ECE 5745 Final Project Report: Rapid Security Algorithms in Hardware

Aidan McNay (acm289)

1 Introduction

In today's growing network of computational devices, we need to make sure that our security can appropriately scale as well. When we are transmitting data at ever-faster speeds, the demand on the speed of encryption and decryption of the data grows as well to avoid becoming the bottleneck. Security algorithms for our data exist, but often involve complicated arithmetic that is inefficient to perform on general-purpose computing devices. In this lab, we will explore the RSA (Rivest-Shamir-Adleman) cryptography algorithm implemented in application-specific hardware. The RSA algorithm is an asymmetric public-private key algorithm that has been around for many years, and has not been found to be overly vulnerable to any known attacks, making it a common security algorithm that can be found across the Internet, in emails, and in credit card payments [1]. Because of its commonplace, it is critical that RSA encryption and decryption can be computed relatively quickly to avoid slowing down the communication that it empowers. We wish to explore an 32-bit RSA encryption-decryption accelerator that, when coupled with our processor from Lab 2, will significantly speed up the computations involved in encrypting and decrypting messages over RSA.

1.1 Literature Review

When reviewing the literature surrounding the topic, we first looked to the defining RSA paper that published the algorithm [5]. This not only provided details on the algorithm and its theory, but gave starting implementations of the algorithm's components, with the primary one being the modular exponentiation involved in encryption and decryption. It also provided some of the reasoning behind the security of the algorithm, and examined different ways that others might go about breaking it. While they weren't sure whether the algorithm would stand up to the test of time, later reflections confirm that the algorithm still has a high level of security [1].

From here, we delved into some of the more detailed algorithms for the arithmetic involved in modular exponentiation, as they are non-trivial to do in hardware (specifically, multiplication under a modulus). Since the introduction of RSA, a different algorithm has been put forth to solve this issue, using "Montgomery multiplication" (with the namesake coming from the author of the defining paper). It defines the 'residue' of a number, and shows how we can use a modulus over a number that is easy to compute (such as a power of two in binary) to obtain a modulus that is hard to compute (such as our desired one) without the need for division. The author notes that the transformation to the modulus-space and residue formats requires non-trivial effort, as well as some key parameters, such as R^{-1}, which can be obtained using the Extended Euclidean algorithm - an algorithm typically seen in RSA key generation, but which can also be used in Montgomery multiplication. However, given that we are often computing over the same modulus multiple times in our algorithm, we can amortize this cost of conversion over multiple runs in an attempt to obtain speedup.

Lastly, our literature research included some tips and tricks for making our hardware run fast, including defining a common value for the key parameter e in public keys (such as 3, 17, or 65537 [6]), as well as making it prime. It also gave some useful proofs about our results, specifically that when calculating the GCD (used not only in key generation, but in Montgomery multiplication), our values at each step would be less than our operands [8], ensuring that we won't run into overflow errors in hardware. These guarantees help to address some of the concerns that may have arisen with low-level algorithms, and streamline the overall design process.

In regards to Montgomery multiplication, while Montgomery leaves some comments on how one might go about fast computations at the end of his paper [4], these are expanded upon in Ko's paper on their implementation of hardware [2]. This paper delves into many aspects of how one might go about implementing RSA algorithms in hardware. While some are too low-level for the scope of this project (such as the implementation of adder circuits), many high-level ideas are useful when determining how various steps are implemented with a modulus, a space where many hardware designers may be unfamiliar (such as modular

addition and multiplication). Specifically, the paper expands on how one might go about interleaving the multiplication and reduction steps involved in Montgomery multiplication in hardware, allowing for speedup.

1.2 Help Received

The biggest help to this project would be Professor Batten. Without his guidance and understanding of steps needing to be taken, this project would likely not have come close to fruition. I also received help from the ECE 5745 Lab 2 baseline design for a RISCV processor coupled to a hardware accelerator, which served as a strong starting point for the baseline design of my system. Finally, I received help from the RSA Python library. This library provided a functional-level model for our desired algorithms, allowing us to verify their functionality and integration with external RSA references.

2 Baseline Design

For our baseline designs, we implemented the RSA encryption/decryption algorithm in C, so that it may be cross-compiled and run on our processor from Lab 2. Specifically, we used the algorithm as described in the defining paper for RSA published in 1978 [5], including the algorithms mentioned for computing modular exponentiation [6]. In addition, we included a modified version that uses Montgomery multiplication [4] in an attempt to get speedup by amortizing the cost of conversion of our numbers over the faster multiplication-remainder operations. These serve to quantify how well the RSA algorithm performs (both in time and power/energy) on a general-purpose computing device. We can also use the area of this device to quantify how much area overhead comes with our accelerator designs

2.1 Algorithm

In this section, we will introduce the overall algorithms and concepts that are involved with RSA. These algorithms are introduced in the abstract as to give the reader background in the subject matter, and later implemented in software in hardware.

2.1.1 Key Generation

The overall RSA algorithm involves modular exponentiation for both encryption and decryption. The key to the security revolves around the difficulty to factor large numbers (with other methods shown to be no easier [1]). Because of this, the algorithm calls for two large prime numbers p and q, and computes the key parameters n, d, and e, such that:

$$\begin{aligned} n &= p * q \\ \phi(n) &= (p-1)(q-1) && \textit{(Euler's totient function)} \\ 2 < e < \phi(n),\ &\gcd(e, \phi(n)) = 1 && \textit{(e and $\phi(n)$ are coprime)} \\ de &\equiv 1 \pmod{\phi(n)} \end{aligned} \tag{1}$$

(Note that $\equiv$ is used to indicate modular equivalency, as is common)

From here, these key factors are packaged into public and private keys. The public key consists of (n, e) (e being used for encryption), and the private key consists of (n, d) (d being used for decryption). e is often fixed at a particular value, with d being computed based on the n in our case; specifically, common choices for e are 3, 17, and 65537, as these are prime (thereby likely coprime with $\phi(n)$) and sparse (requiring fewer computations when performing modular exponentiation with e as the exponent), with it being shown that common values for e don't result in any loss in security [6]. Given that small values for e can be potentially insecure (as it may not increase past the modulus, allowing a private party to simply take the eth root of the ciphertext to decrypt it [1]), we will choose to fix e at the recommended value of 65537 (which our reference library uses as well).

Note that d is only attainable through having p and q to compute the totient. One could theoretically factor n to get these key factors, but Rivest, Shamir, and Adleman show that at their time, factoring these

large numbers at the recommended size of n would take $3.9 * 10^9$ years, making it infeasible [5]. While algorithms for factoring large numbers have improved since then (with the current fastest being the General Number Field Sieve [1]), the complication of the process combined with the increase in number of bits (with current recommendations being 2048 bits for n [3]) mean that RSA (when implemented properly) has suffered no substantial attacks [1]. While this project will tackle the encryption and decryption aspects of RSA instead of key generation (attempting to speed up the element which happens many times, as opposed to key generation, which only occurs once), knowledge of how keys are generated is useful in understanding the entire scope of the cryptographic algorithm.

2.1.2 Encryption and Decryption

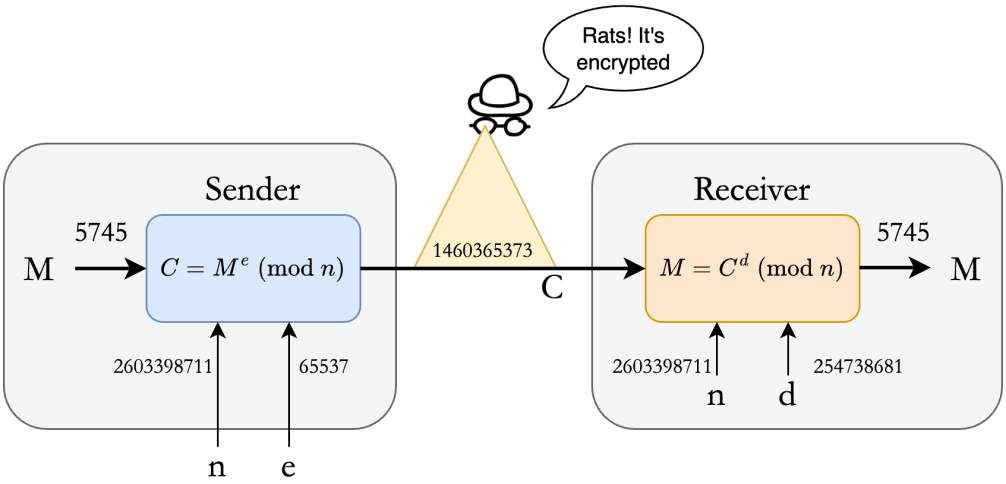


Figure 1: High-level diagram of data transfer with RSA

To understand RSA encryption and decryption, we must first introduce the following equivalency, considering our desired message as M:

$$(M^e)^d = M^{ed} \equiv M \pmod n \tag{2}$$

This was proven by Rivest, Shamir, and Adleman using our restrictions above for n, e, and d, with the additional precondition that $0 \leq M < n$ [5]. We can satisfy the left half of this equality by treating our data as unsigned, and satisfy the right half by breaking apart any pieces of data that are $\geq n$, encrypting them separately, and recombining after decryption.

