RSA Box Design Document

Team members:
Adam Incera (aji2112)
Jaykar Nayeck (jan2150)
Emily Pakulski (enp2111)
Noah Stebbins (nes2137)

1. Introduction

RSA Box is a hardware accelerator for the widely used RSA encryption algorithm. To take advantage of the improved speed that hardware can provide, we'd like as much of the computation as possible to be carried out at the hardware level. In other words, we'd like the hardware to manage all the encryption and decryption.

The original *p* and *q* values are generated (but not saved) at the software layer, since we would like to offer the user the possibility of entering their own private keys and automatically generating private keys requires a highly involved, probabilistic algorithm. The keys are only stored in hardware registers -- this way, there is no interface at neither the software nor the hardware level through which to extract the private key. Only the public key is accessible through the API. Furthermore, we store both our own private key (for decryption) and the remote box's public key (for encryption) in registers at the hardware level so that we can access them as quickly as possible.

Figure 1 shows a high-level block diagram of our design, explained below. This box diagram is preliminary -- the selection of operators will likely grow as we run efficiency tests on different possible algorithms.

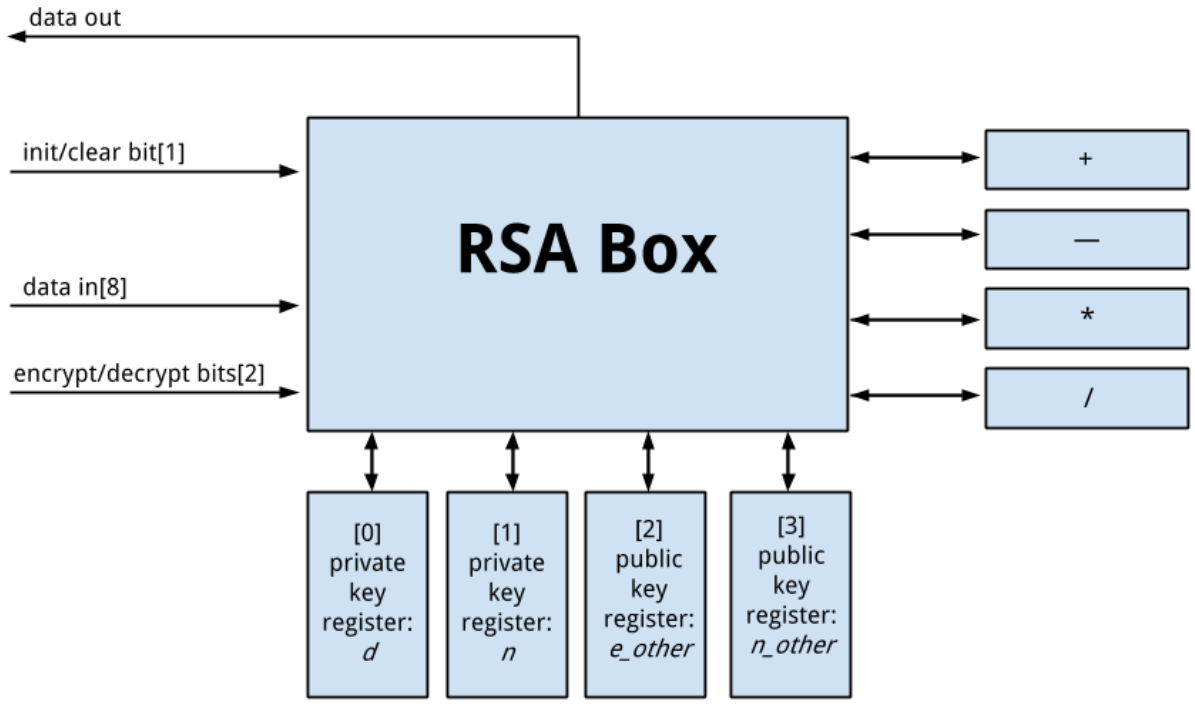

Figure 1

To initiate or end an RSA session, we flip and hold the value of the init/clear bit to 1 and 0, respectively. In other words, the RSA Box will automatically turn off and clear registers if the init/clear bit is not set to high.

Furthermore, the encrypt/decrypt bit determines what to do with the data on data in. If the encrypt/decrypt bits are set to 0, 1, 2, and 3, the box reads off data in and writes into the corresponding register.

If the values are already set, the bit can be set to either 0 or 1 to determine whether the data in is ciphertext to be decrypted or a message to be encrypted.

Data is inputted and outputted serially -- one encrypted or decrypted character at a time -- on data out.

Reusing the encrypt/decrypt bit for different purposes and limiting the number of I/Os allows us to keep the API simple. This minimalist API ensures that it is not possible to access private data from the hardware.

The arithmetic operation modules on the far right of Figure 1 are specialized modules for operations on wide bit-length numbers. For example, the multiplication module leverages the Karatsuba algorithm to efficiently split up the bits in a number before multiplying it in parts and putting it back together.

