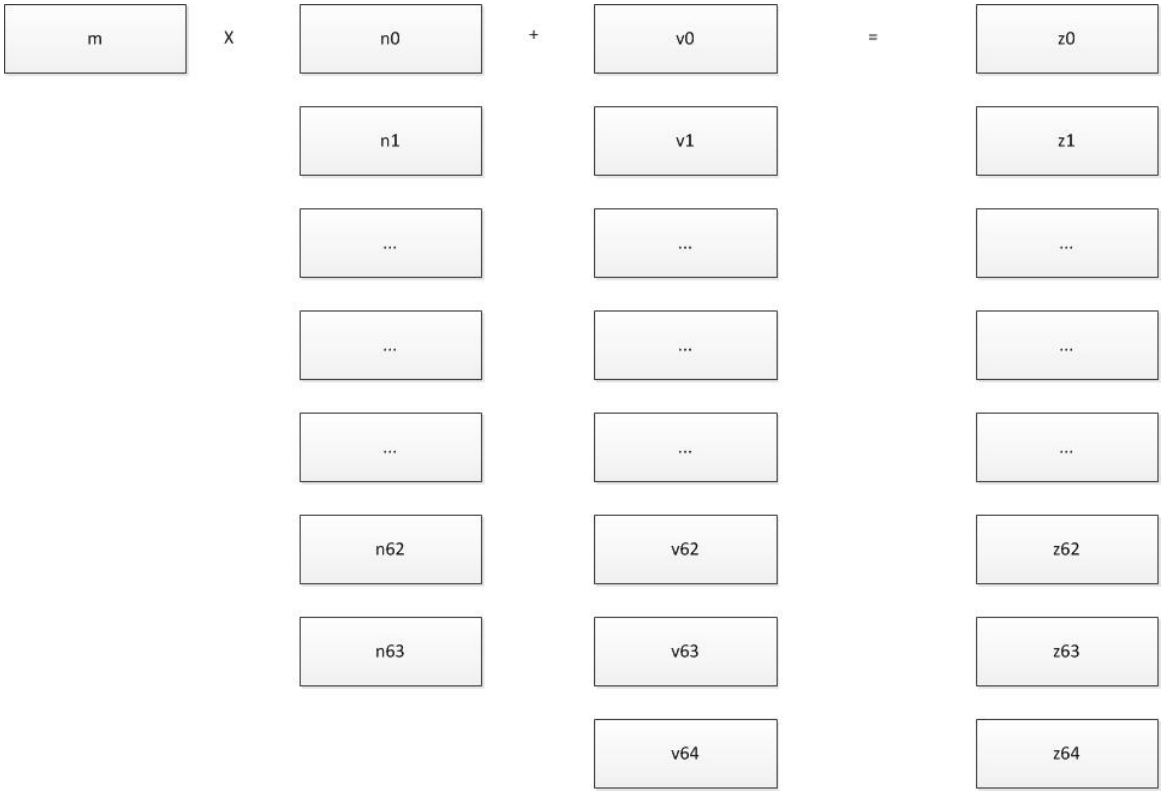

Figure 4: MonPro Step 3: $z = m \cdot n + v$

Step 3a: The computation in step 3 is accomplished using a Multiply-Add block that multiplies two 1-word numbers (m and n_0), adds the previous higher word (C_0), adds another 1-word number (v_0), producing a 2-word number (C_1, S_0); assigning S_0 to z_0, while keeping the higher word (C_1) for the next Multiply-Add step, as follows:

$$(C_1,S_0) = m \cdot n_0 + C_0 + v_0$$

$$z_0 = S_0$$

$$(C_2,S_1) = m \cdot n_1 + C_1 + v_1$$

$$z_1 = S_1$$

$$(C_3,S_2) = m \cdot n_2 + C_2 + v_2$$

$$z_2 = S_2$$

…

Such that the initial value $C_0 = 0$.

Step 4: The resulting partial product z has its LSW set to zero, due to the Montgomery property, and therefore, we shift z to obtain the new 64 word partial product.