From this, we can understand why our encryption and decryption methods work; encryption performs the exponentiation by e, and decryption performs exponentiation by d. This will overall yield our original message, as these operations are shown above to be symmetric. This is further layed out below, with M being our message and C our ciphertext (note that these operate on unsigned integer values, but we can use other schemes before encryption and after decryption to convert any data we have into unsigned integers, thus maintaining the generality of the algorithm):

$$\begin{array}{ll} \textbf{Encryption} & \textbf{Decryption} \\ C \equiv M^e \pmod n & M \equiv C^d \pmod n \end{array} \tag{3}$$

This overall results in our receiver obtaining the value $(M^e)^d \pmod n = M$, our desired message. This transfer of data can also be seen in the example in Figure 1; RSA guarantees us that any observer with the ciphertext C (1460365373), as well as the publicly-available n (2603398711) and e (65537), would not be able to recover M without an amount of work equivalent to factoring n, for which there is no currently viable algorithm [5]. The encryption and decryption algorithms are additionally well-suited for improvement due to having the same function; if you optimize one, you are also optimizing the other.

2.1.3 Modular Exponentiation

As seen above, the RSA cryptographic system is built on modular exponentiation. While this could be implemented naively (calculating the base raised to the exponent, then applying the modulus), this would not only result in loss of data (as with large numbers, the base raised to the exponent would likely overflow the data size), but would be incredibly slow. Instead, this is normally done through "modular exponentiation", also known as "exponentiation by squaring". This can be seen in Algorithm 1 (adapted from [6], Section 11.3) - the result is calculated with repeated squaring of an accumulated result and the base (giving it the latter name) while shifting the exponent. This is our primary baseline algorithm for implementing our RSA encryption and decryption functions. Note that this is the reason why a sparse e is chosen; we only have to perform 17 modular multiplications, instead of possibly a much larger amount with a random $e < \phi(n)$ is used (roughly on the order of 1000 [1]).

Algorithm 1 Modular Exponentiation [6]

Ensure: $r = b^e \pmod{n}$

```
r ← 1
b ← b % n
while e > 0 do
    if e is odd then
        r ← (r * b) % n
    end if
    e ← e/2
    b ← (b * b) % n
end while
```

2.1.4 Montgomery Multiplication

In Algorithm 1, we can see that multiplication under a modulus is used frequently, and is the primary arithmetic operation in the algorithm (thereby using the most time). This operation can be slow, as division (used to compute the modulus) can often be an expensive operation. To avoid this expensive computation, an alternative "Montgomery multiplication" was introduced [4]. This type of multiplication uses a new key value R such that $R > N$, R and N are coprime, and division and modulus by R are relatively easy (where $N = n$ in relation to our previous algorithm - both are the relevant moduli) Because of this, R is often chosen to be a power of 2 so that division and modulus are simply bit-shifting and masking, respectively, as well as that it is guaranteed to be coprime to N in RSA applications (as N is the product of two large primes). Specifically, it is often chosen to be the word-line size as to guarantee being larger than N [4].

Along with this, Montgomery also requires a few other constants. A natural one is R^{-1}, the inverse of R under modulus, so that:

$$RR^{-1} \equiv 1 \pmod{N} \tag{4}$$

This also indicates that RR^{-1} is equal to one more than some coefficient times N. This coefficient is given the symbol N', leading to the equation:

$$RR^{-1} - N'N = 1 \tag{5}$$

This can be solved with the extended Euclidean Algorithm to find R^{-1} and N', similar to how d was computed for our RSA keys. Note that Montgomery mandates that $0 < R^{-1} < N$ and $0 < N' < R$; while our extended Euclidean Algorithm doesn't guarantee this, we can increment R^{-1} and N' by N and R, respectively, without changing the validity of the equation, allowing us to find the coefficients in the desired range.

In addition, Montgomery introduces the idea of *N-residue* as a residue class modulo N [4]. For any value a, the corresponding residue representation is given by $\overline{a}$, where

$$\overline{a} = aR \pmod{N} \tag{6}$$

Here, we can also note that, to perform multiplication in *N-residue* format, we need to compute $\overline{c}$, the *N-residue* of $a * b$, where (using the fact that common arithmetic operations such as multiplication and addition are unchanged in *N-residue* format [4]):

$$\overline{c} \equiv (a * b)R \equiv (aR * bR)R^{-1} \equiv (\overline{a} * \overline{b})R^{-1} \tag{7}$$

While this doesn't seem to help us all that much, Montgomery additionally defines an algorithm titled *REDC(T)* for easily computing $TR^{-1} \pmod{n}$ (an iteration of this algorithm is also known as a "Montgomery Step"). The main advantage is that this algorithm avoids generic modulus and division, and instead only uses modulus and division by R, which was made to be relatively easy before. This allows us to append the multiplication step $(\overline{a} * \overline{b})$ to the algorithm to have a generic algorithm for computing $\overline{c}$.

Algorithm 2 REDC(T)[4]

Require: T such that $0 \leq T \leq RN$
Ensure: $t = TR^{-1} \pmod{n}$
1: $m \leftarrow (T \mod R)\, N' \mod R$
2: $t \leftarrow (T + mN)/R$
3: **if** $t \geq N$ **then**
4: $\quad t \leftarrow t - N$
5: **end if**

Note additionally that we can use this algorithm for conversion in and out of *N-residue* form. To convert x into residue form $\overline{x}$, use our multiplication algorithm with x and $R^2 \mod N$, the latter of which can be pre-computed in advance:

$$\overline{x} \equiv (x * R^2)R^{-1} \equiv xR \pmod{N} \tag{8}$$

To convert $\overline{x}$ out of residue form into x, simply apply the algorithm (or, if only the multiplication implementation is available, perform the multiplication implementation with $\overline{x}$ and 1):

$$x \equiv (\overline{x} * 1)R^{-1} \equiv \overline{x}R^{-1} \pmod{N} \tag{9}$$

This shows how we can have high code re-usability for conversion in and out of *N-residue* form. With this, we can modify Algorithm 1 to use our Montgomery multiplication instead:

- Convert our operands into *N-residue* format
- Perform the algorithm as intended, replacing any instances of modular multiplication with multiplication followed by the *REDC* algorithm
- Convert the final result out of *N-residue* format

This is fully realized in Algorithm 3 (Note that we don't need to take the initial modulus of b, as it is done for us when converting into *N-residue* format). Aside from the pre-calculation of $R^2 \mod N$, this entire algorithm avoids any division (or "trial division", as Montgomery refers to it), allowing it to achieve speedup.

Algorithm 3 Modular Exponentiation with Montgomery Multiplication

Ensure: $r = b^e \pmod{n}$
1: $r \leftarrow REDC(R^2 \mod N)$ ▷ 1 in *N-residue* form
2: $b \leftarrow REDC(b * R^2 \mod N)$
3: **while** $e > 0$ **do**
4: $\quad$ **if** e is odd **then**
5: $\quad\quad r \leftarrow REDC(r * b)$
6: $\quad$ **end if**
7: $\quad e \leftarrow e/2$
8: $\quad b \leftarrow REDC(b * b)$
9: **end while**
10: $r \leftarrow REDC(r)$

A single modular multiplication operation would likely not benefit from this approach, as it would require 4 Montgomery steps (2 to convert the operands into *N-residue* format, 1 to perform the multiplication, and 1 to convert the result out), which may not be faster than the division it is trying to beat. However, with modular exponentiation, the modular multiplication step is repeated many times. Given that a Montgomery step avoids division, it can achieve speedup by amortizing the cost of conversion over the many modular multiplication steps, where it performs faster than a traditional approach that uses division for the modulus. This benefit scales with the length of the exponent; while this length is fixed for encryption (as e is taken to usually be 65537), the exponent d in decryption will scale with the size used for n in RSA, allowing Montgomery multiplication to have a larger impact for RSA implementations using more bits.

For a secondary baseline implementation, we will use modular exponentiation still entirely in software, but with the use of Montgomery multiplication as described in Algorithm 3. This will serve to help us quantify the gains that can be made through software/algorithmic improvements.

2.2 Implementation

When implementing these algorithms for our processor, we went through several iterations, including ones in Python, C, as well as using a high-level Python RSA library to serve as a functional-level model. This gradual approach helps by allowing us to realize and check errors in our overall algorithm, before transition to a more concrete, hardware-oriented version with more complexities (such as integer representations) introduced.