2. Hardware

2.1 Overview:

As we explained in our proposal, the 3 major algorithmic operations that need to be carried out at the hardware level are are:

1. Generating keys
    - Compute the multiple of two large primes, yielding n ($n = pq$)
    - (See Section 2.2.1) Find a positive integer that is less than $(p-1)(q-1)$ and is coprime with $(p-1)(q-1)$, a value called e
    - (See Section 2.2.2) Determine d, the multiplicative inverse of $e \pmod{(p-1)(q-1)}$
2. Encrypting and decrypting messages
    - (See Section 2.1.4)

After completing these steps, we store p, q, and d as the private key and publish n and e as the public key.

Section 2.2 (*High-Level Subroutines)* shows implementations of all the algorithms required to compute the values for RSA. These implementations in turn rely on operators that are usually straight-forward to invoke on an FPGA:

1. addition (+),
2. subtraction (-),
3. multiplication (*),
4. division (/),
5. modulo (%).

However, RSA requires invoking these operators on very large bit-length numbers, which adds a layer of complication. Section 2.3 (*Hardware Implementations of Arithmetic Operations*) explains how we intend to efficiently implement these operators on an FPGA for RSA.

2.2 High-Level Subroutines:

The following code snippets provide tested Python implementations of each of the standardized algorithms we intend to use.

2.2.1 Computing *e*

e can be any integer that is coprime to $\Phi(n)$. One way to simplify selection of a value of e is to make e prime. That way, in order to test if it is coprime to $\Phi(n)$, we only need to check if e divides $\Phi(n)$, since we already know that e and $\Phi(n)$ do not have any prime factors. In other words, if $\Phi(n)$ % e returns a non-zero value, then e and $\Phi(n)$ are coprime, and we have found a suitable value of e. We can have a series of prime numbers that we will test against the value of $\Phi(n)$, and use the first one that returns successfully.

The following Python code generates a value for *e*:

```
def generate_e(randomNum1, randomNum2):
	# program takes in two large random numbers
	relative_prime_cap = (randomNum1 - 1) * (randomNum2 - 1);

	# set the value of e
	e = -1

	for i in range(1, relative_prime_cap):
		if relPrimeCap % i != 0:
			e = i
			break
```

2.2.2 Extended Euclidean Algorithm (Computing *d*)

d forms part of the private key, which is computed with e and . $\Phi(n)$ is the number of integers between 0 and *n* that are relatively prime to *n*. We can use this and *e* to calculate the private key component, *d*, by invoking the Extended Euclidean algorithm.

The following Python code executes the Extended Euclidean algorithm:

```
# Python for calculating multiplicative inverse
# adapted from pseudocode for Extended Euclidean Algorithm
# at http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Modular_integers

def inverse(e, phi_n):
    d = 0
    new_d = 1
    r = phi_n
    new_r = e

    while new_r is not 0:
        quotient = r / new_r
        (d, new_d) = (new_d, d - quotient * new_d)
        (r, new_r) = (new_r, r - quotient * new_r)

    if r > 1:
            return -1 # error

    if d < 0:
        d = d + phi_n

    return d
```

2.2.3 Encryption and Decryption

Encryption and decryption are mathematically equivalent operations, except that the exponent used in the calculation comes from a different register. The mathematical formulae are:

- encryption: `cypher = input^(e) mod (pq)`, where input is the message

- decryption: `message = input^(d) mod (pq)`, where input is the cypher

The hardware implementation of ensuring that we read from the right register given a bit is trivial (we can simply use one AND and one NAND gate). Given their equivalency, we will discuss the two subroutines -- encryption and decryption -- in the same terms.

The following Python code represents the encryption/decryption step in RSA:

```
def encrypt(base, exponent, N):
       # c = m^e (mod n) if encrypting
       # m = c^d (mod n) if decrypting
       # ^^ in terms of params, both correspond to base^exponent (mod N)

       result = 1
       while exponent > 0:

         if exponent % 2 == 1:
             result = (result * base) % N

         exponent = exponent >> 1
         base = (base * base) % N

      return result;
```

2.3 Hardware Implementation of Arithmetic Operations

In order to fully abstract away the complications of wide-bit binary numbers, we need to create Quartus modules for each of the operators described in the Overview.

2.3.1 Addition Module

Addition of 2 digits of *n* and *m* bit length respectively yields a sum of maximum bit length *max(n, m) + 1*. Implementing this should be fairly straight-forward -- we can use a look-ahead adder to implement addition of a wide-bit number in as few cycles as possible.

2.3.2 Subtraction Module

This is analogous to the addition module, but in reverse.