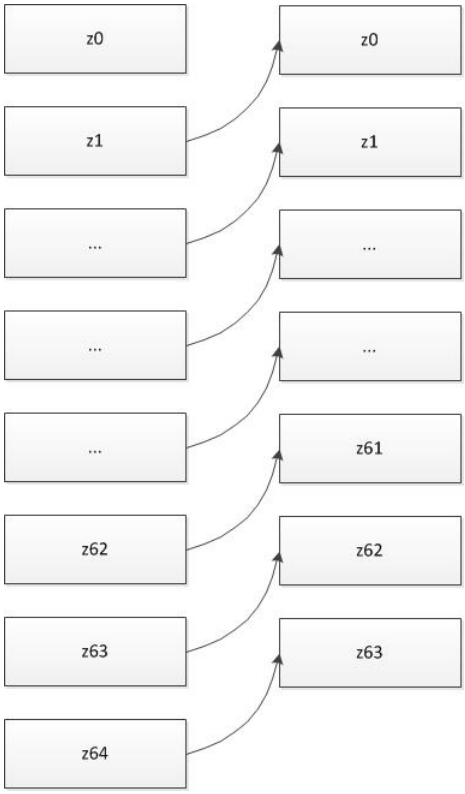

Figure 5: MonPro Step 4

Step 5: In the next step the next word of x, namely x_1 is taken and multiplied with the 64 word y and added to the partial product z to obtain the new temporary result v.