2.2.1 Functional-Level Model

When devising our implementations, we will use a functional-level model to test against. This can not only help when testing our algorithms (making sure that we get the same ciphertext or message under the same stimuli), but can help ensure that others are able to understand our message; we can use a functional-level sender or receiver to verify that our implementation of the receiver or sender can correctly decrypt or encrypt a message, respectfully. For this, I chose to use the RSA library in Python. While this library includes many constructs for RSA encryption and decryption, these include other overheads (specifically, padding schemes for extra security) that mean that they can't directly interface with our other algorithms. However, the core of the library defines `rsa.core.encrypt_int` and `rsa.core.decrypt_int`. These functions take in our message or ciphertext (respectively), e or d (respectively), and n, and perform encryption and decryption as described above. Since this is exactly what we're implementing, this allows us to have a "golden model" for how our implementations should behave, as well as a proof-of-concept that our implementations are abiding by RSA standards (we can verify that these encryption and decryption schemes can generate or recover a message when combined with our custom implementation).

2.2.2 Python

I chose to first implement our RSA algorithms in Python (since both encryption and decryption are the same, this simply involved implementing modular exponentiation). Implementing this in Python allowed for focusing on the conceptual understanding of the algorithm first, and verifying its validity with our golden model, before beginning to worry about more technical details such as bit widths and cross-compiling. In Listing 1, we can see our baseline implementation of modular exponentiation in Python. This follows Algorithm 1 fairly directly, with the main difference being the use of modulus to check whether the exponent e is even.

In addition, we can also implement our modular exponentiation algorithm with Montgomery multiplication in Python, to verify our understanding of the nuances that it involves. This can be seen in Listing 2. For this, I decided to implement a `MontMultiplier` class in Python, to serve as a wrapper for the operations and key constants involved in Montgomery multiplication. We know from before that we need to give our multiplier the key parameters N and R; these are given to the multiplier upon instantiation in line 7. Note that for R, we use $1 << 32$; this is one more than can be represented with 32 bits (how n will be stored), and is therefore guaranteed to be larger than n (as well as being a power of 2, allowing for easy division and modulus operations). From here, our code follows Algorithm 3, with member functions of the `MontMult` object being used to implement multiplication and conversion to and from *N-residue* format.

2.2.3 C

Once we have verified our conceptual understandings in Python, we can then move to a more concrete implementation in C, taking into account bit-widths. Our operands will all be unsigned 32-bit representations (`unsigned int`), as we're working with a 32-bit implementation of RSA, which has no notion of sign. However, many of the intermediate results of our algorithm need to maintain all of the data from multiplication and addition, as to have the correct result after a modulus. To prevent data loss from this, as well as to be able to represent R (which exceeds 32-bits), many of our values are represented as 64 unsigned bits (`unsigned long long`). While our 32-bit RTL processor may not be able to operate on these directly, our compiler will take this into account and use two words for the representation of each of these values.

Listing 3 shows how we can implement our modular exponentiation algorithm in C. This is very similar to Listing 1; the main difference is the fact that we need to explicitly convert our values into `unsigned long long` in order to maintain all of our data. Another change is the fact that we mask the exponent with

Listing 1: Modular Exponentiation in Python

```
1  def mod_exp( base, exponent, modulus ):
2
3      result = 1
4      base = base % modulus
5
6      while( exponent > 0 ):
7
8          if( ( exponent % 2 ) == 1 ):
9              result = ( result * base ) % modulus
10
11         exponent = exponent >> 1
12         base = ( base * base ) % modulus
13
14     return result
```

Listing 2: Modular Exponentiation in Python with Montgomery Multiplication

```
1  def mont_mod_exp( base, exponent, modulus ):
2
3      result = 1
4      base = base % modulus
5
6      # Set up Montgomery multiplier
7      MontMult = MontMultiplier( modulus, ( 1 << 32 ) )
8
9      # Convert into N-residue format
10     result = MontMult.convert_in( result )
11     base   = MontMult.convert_in( base )
12
13     while( exponent > 0 ):
14
15         if( ( exponent % 2 ) == 1 ):
16             result = MontMult.multiply( result, base )
17
18         exponent = exponent >> 1
19         base = MontMult.multiply( base, base )
20
21     # Convert out of N-residue format
22     result = MontMult.convert_out( result )
23
24     return result
```

1 to determine if it's odd, as it obtains the same result as the modulus operation from before, but without division. We also make sure to convert our result back to an `unsigned int` upon completion; since our result is guaranteed to be less than *modulus*, which can be represented as an `unsigned int`, this won't cause any loss of data.

In addition, Listing 4 shows how we can implement our modular exponentiation algorithm with Montgomery multiplication in C. This is very similar to Listing 2, including the wrapping of the Montgomery multiplier, this time into a struct of type `montmult_t`, which is constructed in line 7 with our modulus an R. We then implement the conversion and multiplication functions as functions that take in a pointer to this wrapper, allowing them to access the internal constants. Finally, since our base and result can always be represented by an `unsigned int` (as they are always less than the modulus), we can maintain this representation in our overall `mont_mod_exp` function, with the conversion to higher bit representations occurring within our functions.

2.3 Div-Rem Unit

Our algorithm critically relies on the calculation of a modulus of two numbers. While the 32-bit computation of this would only require the remainder of the two inputs, the 64-bit unsigned-unsigned implementation using 32-bit operations (contained under <__umoddi3> as part of *libgcc* for the GCC compiler) requires both division and remainder functionality. To implement this, we had to implement a division-remainder unit, the

Listing 3: Modular Exponentiation in C

```c
unsigned int mod_exp( unsigned int base, unsigned int exponent, unsigned int modulus )
{
    // Maintain base and accumulated result as long long for precision

    unsigned long long curr_base   = ( unsigned long long ) base % modulus;
    unsigned long long curr_result = 1;
    unsigned long long curr_mod    = ( unsigned long long ) modulus;

    while( exponent > 0 )
    {
        if( ( exponent & 1 ) == 1 ) // LSB is 1
        {
            curr_result = ( curr_result * curr_base ) % curr_mod;
        }

        exponent = exponent >> 1;
        curr_base = ( curr_base * curr_base ) % curr_mod;

    }

    return ( unsigned int ) curr_result;
}
```

Listing 4: Modular Exponentiation in C with Montgomery Multiplication

```c
unsigned int mont_mod_exp( unsigned int base, unsigned int exponent, unsigned int modulus )
{
    unsigned int curr_base   = base % modulus;
    unsigned int curr_result = 1;

    montmult_t multiplier;
    montmult_construct( &multiplier, ( unsigned long long ) modulus, 0x100000000 );

    // Convert into N-residue format
    curr_base   = montmult_convert_in( &multiplier, curr_base   );
    curr_result = montmult_convert_in( &multiplier, curr_result );

    // Perform iterative exponentiation

    while( exponent > 0 )
    {
        if( ( exponent & 1 ) == 1 ) // LSB is 1
        {
            curr_result = montmult_multiply( &multiplier, curr_result, curr_base );
        }

        exponent = exponent >> 1;
        curr_base = montmult_multiply( &multiplier, curr_base, curr_base );
    }

    // Convert out of N-residue format

    curr_result = montmult_convert_out( &multiplier, curr_result );

    return ( unsigned int ) curr_result;
}
```

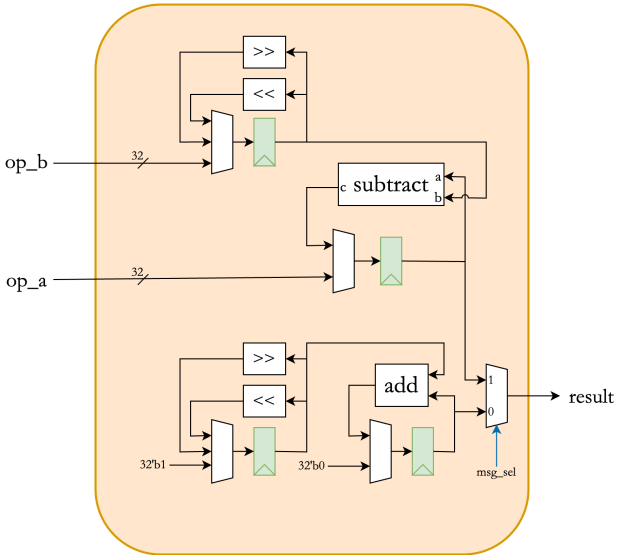


Figure 2: Division-Remainder module block diagram

block diagram for which can be seen in Figure 2 (with the control and *val/rdy* interface omitted for clarity). Here, shifted versions of `op_b` are continually subtracted from `op_a` to compute the remainder. In addition, a separate register (bottom of Figure 2 keeps track of how much we've shifted `op_b`, and adds itself to a stored value every time we subtract from `op_a`, thereby keeping track of how many instances of `op_b` were subtracted, resulting in the quotient. These two values are finally muxed by a select line, `msg_sel`, to give the desired output of the quotient (0) or remainder (1), depending on what the user wants to receive.

