The Report Committee for Brooks Colin Gillmore

Certifies that this is the approved version of the following report:

RSA in Hardware

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____

Jacob Abraham

_____

Mark McDermott

RSA in Hardware

by

Brooks Colin Gillmore B. S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2010

Abstract

RSA in Hardware

Brooks Colin Gillmore, M. S. E

The University of Texas at Austin

Supervisor: Jacob Abraham

This report presents the RSA encryption and decryption schemes and discusses several methods for expediting the computations required, specifically the modular exponentiation operation that is required for RSA. A hardware implementation of the CIOS (Coarsely Integrated Operand Scanning) algorithm for modular multiplication is attempted on a XILINX Spartan3 FPGA in the TLL-5000 development platform used at the University of Texas at Austin. The development of the hardware is discussed in detail and some Verilog source code is provided for an implementation of modular multiplication. Some source code is also provided for an RSA executable to run on the TLL-6219 ARM-based development platform, to be used to generate test vectors.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

CPU: Central Processing Unit

RSA: Rivest Shamir and Adleman (a cryptography scheme)

TLL: The Learning Labs

UCF: User Constraints File

ISE: Integrated Synthesis Environment

FPGA: Field-Programmable Gate Array

CPLD: Complex Programmable Logic Device

INTRODUCTION

RSA cryptography, named for its inventors, Rivest, Shamir and Adleman[1], is a well-established method of public-key cryptography. Public-key cryptography systems rely on "one-way" functions, wherein a certain operation that is easy to do may prove prohibitively time consuming to un-do. In practice RSA cryptography is not an operation for which a general-purpose processor is well equipped. A general-purpose computer typically has a word-size of 32 or 64 bits, and can efficiently perform operations on numbers with bit-lengths less than the word-size of the host system. Larger operations that do not require exact precision can be performed with floating-point hardware, as long as rounding error is acceptable. Because many public-key cryptography systems use 1024-bit numbers or larger, and rounding is not acceptable, a software-based approach must be used to compute the result with full precision. Rather than performing the operation in one or two steps, the computer must operate on chunks of data one word-size at a time and store many intermediate values in memory, which is clocked at a much lower rate than the CPU. Such software systems are said to perform arbitrary-precision or multiple-precision arithmetic, meaning that one can calculate precise results out to an arbitrary number of bits depending only on the amount of memory available to the host system. This report presents a system that can be used to perform such operations with 1024-bit precision in hardware.

PART 1: BACKGROUND INFORMATION

1.1: TLL-5000 Development Platform, TLL6219 and Tools

The TLL-5000 development platform is used in several classes at UT Austin. The TLL-5000 is pictured below

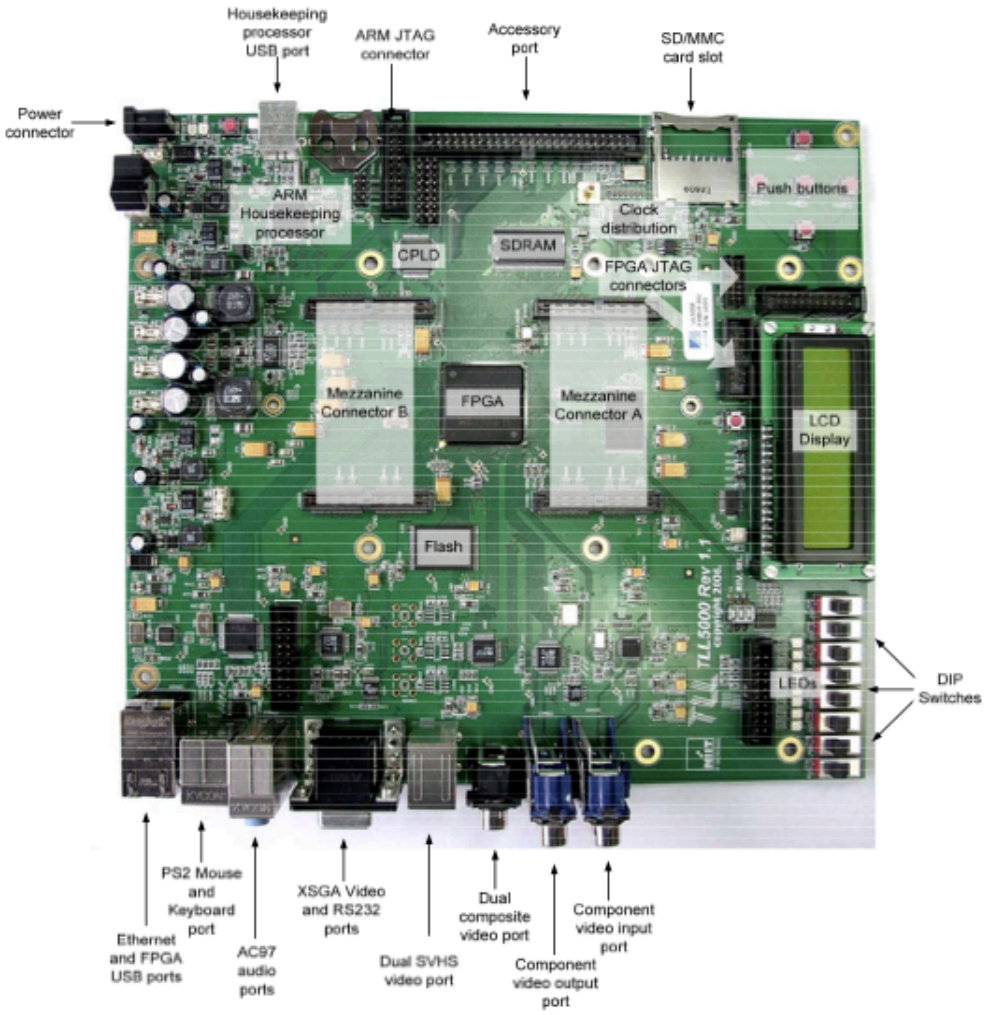

Figure 1.1: TLL-5000 Development Platform [2]

The main item of interest in the context of this report is the 1.5-million gate Xilinx XC3S1500 Spartan3 FPGA [3], in which the hardware is actually implemented. The host operating system runs on a TLL-6219 board mounted on the TLL-5000 Mezzanine A connector. The TLL-6219 [2] is an ARM-based board computer running BusyBox Linux with kernel 2.6. Through the Mezzanine connector and the CPLD the TLL-6219 has access to the FPGA. These systems, being used in several classes at UT, are familiar to the author but much of the code should be easy to port to similar platforms. The design was compiled with Xilinx ISE 11[4], which requires that a constraint file (UCF) file be provided to specify timing constraints and pin placement, which should be designed to the user's specific system. The provided driver code, the GNU MP multiple-precision C libraries, and application was cross-compiled for ARM using the CodeSourcery G++ tool chain. The prime number generating functions were verified (but not exhaustively) using openSSL which performs a similar primality test.

1.2: Public – Key Cryptography

In cryptography, a message which two parties wish to keep secret is exchanged in a pre-arranged way. There are many possible ways to do this, but the most straightforward approaches require that the two parties, traditionally named

Alice and Bob (for party A and B), exchanging the secret message must first have some sort of secure or private contact with each other during which they can exchange "private keys." Essentially, there must be some way for Alice to encode the message such that an eavesdropping third party (usually called Eve) would not be able to understand it, then transmit it "in the clear" on some unsecured channel or network, after which Bob must be able to decode it. In symmetric-key cryptosystems, Alice and Bob both know a private key or a secret algorithm that can be applied, then easily reversed. They must both understand the system and both have access to a pre-determined secret method of encoding and decoding.

It is not always possible to exchange a secret message, or establish a pair of private keys in advance, and this is where public-key cryptography comes in. It turns out that by using one-way functions it is possible to create an "asymmetric" pair of keys, one key that is kept private and one key that can be made public. Anyone can use Alice's public key to encrypt a message and send it to her, then Alice can use the private key to decode it. Because of the one-way functions involved, the public key cannot be used to decrypt the message in a reasonable amount of time.

1.3: RSA Cryptography

RSA cryptography is one such system of asymmetric key cryptography.

RSA key generation consists of five basic steps, each of which requires some sort of mathematical operation.

1) Pick 2 or more prime numbers $n_1, n_2, \ldots, n_{k-1}, n_k$
2) Compute the product of these numbers, call it P.

   This is the modulus for the public key and the private key.
3) Compute the least common multiple of $n_1-1, n_2-1, \ldots, n_{k-1}-1, n_k-1$, call this T.
4) Pick a positive integer that is coprime with and less than T,

   call this number E.

   This is the public-key exponent.
5) Pick a positive integer that, when multiplied by the public-key exponent, is congruent with 1 mod T, call this number D.

   This is the private-key exponent.