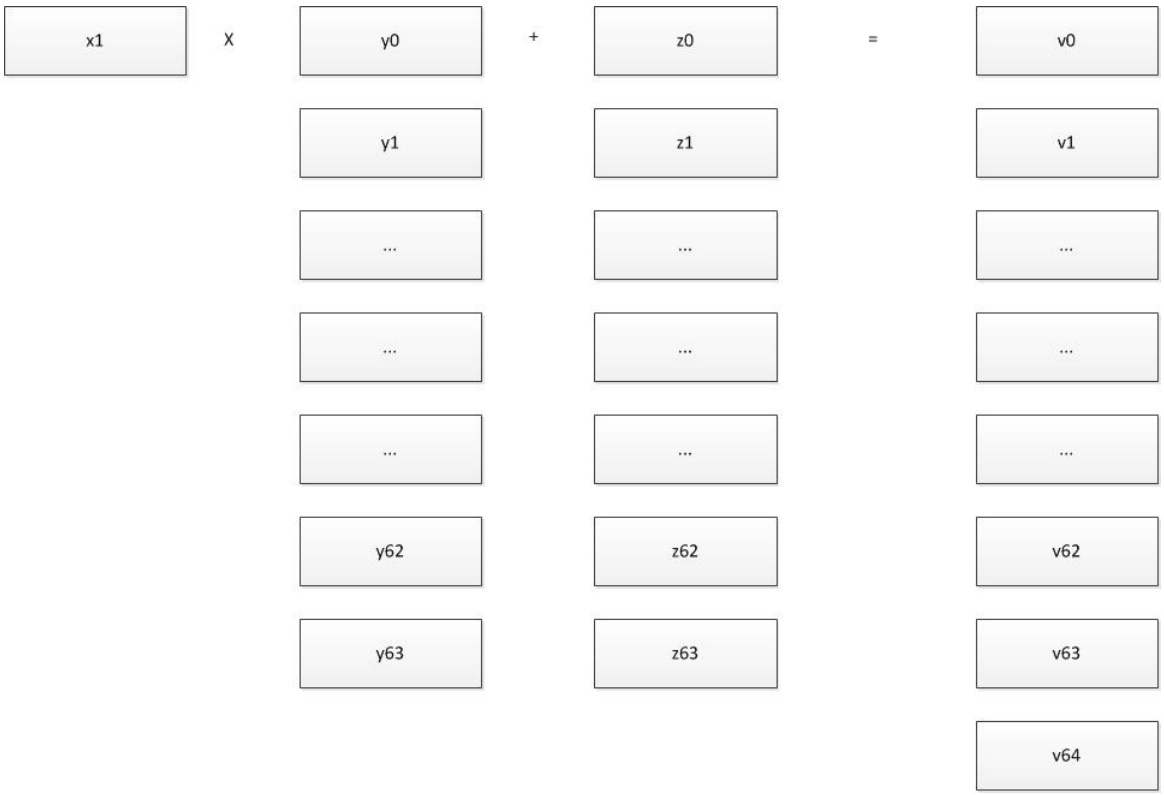

Figure 6: MonPro Step 5: $x_1 \cdot y + z = v$

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

$$m = n_0^{'} \cdot v_0 (\mathrm{mod}\ 2^w)$$

and then the multiplication of m by n, and then the addition of the result to v to obtain the new 65 word partial product z and shifting the value of z by one word (since $z_0 = 0$) to obtain the new 64 word z value.

Step 7: Proceeding in this way, we multiply all x_i values by the multiplicand y and reduce it modulo n for $i = 1,2,3$ to 63.

1.5 Secondary Input Parameters Computation

In Section 1.3 we illustrate all parameters for RSA computation, among which $n_0^{'}$, r, t are derived from n 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 $n_0^{'}$, r and t.

1.5.1 Compute n_0'

n_0' is a one-word integer which is equal to $-n_0^{-1} \pmod{2^w}$. In order to compute $-n_0^{-1} \pmod{2^w}$, we use the algorithm given below which computes $x^{-1} \pmod{2^w}$ for a given odd x.

function *ModInverse(x, 2^w) { x is odd }*

$y_1 := 1$

for $i = 2$ *to* w

if $2^{i-1} < x \cdot y_{i-1}$

then $y_i := y_{i-1} + 2^{i-1}$

else $y_i := y_{i-1}$

return y_w

Compute ModInverse(n, 2^w), negate the result then we get n_0'. [1]

1.5.2 Compute r

r equals to $2^{sw} - n$, which is the 2's complement of $-n$ in sw bits.

1.5.3 Compute t

t equals to $2^{2sw} \bmod n$, which can be computed by aligning the most significant (nonzero) bit of n to the $(2sw - 1)$th position, and then performing successive multi-precision (s-word) subtractions until all most significant positions in the resulting number until to the position $(2sw - 1)$ are zero. [2]

2 HARDWARE IMPLEMENTATION

2.1 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]

2.2 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 m and e are read from input pins. There are w pins for each parameter (64 * 2 = 128 pins). We input w bits to m and e each cycle. After s cycles sw bits m and e are read in.

n, n_0', r and t are initialized by memory initialization files (M4K mif files), which means they are hard coded in the FPGA. The reason is if we input n by input pins and change n then n_0', r and t should all be changed but it is impossible to fit the computation of t 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 m and e 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 w pins for result output using the same way as the input.

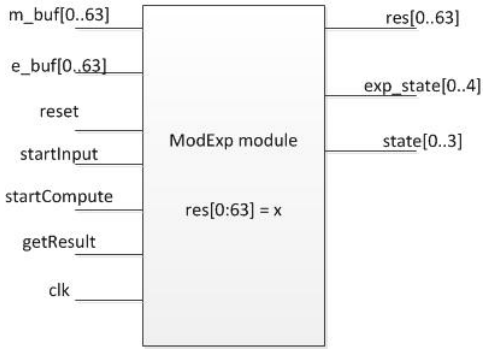

Figure 7: ModExp Module's Input and Output

2.3 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

n, n_0', r and t 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).

2. 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.

3. mul_add.v

Multiplication and addition module which computes $a \cdot b + c + d = e$; all 5 elements are DATA_WIDTH (64) bits.

4. ModExp.v

Core module which reads in m,e,n,r,t,n_0', computes $m^e = c \pmod n$, and output c to the result memory.