2.4 Modifications to Processor

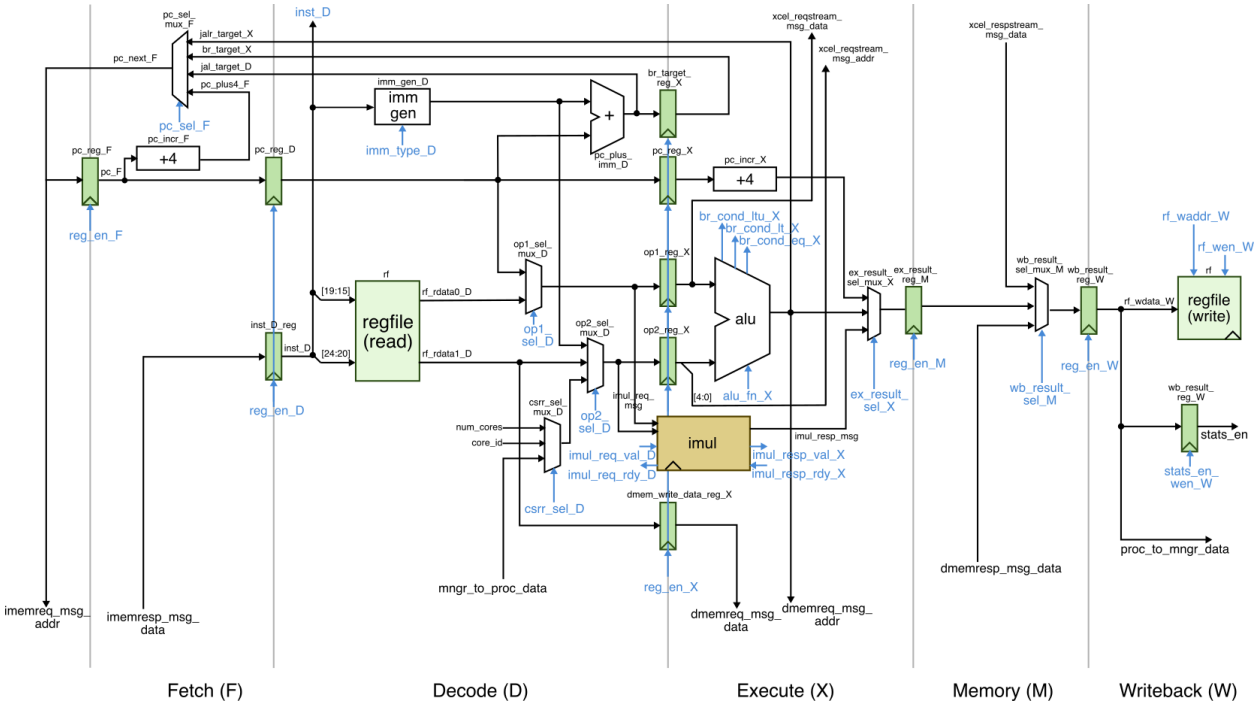


Figure 3: Initial Processor Datapath from Lab 2

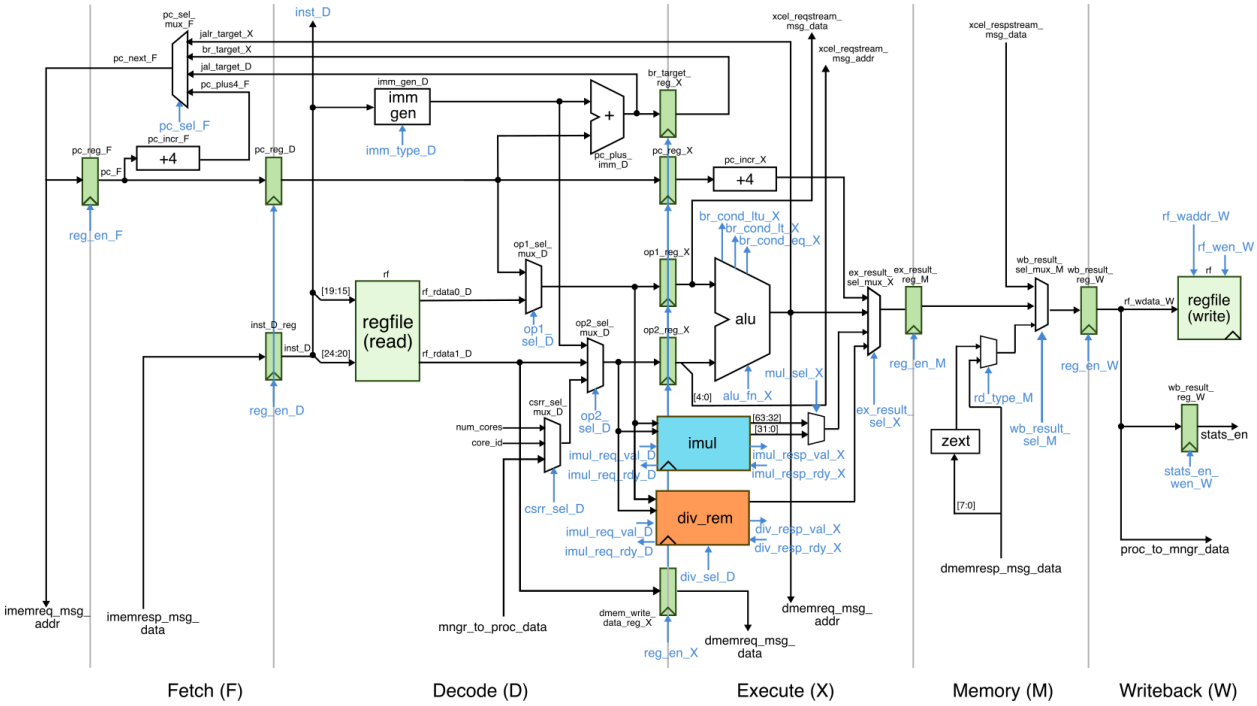


Figure 4: Modified Processor Datapath to support `divu`, `remu`, `mulhu`, and `lbu`,

Once we have implemented the above *div_rem* module, we must then modify our processor to tightly integrate it to support unsigned division and remainder operations (given in RISCV by the `divu` and `remu` instructions [7]). Since our *div_rem* unit also has a *val/rdy* interface, we can integrate it similarly to how we did with our *imul* multiplication unit. We can see the datapath diagram for our initial processor from Lab 2 in Figure 3. Comparing this to our modified datapath in Figure 4, we can see that next to the multiplier, we've inserted our *div_rem* unit similarly.

Lastly, as part of `<__umoddi3>`, our algorithm requires support for two more operations:

- `mulhu` performs a multiplication, but returns the upper 32 bits from the result (as opposed to `mul`, which truncates the top and returns the lower 32 bits). This is used to make sure we maintain all of the data from a multiplication operation; none is lost from overflow. To support this, we modified our *imul* unit to perform 64-bit multiplication, allowing us to get a 64-bit result. Our 32-bit operands are padded with 0's before going in. Once our 64-bit result comes out, the top and bottom 32 bits are muxed, selected by `mul_sel_X`, to pass on the desired section of bits

- `lbu` is similar to `lw`, but only loads a byte (not 4) from the address, and zero-extends it to 32-bits. This can be done by taking the data returned from memory in `dmemresp_msg_data`, computing the correct data for `lbu` by zero-extending the bottom byte, and muxing them based on `rd_type_M` to pass along the correct piece of data. While this relies on word-aligned accesses, this functions for our algorithm due to the use of "magic memory", and isn't the focus of the project, so it will work for our system.

Along with these, we modified our Python assembler to include support for the four new instructions according to their implementation in RISCV [7]. We modified our FL processor model in PyMTL to include these operations, and included more Verilog parameters for our processor to identify and appropriately act on these instructions. With this, our baseline implementations are complete; both naive and Montgomery implementations of RSA can successfully cross-compile and run on our RTL processor.

3 Alternative Design

For our alternative designs, we devised hardware accelerators for our RSA algorithms. Specifically, we have two alternative designs; one accelerator used our primary "naive" baseline approach as was suggested in Rivest, Shamir, and Adleman's paper [5]. This will represent the benefits that can be gained from custom hardware. Our other alternative design focuses on using the Montgomery representation of numbers to achieve speedup, representing the benefits that can be gained from combining ideas from software and hardware into one unit. By simulating these designs, as well as pushing them through the flow, we can arrive at quantitative results as to how our different systems compare. Additionally, due to the similarities in architecture, we can utilize a modular and hierarchy approach to our designs, allowing us to re-use many components from the naive approach in our Montgomery version.

3.1 Naive

Our first alternative design takes a "naive" approach, and simply looks to speed up Algorithm 1 in hardware. To do this, we first implemented a `MulRem` unit. The underlying arithmetic of this algorithm relies on multiply-remainder operations, which the `MulRem` unit is designed to accomplish. This design is simply composed of a multiplier fed into a remainder unit, as can be seen in Figure 5; the design takes in a message composed of two multiplier operands `opa` and `opb`, as well as a modulus `n`, and returns the product of the first two under the modulus of `n` (all with a latency-insensitive *val/rdy* interface, which all sequential blocks in this project will have). Our multiplier and divider both operate on 64 bits, as to avoid loss of data from the product of the two 32-bit operands. This allows us to achieve speedup over hardware; our processor only has the capability to operate on 32-bit values, and therefore must emulate 64-bit operations through many more arithmetic instructions. By utilizing custom hardware for the precision we need, we can eliminate the extra cycles and operations needed for emulation.