This is useful because of the way numbers modulo the composite number P behave under modular multiplication. For convenience, we want to pick a T that is the smallest number for which $x^T = 1 \bmod n_k$ for all values of n_k and for almost all values of x. This means we need $n_1-1, n_2-1, \ldots, n_{k-1}-1, n_k-1$ to be divisors of T, so T must be the least common multiple of $n_1-1, n_2-1, \ldots, n_{k-1}-1, n_k-1$. For encryption and decryption, RSA exploits the property:

$$x^{(T+1)} = x \ (mod\ P) \text{ for any } 1 < x < T$$

The above equation is implied by the Chinese Remainder Theorem, which states that if:

$$x = 0 \ (mod \ \ n_k)$$

then it follows that:

$$x^{(T+1)} = x \ (mod \ \ n_k)$$

and since this is true for all n_k , and if P is a composite of all values of n_k, then it is true for P as well:

$$x^{(T+1)} = x \ (mod \ P).$$

In RSA cryptography, encryption is done with the public key. Once one has generated a pair of public and private keys, anyone can use the public key to encrypt a message that you can then decrypt with the private key. The encryption process is computationally costly, but decryption using only the public key is computationally prohibitive. Decryption can be done with the private key (in fact, this is the purpose of the private key), but decryption using the private key is still computationally expensive. The hardware implemented here targets the encryption/decryption process, specifically modular exponentiation.

Before a plaintext message can be encrypted, it is typically "padded." Essentially, padding adds text to the message until it is of a specific length. Padding can prevent certain cryptographic attacks that exploit different ciphertext lengths as the plaintext changes [6]. For example, a Morse code message where dots and dashes are then encrypted would not be very well hidden if the data packet that

meant “dot” turned out to be half the size of the data packet meaning “dash”. Conveniently, this also makes hardware design less difficult because we now only have to worry about doing modular arithmetic on numbers that are a certain multiple of a known word size, rather than doing modular math on arbitrarily long numbers.

The padded plaintext is then used as “x” in the equations above. If the plaintext is m, the ciphertext c is given by:

$$c = m^E \ (mod\ P)$$

And the ciphertext can now be decrypted by computing c^D (mod P), which is true because

$$c^D \ (mod\ P) = (m^E)^D = M^{(ED)} = m \ (mod\ P)$$

so c^D contains the original plaintext message m.

RSA is uniquely useful because not only can the keys be used for encryption and decryption; they can also digitally sign documents using the same math. Signing a message is done by computing

$$s = m^{(D)} \ (mod\ P)$$

giving the signed message (m,s). To verify that the message comes from the owner of the private key, anyone with the public key can now verify that

$$s^E = m \ (mod\ P)$$

to authenticate the message because only the holder of the private key knows D, the private exponent used to compute the signed message.

1.4: Modular Exponentiation Algorithms

To accelerate this process in hardware, one must be able to compute c^D (mod P) quickly, and there are several ways of doing this. Hardware division is slow, and a naïve approach to this calculation requires that we divide many times by the modulus P, so accelerators usually try to avoid division with some clever manipulation that is faster than a normal division operation. RSA keys are very long, typically 1024 to 2048 bits in width, but RSA Security believes[7] that 3072-bit keys are required for security beyond 2030. We will outline two methods for fast modular multiplication.

One algorithm is Barrett's [8] modular reduction method To compute the modulus of a number:

$$Z \bmod N = Z - [Z/N]\ N = z - qN \text{ where } q = Z/N$$

But computing the quotient q is slow in hardware. To get around this you can compute this:

$$q = \{ [Z / 2^{n-1}] * [2^{2n} / N] \} / 2^{n+1}$$

This looks much more complicated but $[2^{2n} / N]$ can be pre-computed and stays constant for a given modulus. Division by 2^{n-1} and 2^{n+1} can be computed quickly by truncating the 2^{n-1} or 2^{n+1} least significant bits. This reduces the whole procedure to

three steps, a truncation, multiplication by a constant, and another truncation. One still needs a multiplier but no longer needs to perform division. This method is fast as long as the modulus does not change often.

The fastest algorithm used to implement modular multiplication seems to be the Montgomery algorithm [9]. This algorithm, like Barrett's method, avoids division by pre-computation. It introduces a transformation constant R, that is coprime with the modulus. Then it introduces the "M-residue" representation of two integers A and B, both less than M are defined as:

$$X = A \bullet R(mod\ M),\ Y = B \bullet R(mod\ M)$$

This is true for any R that is coprime with M and larger than M. Speed advantages appear when R is a power of 2. The Montgomery algorithm actually computes this:

$$C = A \otimes_n B = ABR^{-1}\ mod\ M = ABr^{-n}\ mod\ M$$

in a r-radix number system. For a word of length n where $M > A,B \geq 0$ and inputs X,Y (this is the same X and Y given earlier) then the algorithm computes

$$C' = XYR^{-1} = A \bullet R \bullet B \bullet R \bullet R^{-1} = ARB\ mod\ M$$

So start with Y, X, R and M, then after computing A and B, one can compute AB R mod M from that information using this method. The result AB R mod M still contains an extra factor of R that we do not need, but performing another Montgomery step with C' and 1 will remove this factor and yield the desired outcome, C, because:

$$RX \otimes 1 = X \bullet R \bullet A \bullet 1 \bullet R^{-1} = X$$

and because there of the 1-to-1 mapping shown in [9] this is also true for their products.

The above sequence represent a "Montgomery step." To perform modular multiplication with Montgomery's method, one must perform a Montgomery step to transform the number to "Montgomery space;" after one has obtained the Montgomery representation of the number, one more additional step will perform modular multiplication. After multiplication is performed, the final result is obtained by performing a Montgomery step with the product from the second step and unity, which transforms the result of the Montgomery multiplication out of Montgomery space. Montgomery's algorithm is not likely to be faster than a naïve modular multiplication, because one modular multiply actually requires three Montgomery steps. It is in modular exponentiation that the benefit is realized, because exponentiation algorithms require many successive multiplications, for example when performing exponentiation by squaring.

```
Code:

1     X = MontMult(a,b,n) {
2          r = 2^k  // k is bit-width of n
3          r_inv = inverse(r mod n)
4          n_inv = (1 – (r * r_inv))/n
5          retval = (a * b * n_inv ) mod(r)
6          if (retval) > n,) {
7               return retval – n
8          } else return retval
9     }
```

Figure 1.2: Montgomery Multiplication Algorithm [9]

With certain restrictions on the variables, it is possible to avoid the subtraction step at the end of this algorithm [10], and it is not implemented in the code provided. Typically one key will be used for many operations, so we can assume that many of the inputs will remain the same between most operations, which allows us to pre-compute r, r_inv and n_inv Figure 1.2 because n will not change unless the key changes and many multiplication steps will typically be performed even for one modular exponentiation. From Figure 1.2, in a typical Montgomery multiplication operation we end up replacing the mod n operation with a modulo r, and since r is a power of the radix of our number system (in this case binary, a power of 2) then we have replaced division with truncation as in Barrett's method. This step will be much faster in hardware than actually calculating mod n.

If we use a real-world (that is, large) value for the word-length of the arguments in Figure 1.2 such as 1024, then we see that line four implies a multiply operation as large as 1024bit x 1024bit x 1024bit. For power and area, it is still desirable to limit the word-size of the hardware implementation, so the author has chosen to implement a 256-bit word-size using an operand-scanning algorithm, as given in [11]. The coarsely-integrated operand-scanning architecture used here provides good speed, but for word-size w it still requires an addition operation of w2 + 2w. So for a 1024-bit Montgomery multiplier, the CIOS algorithm requires us to store 513-bit wide results (512 bits + 1 carry bit).

The CIOS algorithm is given in Figure 1.3. It consists of two nested loops, the i-loop and a pair of inner j-loops. The operands are "scanned" in that the algorithm only requires w bits at a time and the loops run for s words until the entirety of both operands has been read and acted on. C and S in Figure 1.3 are for carry and sum values for each word-size piece of the operand.

```
CODE:
CIOS (a.b.n.n') {

for i=0 to s-1 loop
    C = 0
    for j= 0 to s-1 loop
        (C,S) = t(j)+ a(j) * b(i) + C
         t(j) = S
    end loop
   (C,S)   = t(s) + C
    t[s]   = S
    t[s+1] = C
    c = 0
    m = t(0) * n'(0) mod W
   (C,S) = t[0] + m * n[0]
    for j = 1 to s-1 loop
       (C,S)   = t(j) + m * n(j) + C
        t(j-1) = S
    end loop
   (C,S) = t(s) + C/(s-1) = s
    t(s) = t(s+1) + C
end loop
```

Figure 1.3: CIOS Algorithm [11-13]

The CIOS algorithm in software will use whatever word-size the host machine uses, typically 32 or 64 bit as long as the hardware can store the result of 2w+2w addition. When using the Montgomery algorithm in hardware, we are free to choose our own word-size. There is a practical limit due to power and area concerns, but for the 1024-bit Montgomery multiplier detailed here, the author has chosen to divide the 1024-bit key into four 256-bit words. The design uses a state machine based on [13] to unroll the loops in the CIOS algorithm. As it runs through each state, it can perform the multiply-add steps implied in the j-loops as well as an

additional multiply and add operation implied by the code in the i-loop that is not in a j-loop. In total, 18 states are required to scan in all four words of each operand and obtain the final result.