2.6 Module Description

1. mul_add module

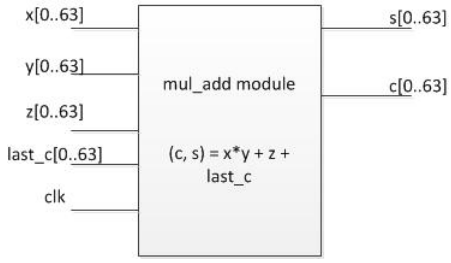

Figure 7: Mul_add Module Diagram

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

2. MonPro module

MonPro implements this equation: $z = x * y * r \bmod n$. 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 $v = x_i * y + prev(v) + z$, which corresponds to step 1a in Section 1.4;

State 1 computes the highest word of v: $(C,S) = v[s] + C, v[s] = S, v[s+1] = C$;

State 2 computes $m = \left(v[0] * n0^{'}\right) \bmod 2^w$, which corresponds to step 2 in Section 1.4;

State 3 computes $v = (m * n + v) \gg w$, which corresponds to step 3a in Section 1.4;

State 4 computes the second highest word of v: $(C,S) = v[s] + C, v[s-1] = S$;

State 5 computes the highest word of v: $v[s] = v[s+1] + C$;

State 6 dumps the result;

State 7 is the terminate state.

3. 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 m, e from input pins, go to LOAD_N;

LOAD_N: initialize n, $\mathrm{n}_0^{'}$, r and t from M4K memory, go to WAIT_COMPUTE;

WAIT_COMPUTE: wait the startCompute button, if pressed go to CALC_M_BAR;

CALC_M_BAR: computes $\overline{\mathrm{m}} = \mathrm{MonPr}(\mathrm{m},\mathrm{t})$ and $\overline{\mathrm{c}} = \mathrm{r}$, go to GET_K_E;

GET_K_E: initializes the leftmost nonzero bit of e, 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 $\overline{\mathrm{c}} = \mathrm{MonPro}(\overline{\mathrm{c}},\overline{\mathrm{m}})$, which will be used by BIGLOOP when $\mathrm{e}[\mathrm{i}] = 1$;

CALC_C_BAR_1: computes $\mathrm{c} = \mathrm{MonPro}(1,\overline{\mathrm{c}})$; 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

3.1 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.

2. 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.

3. Maximum Clock Frequency

Using TimeQuest Timing Analyzer Wizard [5], Quartus II suggests that the maximum frequency $\mathrm{Fmax} = 49.23\mathrm{MHz}$.

3.2 Time Usage

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

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 w -bit mul_add operation.
2. MonPro module follows the pseudo-code below

$for\ i = 0\ to\ s$ *(1)*

$z = x[i] * y + v$ *(2)*

$m = z[0] * n_0'$ *(3)*

$v = m * n + z$ *(4)*

output v *(5)*

Step 2 needs $2 * \mathrm{s}$ cycles; Step 3 needs 2 cycles; Step 4 needs $2 * \mathrm{s}$ cycles; Step 1 is a for loop, which means Step 1-4 needs $\mathrm{s} * (2 * \mathrm{s} + 2 + 2 * \mathrm{s})$ cycles; Step 5 needs s cycles;

Overall, MonPro needs $4 * \mathrm{s}^2 + 3 * \mathrm{s}$ cycles.

3. ModExp module follows this pseudo-code below

Initialize m, e*;* *(1)*

Initialize $\mathrm{n}, r, \mathrm{t}, \mathrm{nprime0}$*;* *(2)*

Find the index $k < 4096$ *of the leftmost* 1 *in* e *(3)*

$\overline{m} = MonPro(m,t)$ *(4)*

$$\overline{c} = r \quad (5)$$

$$For\ i = k - 1\ downto\ 0 \quad (6)$$

$$\overline{c} = MonPro(\overline{c},\overline{c}) \quad (7)$$

$$If\ e_i = 1\ then\ \overline{c} = MonPro(\overline{c},\overline{m}) \quad (8)$$

$$c = MonPro(\overline{c},1) \quad (9)$$

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

Assume the length of e is $\mathrm{L(e)}$ and the Hamming weight of e is $\mathrm{H(e)}$.