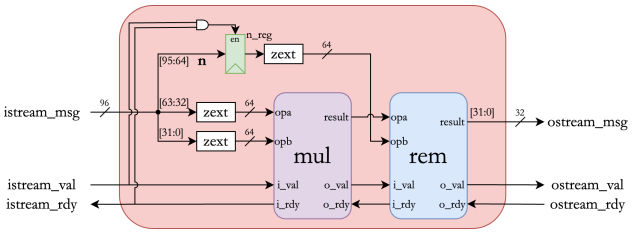


Figure 5: Block diagram of the `MulRem` unit

From here, we can instantiate this component in our `ModExp` block, seen in Figure 6. This block is designed to implement the outer loop in Algorithm 1. It takes in a message containing our base (`b`), exponent (`e`) and modulus (`n`) operands, and returns b^e mod n. Once the operands are registered, it loops through in the FSM control logic while the exponent `e` is not zero. On each iteration, it will shift `e` to the right once, update `b` with $(b*b)$ mod n, and conditionally update `r` with $(r*b)$ mod n, depending on the LSB of `e`. At the end, it will return the value stored in r. Note that this exactly follows the high-level process described in

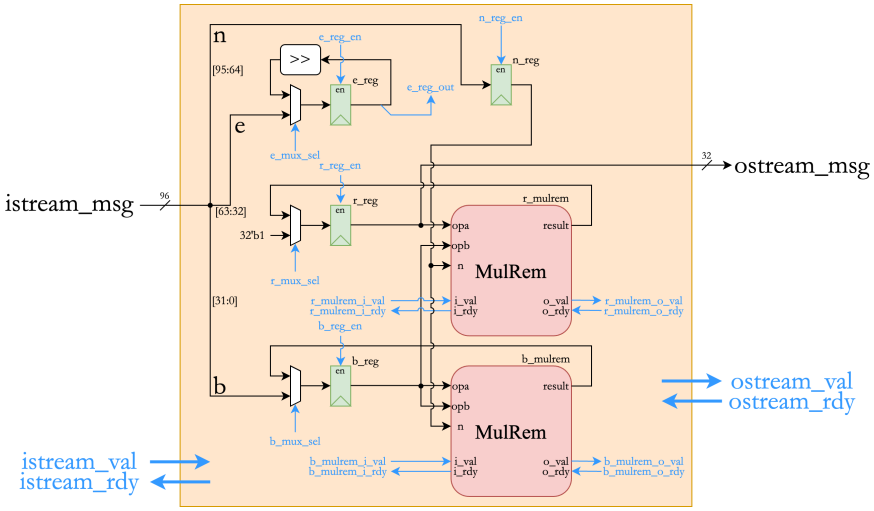


Figure 6: Block diagram of the `ModExp` unit

Algorithm 1. However, when our processor implemented this algorithm, it had to follow a linear program, and therefore had to execute the `MulRem` operations in series. By executing these operations in parallel (when two are needed to update both r and b), our custom hardware can achieve speedup compared to our general-purpose processor.

Finally, to interface our custom hardware with our processor, we must make sure it can utilize our accelerator interface from Lab 2. To achieve this, we defined a simple `XcelAdapter` block. This block takes in our operands (`b`, `e`, `n`) across the accelerator interface in serial instructions, handling the responses appropriately. When instructed to "go", it sends this data to our `ModExp` unit (with the operands being sent in parallel). In addition, when our `ModExp` unit is finished, it takes the result back, and sends it along the accelerator response when prompted by our processor. These modues are finally coupled in our `RSAXcel` block (shown in Figure 7), finally forming a fully function hardware accelerator for RSA that can be utilized by our processor.

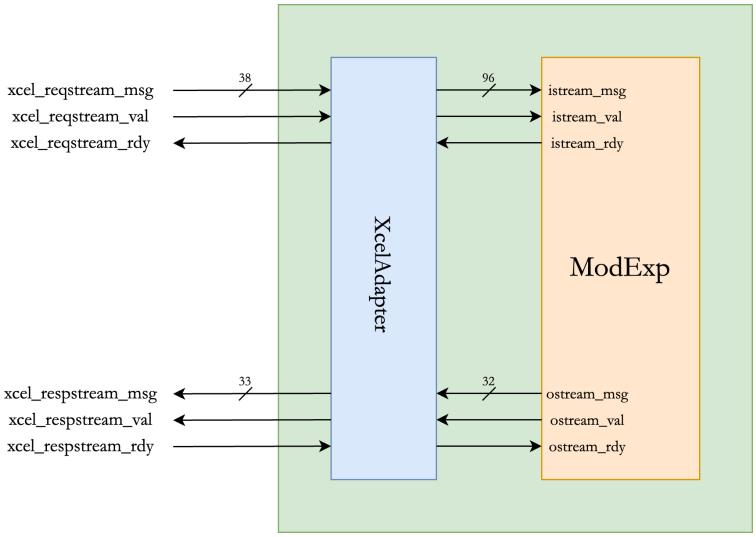


Figure 7: Block diagram of the `RSAXcel` unit

3.2 Montgomery

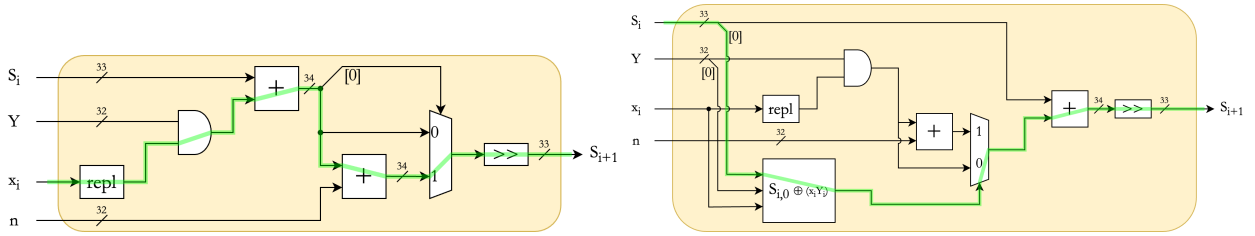
When implementing our Montgomery algorithm in hardware, Montgomery proposes a modified algorithm for computing the reduction (shown in Algorithm 2) of a product of two values X and Y [4]. Notably, this algorithm assumes that $R = 2^k$, where k is the number of bits in our operand; to achieve this, we will set $R = 2^{32}$, same as was done in software. In addition, it calls for N, our modulus, to be odd, as well as that $0 \leq Y < N$. While these seem like severe limitations, these are already satisfied by the preconditions on RSA overall (our modulus will always be odd, as it is the product of two primes, and our initial base will satisfy $0 \leq b < N$, with all intermediate results additionally satisfying this as a result of being computed under the modulus of N). Because of this, we can implement this algorithm with no loss of generality for RSA applications.

The proposed algorithm can be seen in Algorithm 4, with representation inspired by K.Ko's paper [4, 2]. Here, we can see that the reduction of the product (done with the division by 2 on line 7, as well as the conditional addition of the modulus to remain under it in line 5) is heavily intertwined with the overall multiplication (primarily accomplished by the iterative addition of Y depending on X in line 3). However, what is perhaps even more relevant is that this algorithm allows us to avoid large, expensive computations. The "multiplication" on line 3 can be achieved with an AND gate (as it is multiplying only by a single bit of X at a time), and the "division" on line 7 can be implemented as a shift to the right by 1, as it is division by 2 (this occurs k times to achieve multiplication by $R^{-1} = 2^{-k}$).

Algorithm 4 Montgomery Multiplication-Reduction in Hardware

Require: $R = 2^k$, N is odd, $0 \leq Y < N$
Ensure: $S = XYR^{-1} \pmod N$
1: $S \leftarrow 0$
2: **for** $i = 0$ to $k - 1$ **do**
3: $\quad S \leftarrow S + (X_i * Y)$
4: $\quad$ **if** S is odd **then**
5: $\quad\quad S \leftarrow S + N$
6: $\quad$ **end if**
7: $\quad S \leftarrow S/2$
8: **end for**
9: **if** $S > N$ **then**
10: $\quad S \leftarrow S - N$
11: **end if**

Finally, Montgomery notes that the result of the main loop is described by $0 \leq S < 2N$, and that S would be our desired result, but possibly with an extra factor of N. We know that our final result (under the



(a) Naive `AddRed` module (b) `AddRed` module with early LSB computation

Figure 8: Implementations of the `AddRed` module

modulus of N), should be less than N, so we can simply subtract N conditionally (on line 10) if our given output is greater than N.