State	Comp.	Inputs	Outputs
0	*j* loop	a(0), b(0), '0', '0'	C_{out0}, t_{out0_0}
1	*j* loop	a(1), b(0), C_{out0}, '0'	C_{out1}, t_{out0_1}
	Mult	t_{out0_0}, n'(0)	m_0
2	*j* loop	a(2), b(0), C_{out1}, '0'	C_{out2}, t_{out0_2}
	j loop	m_0, n(0), '0', t_{out0_0}	C_{out3}, *null*
3	*j* loop	a(3), b(0), C_{out2}, '0'	C_{out4}, t_{out0_3}
	j loop	m_0, n(1), C_{out3}, t_{out0_1}	C_{out5}, t_{out1_0}
4	Add	C_{out4}, '0'	t_{out0_4}, t_{out0_5}
	j loop	m_0, n(2), C_{out5}, t_{out0_2}	C_{out6}, t_{out1_1}
	j loop	a(0), b(1), '0', t_{out1_0}	C_{out7}, t_{out2_0}
5	*j* loop	m_0, n(3), C_{out6}, t_{out0_3}	C_{out8}, t_{out1_2}
	j loop	a(1), b(1), C_{out7}, t_{out1_1}	C_{out9}, t_{out2_1}
	Mult	t_{out2_0}, n'(0)	m_1
6	Add	C_{out8}, t_{out0_4}	*Csig*, t_{out1_3}
	j loop	a(2), b(1), C_{out9}, t_{out1_2}	C_{out10}, t_{out2_2}
	j loop	m_1, n(0), '0', t_{out2_0}	C_{out11}, *null1*
	Mult	t_{out2_0}, n'(0)	m_1
6	Add	C_{out8}, t_{out0_4}	*Csig*, t_{out1_3}
	j loop	a(2), b(1), C_{out9}, t_{out1_2}	C_{out10}, t_{out2_2}
	j loop	m_1, n(0), '0', t_{out2_0}	C_{out11}, *null1*
7	Add	*Csig*, t_{out0_5}	t_{out1_4}
	j loop	a(3), b(1), C_{out10}, t_{out1_3}	C_{out12}, t_{out2_3}
	j loop	m_1, n(1), C_{out11}, t_{out2_1}	C_{out13}, t_{out3_0}
8	Add	C_{out12}, t_{out1_4}	t_{out2_4}, t_{out2_5}
	j loop	m_1, n(2), C_{out13}, t_{out2_2}	C_{out14}, t_{out3_1}
	j loop	a(0), b(2), '0', t_{out3_0}	C_{out15}, t_{out4_0}
9	*j* loop	m_1, n(3), C_{out14}, t_{out2_3}	C_{out16}, t_{out3_2}
	j loop	a(1), b(2), C_{out15}, t_{out3_1}	C_{out17}, t_{out4_1}
	Mult	t_{out4_0}, n'(0)	m_2
10	Add	C_{out16}, t_{out2_4}	*Csig1*, t_{out3_3}
	j loop	a(2), b(2), C_{out17}, t_{out3_2}	C_{out18}, t_{out4_2}
	j loop	m_2, n(0), '0', t_{out4_0}	C_{out19}, *null2*
11	Add	*Csig1*, t_{out2_5}	t_{out3_4}
	j loop	a(3), b(2), C_{out18}, t_{out3_3}	C_{out20}, t_{out4_32}
	j loop	m_2, n(10), C_{out19}, t_{out4_1}	C_{out21}, t_{out5_0}
12	Add	C_{out20}, t_{out3_4}	t_{out4_4}, t_{out4_5}
	j loop	m_2, n(2), C_{out21}, t_{out4_2}	C_{out22}, t_{out5_1}
	j loop	a(0), b(3), '0', t_{out5_0}	C_{out23}, t_{out6_0}
13	*j* loop	m_2, n(3), C_{out22}, t_{out4_3}	C_{out24}, t_{out5_2}
	j loop	a(1), b(3), C_{out23}, t_{out5_1}	C_{out25}, t_{out6_1}
	Mult	t_{out6_0}, n'(0)	m_3

Figure 1.4: 1024-bit CIOS state machine for 256-bit word size [13]

1.5: GNU Multiple Precision Libraries

The software makes use of the GNU MP [14] multiple-precision libraries for generating large primes to be read into the hardware. The functions in these libraries take the form function(arg1, arg2, arg2, arg4). These functions are designed to be recognizable as their GNU C library equivalents and should be read with arg1 as the return value and all subsequent arguments as inputs to the function. For example mpz_powm(x, a, b, n) would calculate a^b mod n and put the result in x.

PART 2 – IMPLEMENTATION

2.1: Implementation Overview

The system consists of three parts: the hardware multiplier, a driver that allows applications to access the hardware, and an application to perform benchmarks on the hardware.

2.2: Software

The first step for RSA is to find large prime numbers, which is a complex problem in its own right. The application makes heavy use of the GNU-MP multiple precision libraries, which allows arbitrary precision arithmetic. First, GNU-MP must be compiled and installed on the machine with the FPGA, in this case an ARM-based development platform. The application can then be compiled to include the GNU-MP libraries. The step in the application is to generate a large random number, which for real cryptographic purposes should come from a more ideally random source.

For purposes of benchmarking the hardware it is sufficient to pick a random

seed that is constantly changing, even if it is not truly random, so this application uses the system time as a seed for the GNU-MP random number generators, then generates a large random number. This number is not necessarily a prime number, so the next step is to test the number for primality, then if it is not prime, throw it out. The primality test first performs a bitwise AND operation to test if the number is even (and therefore not prime), and immediately move on to the next candidate if it is. Then the test divides the number by all the primes less than 1000 before moving on to the most costly test, the Rabin-Miller probabilistic test.

The Rabin-Miller test does not actually test for primality, but for compositeness. One iteration of the Rabin-Miller algorithm that does not reveal a composite factor indicates a 75% probability that a number is in fact prime. Repeating the test 64 times ensures that there is only a 2^{-128} probability that the number is composite. After finding a pair of large primes, the application performs modular exponentiation in software, then hardware, and compares the latency of each operation.

The software calls the driver to execute Montgomery multiplies in hardware.

2.3: Driver

The hardware is controlled by a linux kernel module. The driver uses MMIO writes to control the state transitions of the hardware. On receipt of a write

command the driver reads in a, b, one word (256 bits) of n', and n. Arguments are read in 32 bits at a time, and when the driver receives an interrupt, it will read the result of the Montgomery multiplication out. All arguments are read in 32 bits at a time, least significant 32-bit word to most significant.

2.4: Hardware

On receipt of a command the hardware transitions between three states in a top-level state machine. In the idle state, the hardware can receive commands through MMIO writes. The hardware can receive two commands: 1 – get new input, 2 – multiply. When the hardware receives a new input command, it will expect to see the Montgomery residue values a and b, then one word of n', then n. It stores these in 1024-bit wide registers. When the hardware receives the multiply command it performs the CIOS Montgomery multiply algorithm on the input registers. The word size is 256 bits, and the 18 steps in the state machine shown in Figure 1.4 are performed. The hardware then asserts the interrupt, and reads out the result, 32 bits at a time. A block diagram of the hardware is provided in figure 2.1