Step 1 and 2 needs $\mathrm{s} + 2\mathrm{s}$ cycles;

Step 2 needs $4096 - \mathrm{L(e)}$ cycles;

Step 3 needs $4 * \mathrm{s}^2 + 3 * \mathrm{s}$ cycles;

Step 4 needs s cycles;

Step 6 needs $\mathrm{L(e)} * (4 * \mathrm{s}^2 + 3 * \mathrm{s})$ cycles;

Step 7 needs $\mathrm{H(e)} * (4 * \mathrm{s}^2 + 3 * \mathrm{s})$ cycles;

Step 5-7 needs $(\mathrm{L(e)} + \mathrm{H(e)}) * (4 * \mathrm{s}^2 + 3 * \mathrm{s})$ cycles;

Step 8 needs $4 * \mathrm{s}^2 + 3 * \mathrm{s}$ cycles.

Overall, ModExp needs

$$\mathrm{Tc} = 7 * \mathrm{s} + (4096 - \mathrm{L(e)}) + 4 * \mathrm{s}^2 + 3 * \mathrm{s} + (\mathrm{L(e)} + \mathrm{H(e)}) * (4 * \mathrm{s}^2 + 3 * \mathrm{s}) + 4 * \mathrm{s}^2 + 3 * \mathrm{s}$$

$$= 4096 - \mathrm{L(e)} + 10 * \mathrm{s} + 4 * \mathrm{s}^2 + (2 + \mathrm{L(e)} + \mathrm{H(e)}) * (3 * \mathrm{s} + 4 * \mathrm{s}^2) \quad \text{cycles.}$$

Cycle length is the inverse of the frequency F. The total time in seconds Ts is given as $\mathrm{Ts} = \mathrm{Ts/F}$.

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

The table below enumerates the total time Ts in terms the parameters, F: The frequency, $\mathrm{L(e)}$: The length of the exponent, and $\mathrm{H(e)}$: 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

4. FUTURE WORK

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

5. 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.*

6. APPENDIX

6.1 Appendix A: Pin Assignments

6.1.1 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

6.2 Appendix B: Simulation Example

6.2.1 Summary

In this appendix we provide a complete example to verify the implementation: first we create m, e and some control signals in test bench file, and create n, r, t, nprime0 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.