For this accelerator, I chose to unroll the loop in Algorithm 4 in hardware, preferring speed (the main goal of Montgomery computations) over area by allowing for combination communication between iterations of the loop. We can implement a module to execute an iteration of the loop, and connect k of them (in our case, 32) in series to implement the entire loop. We first started by implementing the arithmetic inside the loop (contained in `AddRed` modules, standing for Add-Reduce), and considered two different implementations shown in Figure 8. Figure 8a shows our initial implementation suggested by Montgomery [4]. We can see the direct connection between it and the inner loop in Algorithm 4; X_i is replicated to AND with each bit of Y, and then added to S_i. We pass along either this sum, or the sum incremented by N, depending on the LSB of the sum. Finally, we shift the passed-along value to the right by 1 (dividing it by 2) to obtain S_{i+1}. For comparison, the critical path through this module is highlighted in green.

However, in his paper, Ç. K. Koç proposes a re-ordering of the arithmetic in order to achieve speedup, with the proposed module shown in Figure 8b. Here, we can utilize the fact that X, Y, and N are typically available much earlier than S_i; the former are available as soon as the message arrives, whereas the latter must propagate through all of the connected `AddRed` modules in series. Additionally, the LSB that controls the conditional add can be quickly computed as $S_{i,0} \quad (X_i Y_i)$ without computing the entire sum [2]. Because of this, we can compute both $X_i \cdot Y$ and $(X_i \cdot Y) + N$ in advance. As soon as S_i arrives, we can quickly compute the LSB of $S_i + (X_i \cdot Y)$, and use it to determine which of the above factors ($X_i \cdot Y$ and $(X_i \cdot Y) + N$) should be added to S_i and subsequently shifted to achieve the correct result. The common critical path through this module is highlighted in green as well (although the first instance of it may have a different critical path, due to all the operands arriving at the same time). We can note that this new critical path, when compared to our previous implementation, avoids an AND and 33-bit sum computation for some simple single-bit logic (which we assume is faster), allowing for faster combinational computation through the module. Because of this, we will use the implementation shown in Figure 8b for our implementation.

To finally finish the implementation of Algorithm 4, we instantiate 32 of the `AddRed` modules, with $S_0 = 0$ being fed into the first. While these modules could in theory communicate combinationally, that design was not able to meet timing when pushed through the flow under a clock constraint of $3ns$ (given from our processor). Since adding the full combinational communication would hurt the maximum clock

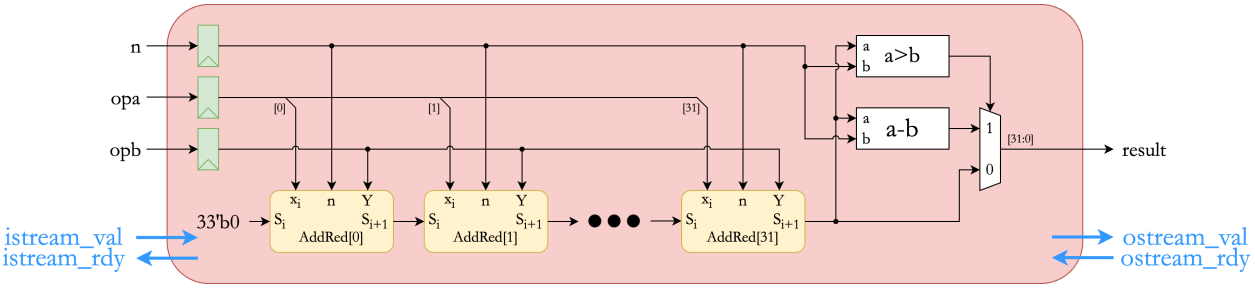


Figure 9: Block diagram of the `MontMulRem` unit

speed of our processor (thereby hurting every other application it was used in), we instead opted to pipeline the `AddRed` modules to reduce the critical path, with trial-and-error determining that we needed 4 stages with 8 `AddRed` modules per stage. Finally, we can use a simple comparator, subtractor, and multiplexer to implement the conditional subtraction at the end of the loop (note that, since this value is guaranteed to be less than N, which can be represented by 32 bits, we also only need 32 bits to represent this result, and can safely disregard the higher ones). This is implemented in the `MontMulRem` module, which can be seen in Figure 9. Note that, as the naming suggests, it serves a similar function to the `MulRem` unit from before (computing the equivalent operation in Montgomery Form), but seeks to achieve speedup by avoiding the extensive operation of iterative division (with our 4 stages only taking 4 cycles to compute the multiplication-remainder operation).

Due to this similarity, we can re-use much of the framework from our `ModExp` unit to create our `MontModExpMul` unit, replacing our `MulRem` instantiations with `MontMulRem` units (the reason that this isn't our full `MontModExp` unit is because it doesn't handle to convertion in and out of Montgomery form, only the multiplication as part of the modular exponentiation algorithm). The block diagram of the `MontModExpMul` module can be seen in Figure 10. To support our Montgomery operations, we only need to make a few changes. In our naive version, our result r started as 1. However, we now need to start with the Montgomery representation of 1 for r; we will take this as an input, making our overall input message carry four operands; `b`, `e`, `n`, and `r`, with `b` and `r` in Montgomery form. In addition, to compute the conversion out of Montgomery form, we need the modulus n, so we add that to our output message as well. However, besides these minor changes, the module overall stays the same, allowing for high code reusability (especially our control logic, to which no changes were necessary).

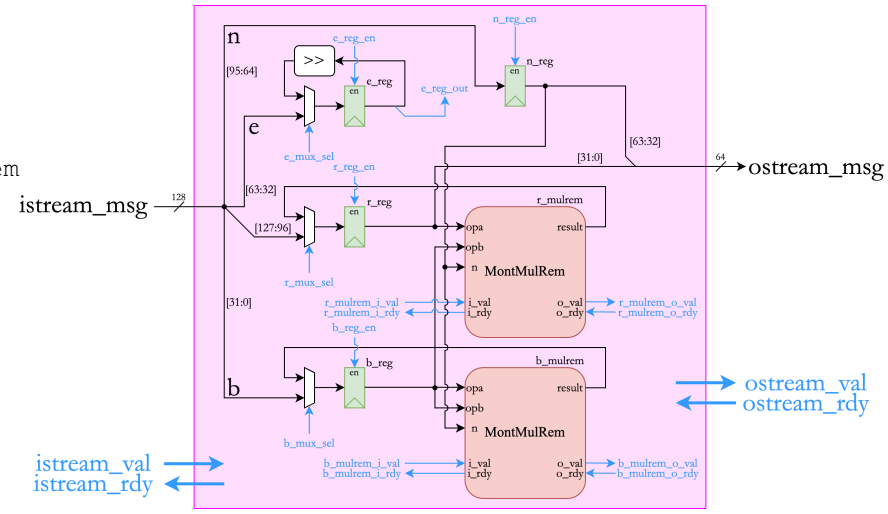


Figure 10: Block diagram of the `MontModExpMul` unit

However, in order to fully implement our `MontModExp` unit, we must additionally handle the conversion in and out of Montgomery form. Figure 11 show a block diagram of our `MontConvertIn` module, which is used to implement Equation 8 and convert our base `b` and initial result `r` = 1 into Montgomery form. We first use our remainder module to compute R^2 mod N, as we give it the inputs of 2^{64} (which is R^2, in our case), as well as the passed-in value of N. It then proceeds to use `MontMulRem` units to multiply both `b` and

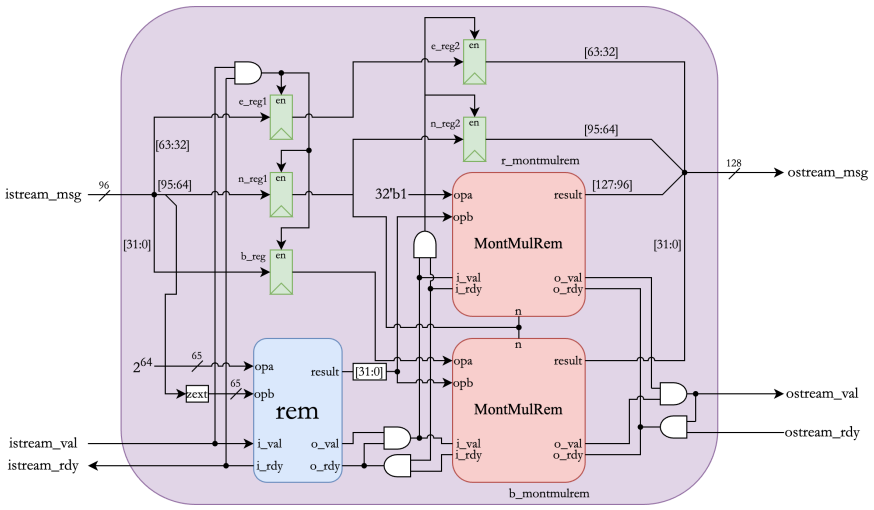


Figure 11: Block diagram of the `MontConvertIn` unit

1 (our initial value for `r`) by $R^2 \mod N$, thus converting them into Montgomery form by Equation 8. It additionally makes sure to use extra registers to properly pass along e and b as this conversion is happening, making sure all messages are kept together at the two stages. Notably, this module takes in operands as a `ModExp` unit would expect them, and outputs operands as our `MontModExpMul` would expect them.