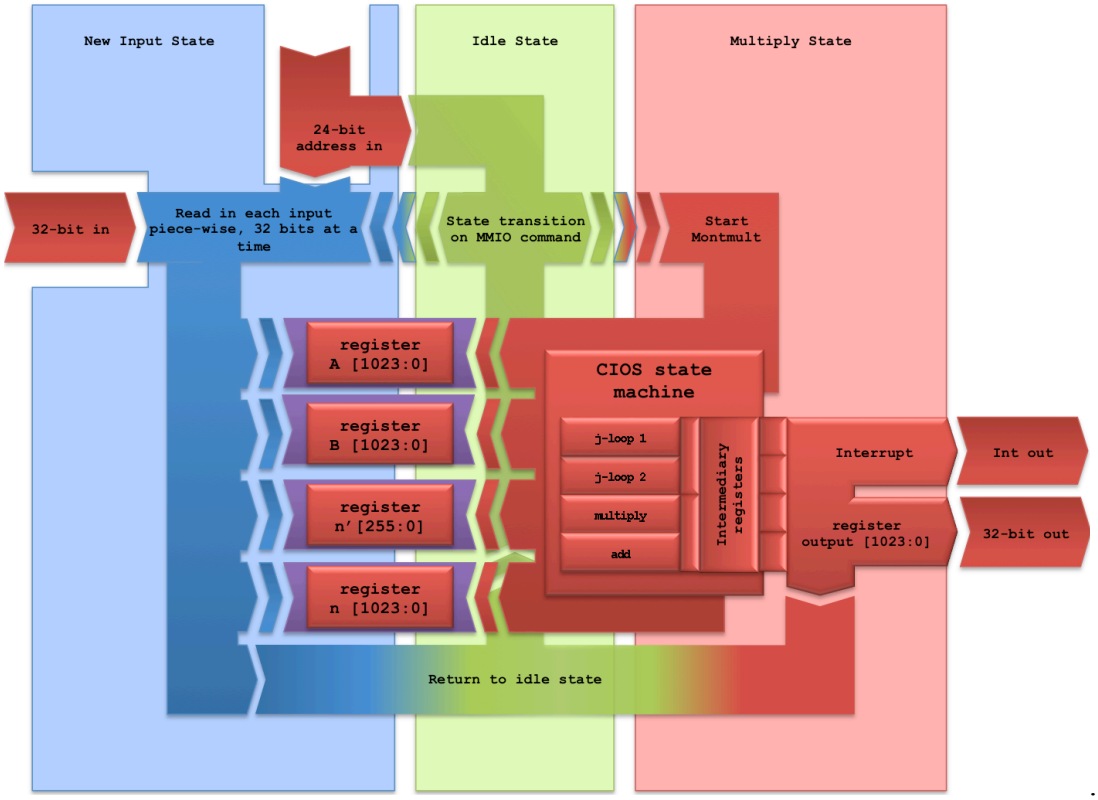

Figure 2.1: Montgomery Multiplier Block Diagram

In the above figure, the idle state is represented in green. The hardware boots into the Idle state, then expects an MMIO write command that will send it either to the New Input state or the Multiply state. In the New Input state, the hardware takes the 32-bit data bus values and reads them in to several registers that represent the actual 1024-bit arguments for the Montgomery multiplier. When all the arguments A, B, n and n' have been read in, it returns to idle and waits for a multiply command. After receiving a multiply command, the 18 steps outlined earlier in the CIOS state

machine are performed on the input operands. The CIOS algorithm consists of several nested loops, and in one cycle the hardware may perform any of the four operations in the CIOS state machine, there are two "j-loop" operations in addition to a plain multiplication and a plain addition. To unroll the algorithm from Figure 1.3 and perform it as in Figure 1.4, one must perform at most two j-loop multiply-add operations in addition to one multiplication and one addition that are separated. The four operations listed above are the mathematical operations required to perform a single pass through the outer loop (the "i-loop") in Figure 1.3. Each state in the state machine performs these operations concurrently, though all the hardware may not be used on each particular step. Figure 2.2 shows the data path for a j-loop.

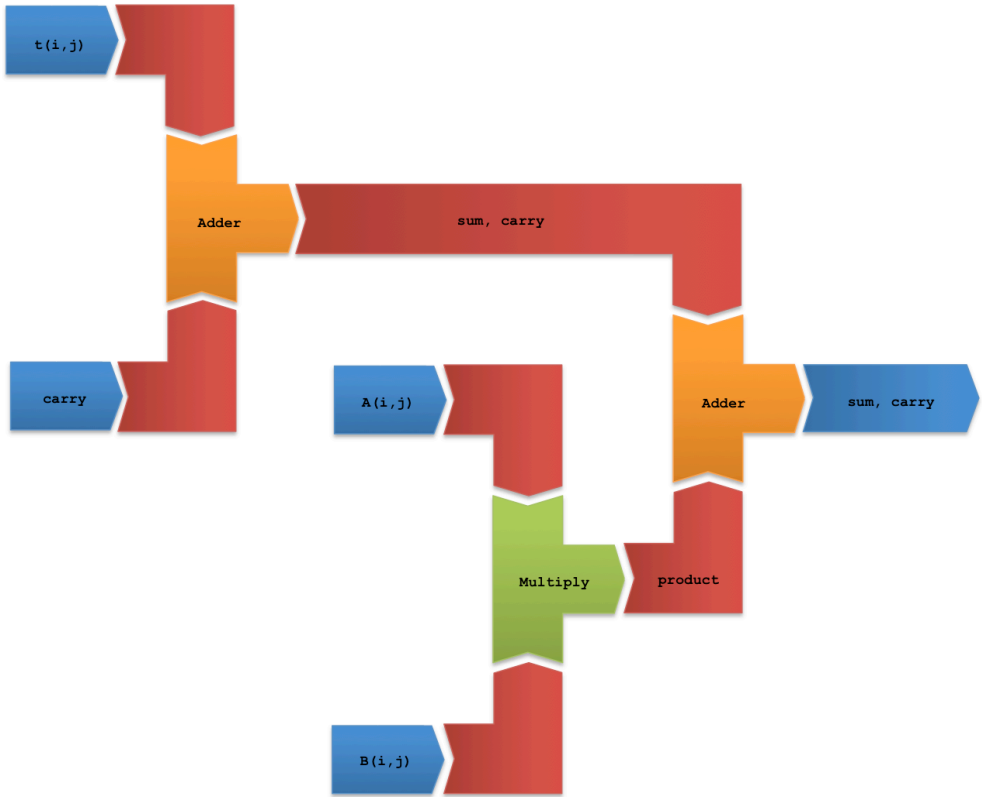

Figure 2.2: Hardware for j-loop multiply-add

PART 3: RESULTS AND FUTURE WORK

3.1: Results

Due to time constraints and amount of time spent settling on the CIOS algorithm, the design has not been fully debugged. Components have been verified to be synthesizable, but the unit as a whole has not been verified to be functionally correct. It is still possible to make some judgments about the speed of this implementation based on the synthesis results. The critical path of the top module is through the j-loop containing the 2w x 2w adder as in [12-13], and the delay is 736.901ns, which leads to a maximum operating frequency of 1.3MHz. Worth mentioning is that most of the delay through the j-loop in this implementation is due to the multipliers (only 42ns of it is due to the adder), where the results in [12-13] using a different (and much newer) FPGA indicated that the 2w x 2w add was the limiting factor. The target Spartan3, uses 18x18 bit multipliers, where the Virtex in [12-13] has access to much wider 38-bit multipliers.

Though this is not a practical application in itself, the author hopes that the code will be useful work for someone interested in implementing the CIOS algorithm in hardware, and to that end, much of the source code is provided in the Appendix.

3.2: Future work

There are several opportunities for future work on this design. The most obvious is the development of a testbench for functional verification. Also, the j-loop is a naïve implementation and could be pipelined. It would also be a good idea to add an extra state to the top-level state machine that reads in only A and B so that n and n_prime can stay inside the hardware, rather than being read in with every add cycle.