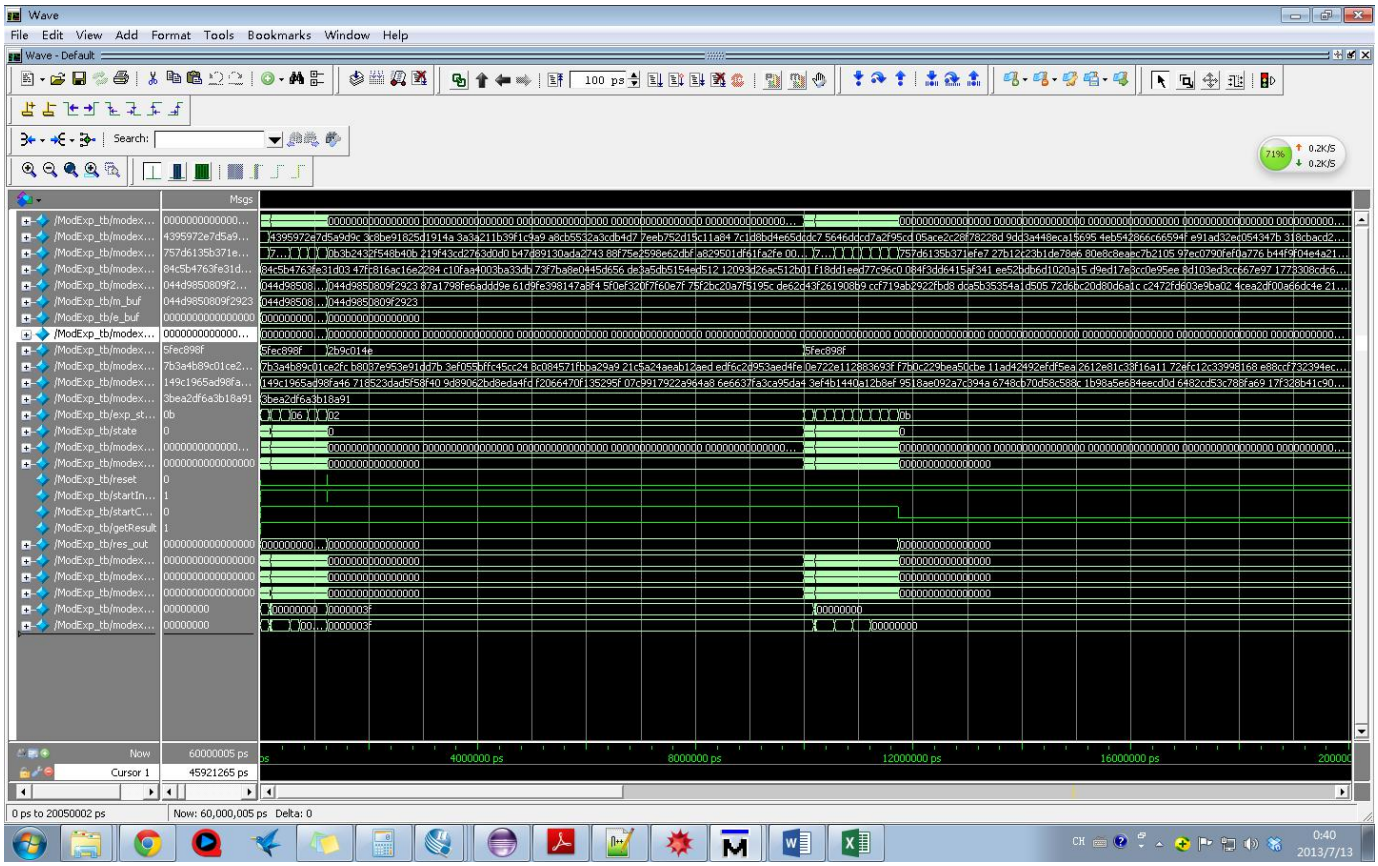

Figure 8: Snapshot of Simulation

6.2.2 Testing Case

m =

0x44d9850809f292387a1798fe6addd9e61d9fe398147a8f45f0ef320f7f60e7f75f2bc20a7f5195cde62d

43f261908b9ccf719ab2922fbd8dca5b35354a1d50572d6bc20d80d6a1cc2472fd603e9ba024cea2df00a
66dc4e21681081399f8a8f10fc9eee0a1727f7ea5f24b6de6fec4b843b2a7d15ab2c21ccc93ff710fce97d7
86e30efce9b2e70b4d4dfccb7d779cc4b5ca436953c178e61067a8cd7a3283c27e969e2c8bf23fb9a431f
7a41c30359dfde228125fb5f3d866d7002091472ad52631db9d17034ce51797350e6256403bf3df0bbf6
6ac168b4a1ca795718ada2027c013f38018399ee6a8e2f9c19ed348af5890333b5b3cedfec4623ab8996
05a2939b3b7fa74d8aff88ec827f99d273d5627386528cc241e345ac72eac39204ade7cef37ed2ec2f856f
3d95e0ae1a1b6c596216ae0fdbc8a36bcb0167e98363905c053b25fdacbe7ce71b48fba52e5998a33736
fd1ac7ce1ad0a6f226bdd974d3b564b08be04c3e5c94938160c6b3ed755a3ac132ae2a201ac902ee2577
7cf09f9821883744da64cc249558f2ad985fff3e0ba10ac728b4a41865bf350d278d41a8a6e165e049937f
411fed1e70e79933a1d1c2ad4ab155c09fcd8f739cd488869bdbd2e72bb5b707120911b3b68b57da54f
267dd138266d26d53961058fe8c1d7173e55bc7fdeb31234efe6e6480432aa50f4ec6f0093395d180514
2cb6d1dL