2.3.3 Multiplication

For multiplication, we can use the Karatsuba algorithm. Karatsuba is an exceptionally fast algorithm for multiplying wide-bit numbers. It is a way to get past having to multiply numbers in n additions and multiplications, which the *grade school* algorithm uses and instead performs multiplication in 3 additions and 3 multiplications.

The implementation below assumes a 36 bit length number, and uses a black box function that multiplies 18 bit numbers, to demonstrate how to multiply the 36 bit numbers. We used 18 bits because the specialized multiplier blocks inside the Cyclone V FPGA are 18 * 18 bits.

The Karatsuba implementation below is simply a proof-of-concept: even though we are using 36 bits, we can arbitrarily extend it. For example, if we wanted to multiply 72x72 bit numbers, we could mimic the same logic but call our function to multiply 36x36 bits for a 72x72 bit multiplication. This implementation only uses bit shifts, additions, and the black box multiplication, simulating what would be available to us on the FPGA.

```
def multiply_18x18(a, b):
	return long(long(a) * long(b))

def multiply_36x36(a, b):
	count = long(0)
	nbits = 18

	# By &ing with the first nbits, we get the low bit values.  this number can be
hardcoded in verilog.
	for i in xrange(nbits):
	    count = count << 1 | 1

	#getting first half of bits and last half of bits this will probably be much
easier in verilog, because we can simply split the input wire
	A_high = long(a >> nbits)
	A_low = long(a & count)
	B_high = long(b >> nbits)
	B_low = long(b & count)

	#3 multiplications and 3 additions
	temp1 = multiply_18x18(A_high, B_high)
	temp3 = multiply_18x18(A_low, B_low)
	temp4 = long(multiply_18x18(long(A_high + A_low), long(B_high + B_low)) - temp1
- temp3 )
	return long( (temp1 << (nbits<<1)) + temp3 + (temp4 << (nbits)) )
```

2.3.4 Division

For division, we can use a variation of the MegaWizard function LPM_DIVIDE, which calculates the reciprocal of a value and supports bit widths in the denominator of up to 64 (https://www.altera.com.cn/zh_CN/pdfs/literature/ug/ug_lpm_divide_mf.pdf), and then use the efficient multiplication aglorithm described above to multiply with the reciprocal.

The Newton Raphson Method (http://en.wikipedia.org/wiki/Division_algorithm) implements this approach highly efficiently:

```
	# Express D as M × 2e where 1 ≤ M < 2 (standard floating point
	representation)
	D' = D / 2e+1   # scale between 0.5 and 1, can be performed with bit
	shift / exponent subtraction
	N' = N / 2e+1
	X = 48/17 - 32/17 * D'   # precompute consts with same precis as D
	repeat time
```

```
    X := X + X × (1 − D' × X)
    end
return N' × X
```

2.3.5 Modulo

The modulo operator can be implemented either using division, multiplication, and a floor function or by repeatedly subtracting one value from another (stackoverflow.com/questions/2660997/):

- *Option 1:* `a % b = a - (b * floor(a / b))`
- *Option 2:* using repeated subtraction

```
def mod(a, b):
      remainder = a;
      while (b < temp):
            remainder = a - b
      return remainder
```

Since looping operations would be costly in hardware, we chose to implement Option 1. Option 1 requires us to implement a floor function.

3. Software

The software layer exists to simplify programmer usage of RSA Box, as well as to prove that the hardware functions correctly. As such, the sections below describe the following programs:

1. a **driver** to connect to the RSA Box,
2. a **C library** that exposes a simple API and generates the seed values,
3. a **demo chat application** to provide an elegant demonstration that the RSA Box platform is working and ready to be used by other developers.

3.1 Driver

We'd like to build a kernel-space driver that connects to the RSA box that supports the following functionality:

1. Request a new public key,
2. Express individual ASCII characters as 8-bit wide values that are sent to the RSA and return them in a 128-bit wide encrypted form.

We would write a kernel module similar to what we wrote in lab 3.

3.2 C Library Wrapper

The C library makes it easy for an application programmer to use RSAB. The programmer can initiate a session -- that is, generate a unique public key with a unique hidden private key -- and then encrypt or decrypt messages.

The interface that the C library would expose looks as follows:

```
// Start session.
// Use user-specified private key, return public key.
long RSA_init(long p, long q);

// Start session.
// Use auto-generated private key, return public key.
long RSA_init();

// Save remote keys to register.
void set_remote_keys(long e_other, long n_other);

// Encryption and decryption using values stored in registers.
// Raise exception and set errno if relevant register not set.
char *encrypt(char *msg);
char *decrypt(char *cypher);

// End session and clear registers.
void RSA_end();
```

3.3 Demo Chat Application

Finally, we'll implement a simple chat client to demonstrate the functionality. The chat client will allow two users to enter an IP address and a port to connect to in order to initiate a chat session. The users will then be able to chat normally, but under the hood, none of the transmitted data will be sent as cleartext.