APPENDIX

Software Source Code

```
#include <stdio.h>
#include <stdarg.h>
#include <gmp.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <math.h>

//#define DEBUG // some debugging statements to show which functions we're in
//#define RM_DEBUG // show each time we run rabin miller test

//=======================
// Global Variables
//=======================
// This is a data structure to hold the RSA key
struct rsa_key { // n, d, d, p, q are all the values required to make an RSA
public/private keypair
        mpz_t n; // product of p and q
        mpz_t e; // random intiger coprime to phi(pq), the euler totient or (p-1)(q-
1), that is coprime with it, that is, gcd(totient, e) == 1
        mpz_t d; // d such that de is contruent to 1 mod (psi(pq))
        mpz_t p; // large random prime
        mpz_t q; // another large random prime
};

//=======================
//function prototypes
//=======================

int main(int argc, char **argv);
int generateLargePrime (int keyLength, mpz_ptr largePrime, int seed);
int rabinMiller(mpz_ptr n, int seed);
int isPrime(mpz_ptr n, int seed);

//=======================
//actual program
//=======================
int main(int argc, char **argv) {

        mpz_t p_minus_one;
        mpz_t q_minus_one;
        mpz_t totient;
        mpz_t l;
        mpz_t plainText;
        mpz_t cipherText;
        mpz_t computedPlainText;
        mpz_t negative_1;

        int keysize;
        int seed;
        int i;
        int messageLength;
        int nextChar;
        int plaintTextArray[];
        int outputArray[];

        FILE *plainTextFile;
        FILE *outputTextFile;

        char *fileName;
```

```
        struct rsa_key key;
        mpz_init(key.n);
        mpz_init(key.e);
        mpz_init(key.d);
        mpz_init(key.p);
        mpz_init(key.q);
        mpz_init(p_minus_one);
        mpz_init(q_minus_one);
        mpz_init(totient);
        mpz_init(l);
        mpz_init(plainText);
        mpz_init(cipherText);
        mpz_init(computedPlainText);
        mpz_init(negative_1);

        time_t seconds;
        time(&seconds);

        mpz_set_si(negative_1, -1);
        // program requires:
        //      1: key size
        //      2: path to a plain text file to encrypt
        //      3: optional random seed for repeatability/debug, if no random seed is
passed program will use system time for random seed

        if ((argc != 3)||(argc != 4)) // error if we don't see the command-line
arguments we expect
        {
                fprintf(stderr, "Usage: %s <key size in bits> <path to file to
encrypt/decrypt, must be less than 100 characters long> <(optional, for
repeatability) random seed>\n", argv[0]);
                return(-1);
        }

        keysize = atoi(argv[1]); // get key size from command line arguments

        if (argc == 3) { // no random seed passed from command line
                time(&seconds); // get a random seed from the system time
                srand((unsigned int) seconds);
                seed = rand();
                printf("Seed = %X\n", seed);
        }

        if (argc == 4) { // command line seed was passed
                srand((unsigned int) atoi(argv[2])); // use command-line random seed
                seed = rand();
                printf("Seed = %X\n", seed);
        }

        printf("Generating %d - bit key\n", keysize);
        printf("Main: Generating p...\n");
        generateLargePrime(keysize, key.p, seed); // generate RSA p

        seed += rand();
        printf("New seed = %X\n", seed);
        printf("Main: Generating q...\n");
        generateLargePrime(keysize, key.q, seed); // generate RSA q

        printf("Main: Calculating n = p * q, this is the RSA modulus...\n");
        mpz_mul (key.n, key.p, key.q); // compute n, the product of p and q
        gmp_printf("RSA Key mudulus n = \n%ZX\n which is %d bits long\n", key.n,
mpz_sizeinbase(key.n, 2));

        printf("Main: Calculating totient phi(pq) = (p-1)*(q-1)...\n");
        mpz_mul (key.n, key.p, key.q); // this part calculates euler's totient for p
```

```
and q
        mpz_sub_ui(p_minus_one, key.p, 1ul);
        mpz_sub_ui(q_minus_one, key.q, 1ul);
        mpz_mul (totient, p_minus_one, q_minus_one);
        gmp_printf("Totient = \n%ZX\n which is %d bits long\n", totient,
mpz_sizeinbase(totient, 2));

        seed += rand();
        printf("New seed = %X\n", seed);
        printf("Main: picking random number 1 < e < totient, must be coprime with
totient...\n");
        i = 0;
        do {
                srand(seed);
                seed = rand() % 32765 + 23; // generate number between 23 and 32765 to
use as candidate for e,
                i++;
        } while((mpz_gcd_ui(key.e, totient, seed) != 1)||(i < 20000)); // generate a
new random number if not comprime until 20,000 tries

        if (i=200000) {
                printf("Main: Error: Failed to find RSA e after 20,000 tries!\n");
                return -1;
        }
        gmp_printf("e = %ZX, which took %d tries.\n", key.e, i);

        // compute private key d
        printf("Main: Computing d = e^(-1) mod totient.\n", key.e, i);
        mpz_powm(key.d, key.e, negative_1, totient);
        gmp_printf("d = %ZX\n", key.d);

        // open plaintext input file
        fileName = arg[2];
        plainTextFile = fopen(fileName, r);
        if (plainTextFile == null) {
                printf("Error: Plain text input file does not exist!\n");
        }
        else {
                i = 0;
                while (((nextChar = fgetc(plainTextFile)) != EOF) && i<=100){
                        plainTextString[i] = nextChar;
                        i++;
                }
                plainTextString[i] = 0;
                messageLength = i;
                .fclose(plainTextFile);
        }

        // convert our plain text integer array to a gnu-mp int
        printf("Main: Converting plain text file to plain text m.\n");
        mpz_import(plainText, messageLength, 1, sizeof(plainTextArray[0], 0, 0,
plainTextArray); // imports plain text int array to gnu mp integer "plainText"
        gmp_printf("m = %ZX\n", plainText);

        // to-do: add a padding scheme

        // compute the cipher text from plain text and our *public* RSA key (n and e)
        printf("Main: Computing cipher text M: M = m^e mod n.\n");
        mpz_powm(cipherText, plainText, key.e, key.n);
        gmp_printf("M = %ZX\n", cipherText);

        // compute the plain text from cipher text and our *private* RSA key (d and
n)
        printf("Main: Computing plain text m: m = M^d mod n.\n");
        mpz_powm(computedPlainText, cipherText, key.d, key.n);
        gmp_printf("m = %ZX\n",cimputedPlainText);
```

```
        if (mpz_cmp(plainText, computedPlainText) != 0) {
                gmp_printf("Main: Error: Computed plain text does not match original
plain text.\n");
                return -1;
        }

        // to-do: compute the cipher text using RSA hardware

        // to-do: compute the plain text from the cipher text using RSA hardware

        // to-do: reverse the padding scheme

        // to-do: convert the computed cipher text to an array

        printf("Main: CMoving plain text m to integer array.\n");
        mpz_export(plainTextArray, messageLength, 1, sizeof(plainTextArray[0]), 0, 0,
computedPlainText); // exports gnu mp int "computedPlainText" to integer array
"outputArray"

        // write out computed plaintext to a file called output.txt
        fileName = "output.txt";
        outputTextFile = fopen(fileName, w+); // overwrite any existing output.txt
file
        if (outputTextFile == null) {
                printf("Error: Cannot open output file.\n");
        }
        else {
                printf("Main: Writing computed plain text to file "output.txt".\n");
                for     (i=stringLength-1; i>=0; i--) {
                        fputc(plainTextString[i], outputTextFile);
                }
                plainTextString[i] = 0;
                fclose(outputTextFile);
        }

        printf("Main: Done!\n");

    mpz_clear(key.n);
        mpz_clear(key.e);
        mpz_clear(key.d);
        mpz_clear(key.p);
        mpz_clear(key.q);

        return 0;

}

//=======================
//actual functions!
//=======================

//  This is the is the function that actually spits out a prime
int generateLargePrime (int keyLength, mpz_t largePrime, int seed) {
    int i;
        double tries;  // the maximum number of tries for random number generation
        unsigned long int triesint;
    gmp_randstate_t r_state;

        tries = 1000 * log( (double)keyLength) + 1;
        triesint = (unsigned long int)tries;

    gmp_randinit_default (r_state);
    gmp_randseed_ui(r_state, seed);

        #ifdef DEBUG
```

```
        gmp_printf("Will test for primality %d times.\n", triesint);
        #endif /*DEBUG*/
    for(i = 0; i < triesint; ++i) {

                mpz_urandomb(largePrime, r_state, (unsigned long int)keyLength);

                if (isPrime(largePrime, seed) == 0) {
                        gmp_printf("Found large prime = \n%ZX\n after %d tries\n which
is %d bits long\n", largePrime, i, mpz_sizeinbase(largePrime, 2));
                        gmp_randclear(r_state);
                        return 0;
                }
/*              else {
                        gmp_printf("Failed to find large prime after %d tries!\n", i);
                }*/
        }
        gmp_randclear(r_state);
        gmp_printf("Failed to find large prime after %d tries!\n", i);
        return 1;
}

//  This is the function that tests for primality
int isPrime(mpz_ptr n, int seed) {

        //lowPrimes is all primes (sans 2, which is covered by the bitwise and
operator)
        //under 1000. taking n modulo each lowPrime allows us to remove a huge chunk
        //of composite numbers from our potential pool without resorting to Rabin-
Miller

        int i;
        mpz_t r;

        unsigned long int lowPrimes[168] ={2ul, 3ul, 5ul, 7ul, 11ul, 13ul, 17ul,
19ul, 23ul, 29ul, 31ul, 37ul, 41ul, 43ul, 47ul, 53ul, 59ul, 61ul, 67ul, 71ul, 73ul,
79ul, 83ul, 89ul,
                101ul, 103ul, 107ul, 109ul, 113ul, 127ul, 131ul, 137ul, 139ul, 149ul,
151ul, 157ul, 163ul, 167ul, 173ul, 179ul,
                181ul, 191ul, 193ul, 197ul, 199ul, 211ul, 223ul, 227ul, 229ul, 233ul,
239ul, 241ul, 251ul, 257ul, 263ul, 269ul,
                271ul, 277ul, 281ul, 283ul, 293ul, 307ul, 311ul, 313ul, 317ul, 331ul,
337ul, 347ul, 349ul, 353ul, 359ul, 367ul,
                373ul, 379ul, 383ul, 389ul, 397ul, 401ul, 409ul, 419ul, 421ul, 431ul,
433ul, 439ul, 443ul, 449ul, 457ul, 461ul,
                463ul, 467ul, 479ul, 487ul, 491ul, 499ul, 503ul, 509ul, 521ul, 523ul,
541ul, 547ul, 557ul, 563ul, 569ul, 571ul,
                577ul, 587ul, 593ul, 599ul, 601ul, 607ul, 613ul, 617ul, 619ul, 631ul,
641ul, 643ul, 647ul, 653ul, 659ul, 661ul,
                673ul, 677ul, 683ul, 691ul, 701ul, 709ul, 719ul, 727ul, 733ul, 739ul,
743ul, 751ul, 757ul, 761ul, 769ul, 773ul,
                787ul, 797ul, 809ul, 811ul, 821ul, 823ul, 827ul, 829ul, 839ul, 853ul,
857ul, 859ul, 863ul, 877ul, 881ul, 883ul,
                887ul, 907ul, 911ul, 919ul, 929ul, 937ul, 941ul, 947ul, 953ul, 967ul,
971ul, 977ul, 983ul, 991ul, 997ul};

        mpz_init(r);
        #ifdef DEBUG
        gmp_printf("isPrime: checking against lowPrime values\n", i);
        #endif /*DEBUG*/
        for (i=0; i<167; i++) {
                mpz_mod_ui(r, n, lowPrimes[i]);
                #ifdef DEBUG
                gmp_printf(" r = %ZX\n n = %ZX\n i = %X\n lowPrime = %Xul\n" , r, n,
i, lowPrimes[i]);
                #endif /*DEBUG*/
                if (mpz_cmp_ui(r, 0ul) == 0) {
```

```
                        return 1; // if a lowPrime divides evenly, then quit before
rabinMiller
                        gmp_printf(" r = %ZX\n" , r);
                }
        }
        #ifdef DEBUG
        gmp_printf("isPrime: starting Rabin - Miller test.\n", i);
        #endif /*DEBUG*/
        if (rabinMiller(n, seed) == 0) {
                mpz_clear(r);
                return 0;  // probably prime!
        }
        return 1; // definitely not prime
}


//  This is the rabin-miller algorithm for primality testing
int rabinMiller(mpz_t n, int seed) {

        mpz_t s;
        mpz_t a;
        mpz_t v;
        mpz_t n_minus_one;
        unsigned long int k, j, t, retval;
    gmp_randstate_t r2_state;
    gmp_randinit_default(r2_state);
    gmp_randseed_ui(r2_state, seed);

        mpz_init(s);
        mpz_init(a);
        mpz_init(v);
        mpz_init(n_minus_one);
        // n-1
        mpz_sub_ui(n_minus_one, n, 1ul);
        mpz_sub_ui(s, n, 1ul);
        t = 0ul;

        #ifdef DEBUG
        gmp_printf("n = %ZX\n" , n);
        #endif /*DEBUG*/

        while (mpz_even_p(s)) {
                #ifdef DEBUG
                gmp_printf("t = %d\n" , t);
                #endif /*DEBUG*/
                mpz_fdiv_q_2exp(s, s, 1);
                t += 1ul;
        }

        k = 0;  // set k and j to zero
        j = 0;

        do {
                mpz_urandomm(a, r2_state, n);
        } while(mpz_sgn(a) == 0); // generate a new random number if we got zero

        #ifdef DEBUG
        gmp_printf("Generated large random number a = %ZX\n" , a);
        gmp_printf("Calculating (a^s) mod n = v\n v = %ZX\n a = %ZX\n s = %ZX\n n =
%ZX\n" , v, a, s, n);
        #endif /*DEBUG*/

        mpz_powm(v, a, s, n);

        #ifdef DEBUG
        gmp_printf("Modular exponentiation (a^s) mod n = v\n v = %ZX\n a = %ZX\n s =
```

```
%ZX\n n = %ZX\n" , v, a, s, n);
        #endif /*DEBUG*/

        for (k=0; k<128; k++) {
                if ((mpz_cmp_ui(v, 1ul)) == 0) {
                        retval = 0; // probably a prime
                        goto exit;
                }
                for(j=0; j<t-1; j++) {
                        if (mpz_cmp(v, n_minus_one) == 0) {
                                retval = 0; // probably a prime
                                goto exit;
                        }
                        mpz_powm_ui(v, v, 2ul, n);
                }
                if (mpz_cmp(v, n_minus_one) == 0){
                        retval = 0; //probably a prime
                        goto exit;
                }
                #ifdef RM_DEBUG
                gmp_printf("k = %d\n" , k);
                #endif /*RM_DEBUG*/
                retval = 1; // not a prime,
        }
exit:
        mpz_clear(s);
        mpz_clear(a);
        mpz_clear(v);
        mpz_clear(n_minus_one);
        return retval;
}
```