e1 = 0x5 e2 = 0xf

n =
0x44c5b4763fe31d0347fc816ac16e2284c10faa4003ba33db73f7ba8e0445d656de3a5db5154ed51212
093d26ac512b01f18dd1eed77c96c0084f3dd6415af341ee52bdb6d1020a15d9ed17e3cc0e95ee8d103
ed3cc667e971773308cdc6b13ab2e47dc0e959f3a518cfe5cd12d5db79ba2a7ae1f3ac7652ccdf8440407
295e4299901c0475491bc354c56c9a9cc9af4ec9546b439f9d01298a449ebe89d9bf020067dba8589890
086a17b9af5b569643d037cdff7c240d4969d495dd81355c53f0e642f43328ad088ded3c9691eb79fa5d
5f576cdeb8fc4c7b297d0b0e5e18baf320cd576d14475b349aae908fb5262cc703806984c8199921167d
8fcf23cae883333218bd91a1b7f03edca7e2dcaa37f463b337d20b5d59db610487c89da11b62397bc701
762741bab9f87ff50592859be3cecb8c497c68a8c24d4244ef7febe8e5b4617589a82b5a702cfa93ea5c4
ed8f33418f3d4e7115804f92283868a29678a5aa33b6fe5078c5fe8f8dc3bf364eb8ac8ce8a245e6b3313
8131c541013d0326324dfb695ffb3a1890c78092b4d42b28fef02b9c014ea5ac06d864c2f2e39403560d9
7dae38d9d643c25fbb230bbd92a4aa2b410d93c4efbc8d60b21fbac78255d6807923986bb968a437d5c
8dfc5eda92d864ac5db9d707107e855c384429e821a4c74803e31ba1621582283d15a9ec0806705fca1
61622bd795fec898fL

r =

0x7b3a4b89c01ce2fcb8037e953e91dd7b3ef055bffc45cc248c084571fbba29a921c5a24aeab12aededf
6c2d953aed4fe0e722e112883693ff7b0c229bea50cbe11ad42492efdf5ea2612e81c33f16a1172efc12c
33998168e88ccf732394ec54d1b823f16a60c5ae7301a32ed2a248645d5851e0c5389ad33207bbfbf8d6
a1bd666fe3fb8ab6e43cab3a9365633650b136ab94bc6062fed675bb6141762640fdff982457a7676ff79
5e84650a4a969bc2fc8320083dbf2b6962b6a227ecaa3ac0f19bd0bccd752f77212c3696e148605a2a0a
893214703b384d682f4f1a1e7450cdf32a892ebb8a4cb65516f704ad9d338fc7f967b37e666dee982703
0dc35177ccccde7426e5e480fc123581d2355c80b9c4cc82df4a2a6249efb7837625ee49dc68438fe89d8
be454607800afa6d7a641c313473b68397573db2bdbb108014171a4b9e8a7657d4a58fd3056c15a3b12
70ccbe70c2b18eea7fb06dd7c7975d69875a55cc4901af873a0170723c40c9b1475373175dba194ccec7
ece3abefec2fcd9cdb20496a004c5e76f387f6d4b2bd4d7010fd463feb15a53f9279b3d0d1c6bfca9f2682
51c72629bc3da044dcf4426d5b55d4bef26c3b1043729f4de045387daa297f86dc679446975bc82a3720
3a1256d279b53a24628f8ef817aa3c7bbd617de5b38b7fc1ce45e9dea7dd7c2ea5613f7f98fa035e9e9dd
4286a0137670L

t =

0x7b3a4b89c01ce2fcb8037e953e91dd7b3ef055bffc45cc248c084571fbba29a921c5a24aeab12aededf
6c2d953aed4fe0e722e112883693ff7b0c229bea50cbe11ad42492efdf5ea2612e81c33f16a1172efc12c
33998168e88ccf732394ec54d1b823f16a60c5ae7301a32ed2a248645d5851e0c5389ad33207bbfbf8d6
a1bd666fe3fb8ab6e43cab3a9365633650b136ab94bc6062fed675bb6141762640fdff982457a7676ff79
5e84650a4a969bc2fc8320083dbf2b6962b6a227ecaa3ac0f19bd0bccd752f77212c3696e148605a2a0a
893214703b384d682f4f1a1e7450cdf32a892ebb8a4cb65516f704ad9d338fc7f967b37e666dee982703
0dc35177ccccde7426e5e480fc123581d2355c80b9c4cc82df4a2a6249efb7837625ee49dc68438fe89d8
be454607800afa6d7a641c313473b68397573db2bdbb108014171a4b9e8a7657d4a58fd3056c15a3b12
70ccbe70c2b18eea7fb06dd7c7975d69875a55cc4901af873a0170723c40c9b1475373175dba194ccec7
ece3abefec2fcd9cdb20496a004c5e76f387f6d4b2bd4d7010fd463feb15a53f9279b3d0d1c6bfca9f2682
51c72629bc3da044dcf4426d5b55d4bef26c3b1043729f4de045387daa297f86dc679446975bc82a3720
3a1256d279b53a24628f8ef817aa3c7bbd617de5b38b7fc1ce45e9dea7dd7c2ea5613f7f98fa035e9e9dd
4286a0137670L