Additionally, we need to handle the conversion of our results out of Montgomery form. However, the process for doing this described in Equation 9 (and therefore the hardware implementation as a `MontConvertOut` unit, shown in Figure 12) is relatively simple. Equation 9 simply requires us to perform our Montgomery multiplication-remainder operation with the desired value and 1 (over the modulus n) to get the value out of Montgomery form. Our `MontConvertOut` unit is therefore just a thin wrapper around a `MontMulRem` unit, with one of the operands hard-coded to be 1. However, it is worth noting that this unit takes in operands as our `MontModExpMul` outputs them, and outputs the result as our `ModExp` unit would output them.

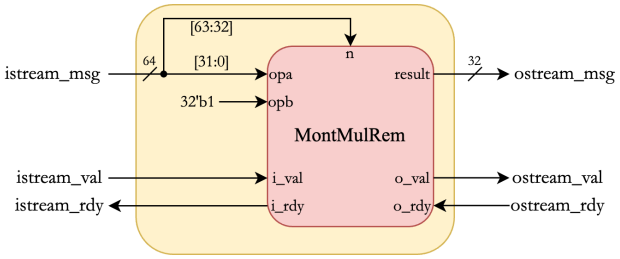


Figure 12: Block diagram of the `MontConvertOut` unit

Once we have all these units, we can finally form our `MontModExp` module, shown in Figure 13. This module simply converts the inputs into Montgomery form using a `MontConvertIn` module, performs the required loop of `MulRem` operations with a `MontModExpMul` module, and converts the final results back out of Montgomery form with a `MontConvertOut` module. However, since we designed our `MontConvertIn` module to take in inputs as our `ModExp` module would, and our `MontConvertOut` module to output values as our `ModExp` module would, our `ModExp` and `MontModExp` modules share the same interface for the same functionality. This allows us to form the overall accelerator module `RSAMontXcel` (shown in Figure 14) by simply replacing the `ModExp` module from our `RSAXcel` unit in place of a `MontModExp` module. This not only allows for very high levels of code reusability, but can also help streamline our testing strategy with test case reusability, as tests that can be applied to one should be applicable to the other due to the common interfaces between the modular exponentiation modules and the overall accelerators.

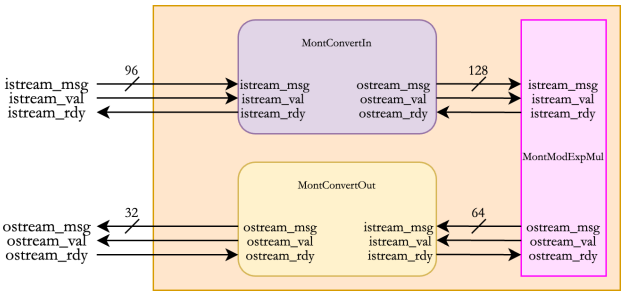


Figure 13: Block diagram of the `MontModExp` unit

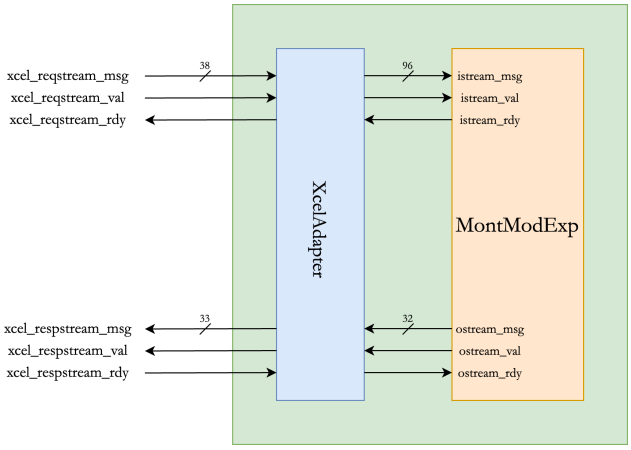


Figure 14: Block diagram of the `RSAMontXcel` unit

4 Testing Strategy

As with all designs, we can only assume that the tested portions of our RSA system are functional, and must therefore make sure to thoroughly test our entire design. We first tested our overall algorithms against our functional level model. From there, we could use them to test our baseline software and alternative hardware at various levels, making sure to unit test our blocks individually before instantiating them. This gradual progression of tests limited the scope we had to look at when encountering bugs (as previous blocks had been verified), making the overall testing process much quicker and smoother. Additionally, while directed testing was used to verify initial functionality, as

well as test corner cases, our functional level models allowed us to know what the correct answer for tests was at runtime, allowing us to perform random testing and achieve a wide breadth of test coverage.

4.1 Functional-Level

Before testing any of our implementations, we took some time to develop a functional-level model of our accelerator in PyMTL. For this, we were able to leverage the RSA Python library, as our model simply called the `encrypt_int` function to perform modular exponentiation on the operands (as encryption and decryption are both implemented by modular exponentiation). We made sure to test this with high-level RSA requests, with the keys additionally provided by the library, specifically with the `newkeys` function. While directed testing was used with simple messages to ensure basic functionality of the system and accelerator protocol, much of our testing was random and implemented with these library functions. This not only allowed for concision in and ease of testing using our high-level functional-level model, but verified that our test cases were functional before using them on our RTL implementations.

4.2 Algorithm

Before implementation, it is important to first test our overall algorithm, to make sure that our understanding is correct. For this, we can use our Python implementations of our algorithm in comparison to our library implementation. For a given set of keys, we generate a random message M such that $0 \leq M < n$. We then encrypt this message using our public key with all three methods (naive, Montgomery, and our library implementation), and verify that all three methods produced the same result. We can then decrypt that message using our private key, and again verify that all three methods produced the same result, as well as that the result is the same as our initial message. For our testing, this process was repeated for 1000 keys generated from `rsa.newkeys` in the Python RSA Library. While this ad-hoc testing isn't comprehensive, it gives us confidence that our understanding of the overall algorithm is correct.

4.3 Div-Rem Unit

Before testing our entire processor, we should also make sure to test our *div_rem* unit in isolation, allowing us to gradually increase the scope of our testing and catch any bugs early on. To do this, we tested the *div_rem* unit first with simple inputs for both division and remainder, to test basic functionality. From here, we additionally tested large inputs to ensure that there was no data loss in higher bits, as well as interleaving division and remainder operations to ensure that our module could change what it computed at arbitrary times. Once this directed testing was completed, we also provided the unit with random inputs to achieve a breadth of testing, giving confidence that our module would function correctly under a variety of stimuli. In testing our division unit separately, we can limit the scope when testing our larger system, as we have confidence that any issues found will not be inherent to the operation of the *div_rem* unit.

4.4 Processor

Before moving forward, we must also test our modifications to our processor to verify that they are working correctly. This way, when we run our code on our processor, we can limit the scope of any issues to the algorithm, and not the processor itself. Previously, each instruction had their own unit tests to verify their functionality. To test our new `divu`, `remu`, `mulhu`, and `lbu` instructions, we can replicate this - supplying the processor with data using the `csrr` instruction, operating on it with the instruction under test, and verifying that we have the correct result with the `csrw` instruction. For each of these, we perform an initial ad-hoc test of written-out assembly, but quickly move to using the templates used for other instructions, allowing us to have high code reusability and more ledgible tests. Our directed tests include testing different corner-case values if applicable (such as multiplication by 0 and 1 for `mulhu`), as well as gradually decreasing the number of `nops` inserted before and after the instruction under test in order to verify that our stall signals are set appropriately when presented with data dependencies. Finally, in addition to directed testing, we include random tests with random data to ensure a large breadth of test coverage and attempt to catch any cases we missed.

These tests are first run on our functional-level processor model to ensure their validity, before being run on our RTL processor implementation. Once these successfully pass, we can have high confidence that our processor will behave correctly when presented with these new instructions.

4.5 Baseline

For our baseline implementations in C, we take a more rigorous approach to testing. We first want to verify the validity of our two different implementations of the modular exponentiation helper function using directed testing. For our naive `mod_exp` function, we first stimulated it with simple, small values, to ensure basic functionality. From there, we stimulated it with multiple values sequentially, as well as large values, to ensure that the function still operated correctly when used repeatedly, as well as that it didn't encounter any errors from loss of data when our values occupied more significant bits.

With our `mont_mod_exp` function using Montgomery multiplication, our testing remained largely the same, as the functions had the same semantics and interface, allowing for high amounts of code reusability. The only this that we needed to modify was some of our moduli, as Montgomery multiplication requires them to be coprime to R. For our implementation, this means that they must be odd (this results in no loss of generality, as n is the product of two large prime numbers, and will therefore always be odd in RSA applications).