Hardware Source Code

File: rsa_core_top.v

```
`timescale 1ns / 1ps

//========================================================================
//==================== Top-level block for RSA crypto core ===============
//========================================================================
//========================================================================

//========================================================================
//================================ Definitions ===========================
//========================================================================

`define IDLE              2'b00    // idle state
`define NEW_INPUT         2'b01    // new inputs A and B
`define MULTIPLY          2'b10    // multiply

//========================================================================
//============== Top-Level Module with Input/Output Declarations ========
//========================================================================

module rsa_core_top(
    input               clk,              // clock
    input               rst,              // reset
    inout    [31:0]     rsa_core_io,      // 32-bit data in/out
    input               address_strobe,   // address strobe
    input    [23:0]     address,          // address data
    output              dtack,            // dtack from FPGA
    output              data_oe,          // output enable
```

```
    output             interrupt          // interrupt output
    );

//=============================================================================
//==================== Internal State-Tracking Registers ======================
//=============================================================================

    reg  [1:0]    rsa_top_state;      // top-level state machine current state
    reg           multiply_done;      // 1 if multiply result ready
    wire          go_new_input;       // 1 if received new input command
    wire          go_multiply;        // 1 if received new multiply command
    wire          idle_state;         // 1 if we are in idle state
    wire          strobe_detect;      // flopped address strobe
    reg  [1:0]    strobe_sync;        // sync latch for strobe signal
    reg  [7:0]    count;              // counter keeps track 32-bit data slices
    reg  [7:0]    keysize;            // number of bits in the A/B input data
    reg  [31:0]   data_i;             // internal data reg
        reg  [23:0]   addr_i;             // internal address reg
        reg           dtack_i;            // dtack reg
        reg           interrupt_i;        // interrupt reg
        reg           data_oe_i;

                                          // Below registers are for CIOS
                                          //  Montgomery multiplier.
         reg [4:0]     mont_mult_state;   // Montgomery Multiplier State
         reg [1023:0]  CIOS_A;            // A residue
         reg [1023:0]  CIOS_B;            // B residue
         reg [1023:0]  CIOS_n;            // modulus
         reg [255:0]   CIOS_n_prime;      // n' for CIOS algorithm (first 256-bit
word)

        reg    CIOS_C_out_0;
        reg    CIOS_C_out_1;
        reg    CIOS_C_out_2;
        reg    CIOS_C_out_3;
        reg    CIOS_C_out_4;
        reg    CIOS_C_out_5;
        reg    CIOS_C_out_6;
        reg    CIOS_C_out_7;
        reg    CIOS_C_out_8;
        reg    CIOS_C_out_9;
        reg    CIOS_C_out_10;
        reg    CIOS_C_out_11;
        reg    CIOS_C_out_12;
        reg    CIOS_C_out_13;
        reg    CIOS_C_out_14;
        reg    CIOS_C_out_15;
        reg    CIOS_C_out_16;
        reg    CIOS_C_out_17;
        reg    CIOS_C_out_18;
        reg    CIOS_C_out_19;
        reg    CIOS_C_out_20;
        reg    CIOS_C_out_21;
        reg    CIOS_C_out_22;
        reg    CIOS_C_out_23;
        reg    CIOS_C_out_24;
        reg    CIOS_C_out_25;
        reg    CIOS_C_out_26;
        reg    CIOS_C_out_27;
        reg    CIOS_C_out_28;
        reg    CIOS_C_out_29;
        reg    CIOS_C_out_30;
        reg    CIOS_C_out_31;

        reg [511:0] CIOS_m0;
        reg [511:0] CIOS_m1;
```

```
        reg [511:0] CIOS_m2;
        reg [511:0] CIOS_m3;

        reg [511:0] CIOS_t_out_0_0;
        reg [511:0] CIOS_t_out_0_1;
        reg [511:0] CIOS_t_out_0_2;
        reg [511:0] CIOS_t_out_0_3;
        reg [511:0] CIOS_t_out_0_4;
        reg [511:0] CIOS_t_out_0_5;

        reg [511:0] CIOS_t_out_1_0;
        reg [511:0] CIOS_t_out_1_1;
        reg [511:0] CIOS_t_out_1_2;
        reg [511:0] CIOS_t_out_1_3;
        reg [511:0] CIOS_t_out_1_4;
        reg [511:0] CIOS_t_out_1_5;

        reg [511:0] CIOS_t_out_2_0;
        reg [511:0] CIOS_t_out_2_1;
        reg [511:0] CIOS_t_out_2_2;
        reg [511:0] CIOS_t_out_2_3;
        reg [511:0] CIOS_t_out_2_4;
        reg [511:0] CIOS_t_out_2_5;

        reg [511:0] CIOS_t_out_3_0;
        reg [511:0] CIOS_t_out_3_1;
        reg [511:0] CIOS_t_out_3_2;
        reg [511:0] CIOS_t_out_3_3;
        reg [511:0] CIOS_t_out_3_4;
        reg [511:0] CIOS_t_out_3_5;

        reg [511:0] CIOS_t_out_4_0;
        reg [511:0] CIOS_t_out_4_1;
        reg [511:0] CIOS_t_out_4_2;
        reg [511:0] CIOS_t_out_4_3;
        reg [511:0] CIOS_t_out_4_4;
        reg [511:0] CIOS_t_out_4_5;

        reg [511:0] CIOS_t_out_5_0;
        reg [511:0] CIOS_t_out_5_1;
        reg [511:0] CIOS_t_out_5_2;
        reg [511:0] CIOS_t_out_5_3;
        reg [511:0] CIOS_t_out_5_4;
        reg [511:0] CIOS_t_out_5_5;

        reg [511:0] CIOS_t_out_6_0;
        reg [511:0] CIOS_t_out_6_1;
        reg [511:0] CIOS_t_out_6_2;
        reg [511:0] CIOS_t_out_6_3;
        reg [511:0] CIOS_t_out_6_4;
        reg [511:0] CIOS_t_out_6_5;


         reg [1023:0]  result;             // final result goes here
                                           // result = (t(3),t(2),t(1),t(0))

//================================================================================
//================================ Architecture ==================================
//================================================================================

    // muxes to interpret mmio writes as commands, ensure we only switch to
    // a new state if we are in idle state, and flop address strobe

        // only accept commands if in idle state
    assign idle_state = (rsa_top_state == `IDLE) ? 1 : 0;
```

```
    // detect new key command
assign go_new_input = idle_state ? strobe_detect ?
                    (address == 24'b00000000_00000000_00000000) ?
                    1 : 0 : 0 : 0;


    // detect encrypt command
assign go_multiply = idle_state ? strobe_detect ?
                    (address == 24'b00000000_00000000_00000010) ?
                    1 : 0 : 0 :0;

   assign dtack = dtack_i;
   assign interrupt = interrupt_i;
   assign data_oe = data_oe_i;

wire [31:0] rsa_top_io = strobe_detect ? data_i : 32'bz;



   // wire in the j-loop hardware, the multiplier and the adder
reg [255:0]  A_1;
reg [255:0]  B_1;
reg [255:0]  t_1;
reg [255:0]  n_1;
reg [255:0]  n_prime1;
reg          C_in_1;
wire [511:0] S_1;
wire         C_out_1;

reg [255:0]  A_2;
reg [255:0]  B_2;
reg [255:0]  n_2;
reg [255:0]  n_prime2;
reg          C_in_2;
wire [511:0] S_2;
wire         C_out_2;

reg  [255:0] Mult_A;
reg  [255:0] Mult_B;
wire [511:0] Mult_Y;
wire         Mult_C_out;

reg [255:0]  Add_A;
reg [255:0]  Add_B;
wire [511:0] Add_Y;
wire         Add_C_out;

j_loop j_loop_1(
   .clk(clk),
   .rst(rst),
   .t_1(t),
   .A_1(A),
   .B_1(B),
   .C_in_1(C_in),
          .S_1(S),
          .C_out_1(C_out)
   );

   j_loop j_loop_2(
   .clk(clk),
   .rst(rst),
   .t_2(t),
   .A_2(A),
   .B_2(B),
   .C_in_2(C_in),
          .S_2(S),
```

```
            .C_out_2(C_out)
        );

        mult mult1(
        .clk(clk),
        .rst(rst),

        .Mult_A(A),
        .Mult_B(B),
                .Mult_Y(S),
                .Mut_C_out(C_out)
        );

        add add1(
        .clk(clk),
        .rst(rst),

        .Add_A(A),
        .Add_B(B),
                .Add_Y(S),
                .Add_C_out(C_out)
        );




//================================================================================
//============================ Address Strobe Sync ===============================
//================================================================================

always @ (negedge clk or negedge rst) begin
    if(!rst) begin
        strobe_sync[1:0] <= 2'b0;
    end
    else if(rst) begin
        strobe_sync[0] <= strobe_sync[1];
        strobe_sync[1] <= address_strobe;
    end
end

assign strobe_detect = strobe_sync[0] && strobe_sync[1] && address_strobe;

//================================================================================
//================================ State Machine =================================
//================================================================================

always @ (posedge clk or negedge rst) begin

    if (!rst) begin            // asynchronous active-low reset
         rsa_top_state      <= 0;
         valid_input        <= 0;
         multiply_done      <= 0;
                interrupt_i        <= 0;
                count              <= 0;
                CIOS_A             <= 0;
            CIOS_B             <= 0;
            CIOS_n             <= 0;
            CIOS_n_prime       <= 0;
                data_oe_i          <= 0;
    end
    else begin
```

```
//======================================================================
// IDLE state ==========================================================
//======================================================================
         if (rsa_top_state == `IDLE) begin
             if (go_new_input == 1) begin
                         rsa_top_state    <= `NEW_INPUT;
                     end
                     else if (go_multiply == 1) begin
                         rsa_top_state    <= `MULTIPLY;
                     end
end
//======================================================================
// NEW_INPUT state =====================================================
//======================================================================

if (rsa_top_state == `NEW_INPUT) begin
    // If we just entered this state, un-set valid input, increment count
    if (count == 0) begin
        valid_input <=0;
                            count <= count + 1;
                end
                    // Now read in the inputs, 32 bits at a time.
                    // This part could be generate statement but
                    // ran into synthesis issues, so replace it
                    // with brute force hard coding.

                    // Start with values for A-residue
                else if (count == 1) begin
        CIOS_A[31:0] <= rsa_top_io;
                            count <= count + 1;
                end
                    else if (count == 2) begin
        CIOS_A[63:32] <= rsa_top_io;
                            count <= count + 1;
                end
                    else if (count == 3) begin
        CIOS_A[95:64] <= rsa_top_io;
                            count <= count + 1;
                end
                    else if (count == 4) begin
        CIOS_A[127:96] <= rsa_top_io;
                            count <= count + 1;
                end
                    else if (count == 5) begin
        CIOS_A[159:128] <= rsa_top_io;
                            count <= count + 1;
                end
                    else if (count == 6) begin
        CIOS_A[191:160] <= rsa_top_io;
                            count <= count + 1;
                end
                    else if (count == 7) begin
        CIOS_A[223:192] <= rsa_top_io;
                            count <= count + 1;
                end
                    else if (count == 8) begin
        CIOS_A[255:224] <= rsa_top_io;