nprime0 = 0x3bea2df6a3b18a91

result1 = m ^ e1 mod n =

0xb3b2432f548b40b219f43cd2763d0d0b47d89130ada274388f75e2598e62dbfa829501df61fa2fe00d
0d48bacddbce72a871abf800ea6317f1efc3b4616738cc54f066e884a40862d03fb41fb1bc483a0933c65
7ba0701d41395cb3568ac95683101a6b83199d45003e69b23b8013f58cd9c2bafd1eef969692c02f8829
a0f00f9fc40252b6453eec60b82c5d44375b8f59ca01c2c5d4702a8d389d808907637ea439e1deb9f395c
114b9818b0ea1ef01cc0bb5e100ec6f5f23ec5f735ab86304c840ee7839bc4d8e5eec11df160e46baf751
6011dd1e8b80812cef2a44ffefbf09bb4afe76a416837edc49be632dd108ebffda0446462ca44e4bef0b1
48a2f1126a5682d7a71c121a313ea26665efc2c6950efdcbfc354fcaae3eaf88fd76ba820cd33c17d38b88
f181325104dc51a0d7da3f2c7759be46117ce5e79e3ca2d2e2825b8b00f308e3561ca46883d2ebaa122c
ecc2dfd697cd4b282896b9f396dc142b8ab94c9c3bdd3baf26f272c2475bed56466eb3c6950bbdfebb8d
392cc0a3e46bba238552d4d3b751a37eb38c46a899af337ee5228a23a6412580a293db06810531628b7
b33e51290f7e3f70b6bc332162818862580ace77f8369e720d831ad0e2c25eb4bc69bd7b7e0290c9047
800f5fb4440a7c38d9d17052960d183537da4b7bd0c18d9f59e562b370f5674bd4ea4f8e0594eb87d0ea
8b2f1ba1f5f23L

result2 = m^e2 mod n =

0x757d6135b371efe727b12c23b1de78e680e8c8eaec7b210597ec0790fef0a776b44f9f04e4a2153a99
efb21baac679dd685c090bba31be4630c611a5fc78e4d8b181e1408df628100d14a434407ebc244079ca
2ac9c5a7f7bfe4e084f53ee6264265058397c46fc397ee34959ab94da9de293947637535b17c1a7a7468
d60c94104d1a3f70f9e246136a75a258ee4029ccf8399ce8940d6349cc02ddca40bd47794965cd87466d
90c9b81405091dbe3e4053f9ac0ae392e94d0171dc9487993adbb3467933e980641c66a1c064410a638
bd8a4f02b2d29a1cd28a9287699cc61b87559bb6dde3c6eccf3e8fbb0f06cbe983951c288e29aef885290
d481eff04b599491cb11e8a780061a402dea12f46ab7db406ddd25d2632b07ad4dc99ba343f46a42225
e264d2fb27bcd2ad94f5c7684703d87cf4921d1afb0f8d89fa01a0a48d5f952aaa402c1a85dfbd6cce3b54
410ec0ad6ca8bac124175eec94d10692c4ad9d446636ae8d5cdaf28024ba2152b61f1e8b76dac96cea3a
6af48f0c3c72f5806b7356768fca01e7b10baddfa62b63c8a43f6dd6a2d70bed09c95bf903c4a968a08ca6
3bbe5fab45648c275f33ca5b370ef222c87298ae7c04088a7f574ca9b1dbe5d630c54a9afefeda768c26e
8c79a5ff20f085afa9e405466b1fdfa98a1a7a0166537a56ec719f464201e9826a1ae90be5e5001914f4ee
c872d6772024L