                            count <= count + 1;
                end
                    else if (count == 9) begin
        CIOS_A[287:256] <= rsa_top_io;


======================================================================
     REPETITIVE CODE REMOVED — CONTACT THE AUTHOR FOR FULL SOURCE
                SKIP TO TRANSITION TO MULTIPLY STATE
```

```
   At point we've already read in A, B, n' and this is the end of n
=========================================================================



                end
                    else if (count == 103) begin
        CIOS_n[991:960] <= rsa_top_io;
                             count <= count + 1;
                end
                    else if (count == 104) begin
        CIOS_n[1023:992] <= rsa_top_io;
                             count <= count + 1;
                end
                    else begin
                        count <= 0;
                             valid_input <= 1;
                             rsa_top_state <= `IDLE;
                    end
end


//=======================================================================
 // MULTIPLY state ======================================================
 //=======================================================================

 if (rsa_top_state == `MULTIPLY) begin
     if (count == 0) begin
                        multiply_done <= 0;
                            mont_mult_state <=0;
                    end
                    else if(count == 1) begin
                    // This is a modular multiply by CIOS
                            if (mont_mult_state == 0) begin
                            // Mont mult state 0

                                    // j-loop
                                    if (~toggle) begin
                                    A_1    <= CIOS_A[255:0];
                                    B_1    <= CIOS_B[255:0];
                                    t_1    <= 0;
                                    C_in_1 <= 0;
                                    end else begin
                                    CIOS_C_out_0 <= C_out_1;
                                    CIOS_t_out_0_0 <= S_1;
                                    toggle <= 0;
                                    end

                                    if (~toggle) begin
                                    toggle <= 1;
                                    end else begin
                                    mont_mult_state <= mont_mult_state + 1;
                                    toggle <= 0;
                                    end

                                end else if (mont_mult_state == 1) begin
                                // Mont mult state 1

                                    // j-loop
                                    if (~toggle) begin
                                    A_1    <= CIOS_A[511:256];
                                    B_1    <= CIOS_B[255:0];
                                    t_1    <= 0;
                                    C_in_1 <= CIOS_C_out_0;
                                    end else begin
                                    CIOS_C_out_1 <= C_out_1;
```

```
                                        CIOS_t_out_0_1 <= S_1;
                                        end

                                        // multiply
                                        if (~toggle) begin
                                        Mult_A    <= CIOS_t_out_0_0;
                                        Mult_B    <= CIOS_n_prime;
                                        end else begin
                                        CIOS_m0 <= M_Y;
                                        end

                                        if (~toggle) begin
                                        toggle <= 1;
                                        end else begin
                                        mont_mult_state <= mont_mult_state + 1;
                                        toggle <= 0;
                                        end

                                    end else if (mont_mult_state == 2) begin
                                    // Mont mult state 2

                                        // j-loop
                                        if (~toggle) begin
                                        A_1    <= CIOS_A[767:511];
                                        B_1    <= CIOS_B[255:0];
                                        t_1    <= 0;
                                        C_in_1 <= CIOS_C_out_1;
                                        end else begin
                                        CIOS_C_out_2 <= C_out_1;
                                        CIOS_t_out_0_2 <= S_1;
                                        end

                                        // j-loop
                                        if (~toggle) begin
                                        A_2    <= CIOS_m0;
                                        B_2    <= CIOS_n[255:000];
                                        t_2    <= CIOS_t_out_0_0;
                                        C_in_2 <= 0;
                                        end else begin
                                        CIOS_C_out_3 <= C_out_2;
                                        end

                                        if (~toggle) begin
                                        toggle <= 1;
                                        end else begin
                                        mont_mult_state <= mont_mult_state + 1;
                                        toggle <= 0;
                                        end

                                    end else if (mont_mult_state == 3) begin
                                    C_in_2 <= 0;
end else if (mont_mult_state == 18) begin
                                    // Mont mult state 18


                                        // add
                                        if (~toggle) begin
                                        Add_A    <= C_out_31;
                                        Add_B    <= CIOS_B[xxx:xxx];
                                        end else begin
                                        x <= Add_Y;
                                        end

                                        if (~toggle) begin
                                        toggle <= 1;
                                        end else begin
```

```
                                        mont_mult_state <= mont_mult_state + 1;
                                        toggle <= 0;
                                        mont_mult_state <= 0;
                                        count <= count + 1;
                                        end
                                end
                            end
                            interrupt_i <= 1'b1;
                            data_oe_i <= 1;
        end
                            // Now read out the final result block, 32 bits at a
time.
                        else if (count == 2) begin
            data_i <= result[31:0];
                                count <= count + 1;
                        end
                            else if (count == 3) begin
            data_i <= result[63:32];
                                count <= count + 1;
                        end
                            else if (count == 4) begin
            data_i <= result[95:64];
                                count <= count + 1;
                        end
                            else if (count == 5) begin
            data_i <= result[127:96];
                                count <= count + 1;
                        end
                            else if (count == 6) begin
            data_i <= result[159:128];
                                count <= count + 1;
                        end
                            else if (count == 7) begin
            data_i <= result[191:160];
                                count <= count + 1;
                        end
                            else if (count == 8) begin
            data_i <= result[223:192];
                                count <= count + 1;
                        end
                            else if (count == 9) begin
            data_i <= result[255:224];
                                count <= count + 1;


                                        CIOS_C_out_3 <= C_out_2;


                                        mont_mult_state <= mont_mult_s


    ========================================================================
         REPETITIVE CODE REMOVED – CONTACT THE AUTHOR FOR FULL SOURCE
                      SKIP TO TRANSITION TO MULTIPLY STATE
       At point we've already read in A, B, n' and this is the end of n
    ========================================================================

                                        //mont_mult_state <= mont_mult_state +
1;
                                    end else if (mont_mult_state == 18) begin
                                    // Mont mult state 18


                                        // add
                                        if (~toggle) begin
```

```
                                        Add_A    <= C_out_31;
                                        Add_B    <= t_out_6_4;
                                        end else begin
                                        result[1023:768] <= Add_Y;
                                        end

                                        if (~toggle) begin
                                        toggle <= 1;
                                        end else begin
                                        mont_mult_state <= mont_mult_state + 1;
                                        toggle <= 0;
                                        mont_mult_state <= 0;
                                        count <= count + 1;
                                        end
                                    end
                                end
                                interrupt_i <= 1'b1;
                                data_oe_i <= 1;
            end
                                // Now read out the final result block, 32 bits at a
time.
                            else if (count == 2) begin
                data_i <= result[31:0];
                                    count <= count + 1;
                            end
                                else if (count == 3) begin
                data_i <= result[63:32];
                                    count <= count + 1;
                            end
                                else if (count == 4) begin
                data_i <= result[95:64];
                                    count <= count + 1;
                            end
                                else if (count == 5) begin
                data_i <= result[127:96];
                                    count <= count + 1;



                            end
                                else if (count == 33) begin

        ========================================================================
             REPETITIVE CODE REMOVED – CONTACT THE AUTHOR FOR FULL SOURCE
                          SKIP TO TRANSITION TO MULTIPLY STATE
           At point we've already read in A, B, n' and this is the end of n
        ========================================================================

                data_i <= result[1023:992];
                                    count <= count + 1;
                            end
                                else begin
                                    count <= 0;
                                        multiply_done <= 1;
                                        rsa_top_state <= `IDLE;
                                        data_oe_i <= 0;
                                end
        end
end
endmodule
```

References

[1] R.L. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Kcy Cryptosystems", Communications of the ACM, 21(2): pp. *120-l26*, February 1978.

[2] The Learning Labs. TLL5000 Electronic System Design Base Module UserGuide. Version 1.3. 2008.

[3] Xilinx. Spartan-3 Generation FPGA User Guide. Revision 1.7. January 2010.

[4] The Learning Labs. TLL6219 Embedded Systems Design Module User Guide.Version 2.0. 2008.

[5] Xilinx. ISE Design Suite Software Manuals and Help UG681 (v 11.4) December 2, 2009

[6] N Furguson and B. Shneier, *Practical Cryptography*, New York : Chichester : Wiley, 2003.

[7] RSA Security. (2003) "TWIRL and RSA Key Size" [online]. Available: http://www.rsa.com/rsalabs/node.asp?id=2004

[8] P. Barrett. "Implementing the Rivest, Shamir and Adleman public- key encryption algorithm on a standard digital signal processor," in, Advances in Cryptology – CRYPTO '86 Proceedings, vol. 263 of *Lecture Notes in Computer Science*, pp *311–323*, Springer-Verlag, 1987.

[9] P. Montgomery. Modular multiplication without trial division. Mathematics of Computation, 44(170) pp. 519–521, 1985.

[10] C.D. Walter, "Montgomery Exponentiation Needs no Final Subtractions", Elecrronics Lerrers, 35(21):1831- 1832,October 1999.

[11] C.K.Koc , T Acar., B.S. Kaliski,: "Analyzing and Comparing Montgomery Multiplication Algorithms". IEEE Micro, Vol. 16, No. 3, pp. 26-33, June 1996.

[12] C. McIvor, M. McLoone, J. V. McCanny, "FPGA Montgomery Multiplier Architectures – A Comparison," in *Proceedings of 12th Annuual Symposium on Field-Programmable Custom Computing Machines*, 2004 IEEE, pp. *279-282*.

[13] M. McLoone, C. Mclvor, J. V. McCanny, Coarsely Integrated Operand Scanning (CIOS) Architecture For High-speed Montgomery Modular Multiplication," in *Proceedings of International Conference on Field-Programmable Technology*, 2004, pp *185 – 191*.

[14] Free Software Foundation. (2010) "GNU-MP Manual" [online] Available: http://gmplib.org/manual/

Vita

Brooks Colin Gillmore is a 2005 graduate of the University of Texas at Austin, where he earned a B.S. in Physics. He has worked in semiconductor process engineering at Advanced Micro Devices and Samsung and now works for Intel as a component design engineer.

Permanent email: brooks.gillmore@gmail.com

This report was typed by the author.