Unfortunately, we could not perform random testing for this helper function, as a basic modular exponentiation approach (simply writing the equivalent of "$x = (a^b) \mod c$" in C) is too slow to be viable under random data (the reason why modular exponentiation is used). However, we can perform random testing one layer higher, on our `encrypt` and `decrypt` functions for both our naive and Montgomery implementations. Here, we can use pre-generated RSA keys, as well as a randomly generated message M (adjusted if needed so that $0 \leq M < n$). After encryption and subsequent decryption, we can verify that our resulting message is the same as our initial one. This random testing gives us broad coverage of our tests, as well as confidence that our algorithm will function under a variety of messages.

These tests were first compiled and run natively to ensure their correctness. From here, we cross-compiled them in RISCV and ran them on our FL model of our processor, to ensure that our processor would theoretically be able to run the tests and algorithms (i.e. the processor supported all of the instructions and functionality needed). Finally, we ran these tests on our RTL processor implementation to ensure that our standalone system implementation was capable of performing RSA encryption and decryption. With this, we can be confident that our baseline implementations successfully implement 32-bit RSA encryption and decryption overall.

4.6 Alternative

Once our baseline designs were tested, we then moved on to testing our accelerator designs. We tested these sequentially, with the hope that test cases for one could be re-used for the other.

4.6.1 Naive

For our naive approach, we first tested out our `MulRem` module, with our functional-level reference being the simple computation of a product followed by a modulus in Python. From there, we additionally tested our `ModExp` module, with our functional-level model being the `mod_exp` function as defined in Listing 1. For both of these, we made sure to initially test with simple , small operands, as well as large operands to make sure that our data was correctly maintained (as the number of bits in each representation changed throughout the system, making it easy to assign to few bits and lose track of needed data). In addition to our directed testing, our functional-level models allowed us to perform random testing by providing access to the correct results for any given inputs.

Once our building blocks were tested, we could test the overall `RSAXcel` module. For this, we were able to re-use our test cases from our functional-level model. This not only allowed for ease of implementation (as we could import the test case table and simulation functions, simply running them with the new model), but helped to prevent errors when re-writing test cases, allowing for a smoother and more transparent testing framework.

4.6.2 Montgomery

For our Montgomery accelerator implementation, with the exception of our `AddRed` block in isolation due to the simplicity, all of the building blocks mentioned were unit tested (including `MontMulRem`, `MontModExpMul`, `MontConvertIn`, `MontConvertOut`, and `MontModExp`). These were tested very similarly to our naive approach, with simple tests to ensure basic functionality, as well as tests with large operands to ensure correct retention of data. For our functional-level model, we used our previously-tested `MontMultiplier` class, as seen in Listing 2. Not only does it provide the overall functionality for the system, but many of the member functions in the class mapped directly to modules in our hardware. For instance, our `MontConvertIn` module implements the same functionality as the `convert_in` member function of the `MontMultiplier` class. Because of this, we were able to use the member functions `convert_in`, `convert_out`, and `multiply` as functional-level models our `MontConvertIn`, `MontConvertOut`, and `MontMulRem` modules, respectively, allowing us to perform random testing for them. Similarly, our `MontModExpMul` implements the `mont_mod_exp` function from Listing 2, only without the initial and final conversions. However, we could still use it to test the functionality of our `MontModExpMul` module with random testing by converting in the operands before they went to the module (using the `convert_in` member function), as well as converting them when they came out (using the `convert_out` member function), before checking them against our result from `mont_mod_exp`. This re-use of our algorithms as functional-level models not only exhibits high levels of code reusability, but showcases the importance of gradually testing along the way; since we had previously verified our `MontMultiplier` class, we could then use it to subsequently test other designs.

When testing our `MontModExp` module, we were unfortunately unable to directly re-use the tests for the `ModExp` module, as the former has more pre-requisites on the data (specifically that the modulus in odd, as well as that the base is less than the modulus, as mentioned in the implementation). However, we were still able to re-use the overall framework of the testing script, only needing to add the additional check that our randomly generated operands followed this before proceeding.

Finally, we tested our overall `RSAMontXcel` module. Here, since it has the exact same functionality as the `RSAXcel` module, and we were testing it under real-life RSA keys and data (where our prerequisites from before are already satisfied), we could directly re-use our test cases from the `RSAXcel` and functional-level accelerator model testing, allowing for high levels of code reusability and minimizing the possibility of errors in our testing code.

5 Evaluation

Once we have our designs, we can evaluate them across different key metrics to obtain concrete evidence about the design tradeoffs that occur across them. For our designs, I will be evaluating them on performance (obtained through RTL simulations), as well as their area and energy usage (obtained through the results of our automated ASIC flow, generated using the `mflowgen` setup from Lab 2).

5.1 Performance

Using our RTL simulations for our processor running our baseline designs, as well as our processor-accelerator system for both the naive and Montgomery accelerator, we can obtain concrete results about the performance of our different RSA systems. For this, I used a pair of keys generated aat random from our FL Python library model, as well as a random message and its corresponding ciphertext. These were used by all models, which were evaluated on their performance on encrypting the message and decrypting the ciphertext. These results are summarized in Table 1. For this evaluation, I also wrote a "dummy" algorithm, which simply returned the answer our test expected whenever the `encrypt` or `decrypt` functions were called; subtracting the performance of this test (9 cycles) achieves our "normalized" performance and eliminates the overhead of getting in and out of the functions, allowing us to focus more on the overall algorithms.

Beginning with our baseline (our processor running our naive algorithm), we can see that our RSA algorithm took a substantial number of cycles; 3523 for encryption, and 8148 for decryption. The large discrepancy between these, even though it is fundamentally the same operation, comes down to the operands. When encrypting, our exponent is e, which is commonly $65537 =$ `0x10001`; this is relatively sparse, thereby

	Proc (Naive)	Proc (Montgomery)	ProcXcel (Naive)	ProcXcel (Montgomery)
Encryption (cycles)	3532	3590	1071	178
Encryption - *Normalized* (cycles)	3523	3581	1062	169
Encryption - *Speedup from* `ProcNaive`	N/A	**0.98**	**3.32**	**20.85**
Encryption - *Speedup from* `ProcMontgomery`	**1.02**	N/A	**3.37**	**21.19**
Decryption (cycles)	8157	5144	1849	243
Decryption - *Normalized* (cycles)	8148	5135	1840	234
Decryption - *Speedup from* `ProcNaive`	N/A	**1.59**	**4.43**	**34.82**
Decryption - *Speedup from* `ProcMontgomery`	**0.63**	N/A	**2.79**	**21.94**

Table 1: Performance of our different RSA systems

only requiring 17 multiplication-remainder (`mulrem`) operations as part of Algorithm 1 for us to obtain the correct result [6]. However, decryption is done using d as an exponent (obtained from Equation 1); this factor depends on n, and is usually much larger and less sparse (for our tests, $d = 977851481 =$ `0x3a48d459`). This leads to more required `mulrem` operations as part of Algorithm 1 to obtain the correct result; bits in more significant places mean that we must execute more `mulrem` operations on `b`, and more density means more times we must execute a `mulrem` operation on `r`. Additionally, our baseline suffered because of the density in particular during decryption; other systems are able to avoid too much extra cost when needing to operate on `r` due to either a fast `mulrem` operation (as part of the Montgomery algorithm) or the ability to do this in parallel with the required operation on `b` (in custom accelerator hardware). Given that our naive software approach had neither of these, it suffered in particular, leading to all other designs having speedup relative to this baseline on decryption rather than encryption.

Our Montgomery algorithm for our processor attempted to achieve speedup by amortizing the cost of converting values in and out of Montgomery form over the many faster `mulrem` operations they can achieve. While it is harder to say whether this will be faster from the abstract algorithm alone, our RTL simulations can give us concrete results about this tradeoff in hardware. For encryption, we can see that the fewer `mulrem` operations meant that the cost of conversion outweighed the gains, leading to a very slight overall slowdown. However, with the many more `mulrem` operations as part of decryption, we can see that our Montgomery algorithm was able to sucessfully overcome the conversion cost with the faster `mulrem` operations, leading to an overall 1.59x speedup. This shows how by understanding our algorithm and implementing it differently, we can achieve performance gains even with no extra hardware.

Switching over to our custom accelerator implementations, we can see that a hardware implementation of the naive algorithm was able to achieve moderate speedup in both encryption and decryption compared to both our naive and optimized software. The first factor that allowed for this was the custom hardware. Our overall processor is a 32-bit machine, but executing `mulrem` operations on 32-bit operands requires 64-bit precision, in order to avoid the data in the higher bits of the product that will become relevant under the modulus. Our processor solves this by emulating 64-bit operations through multiple 32-bit operations and shifting, making the overall arithmetic operations slower than they might normally be. However, with our custom hardware, we can provide the support needed for 64-bit operations (such as 64-bit multiplier and remainder units), eliminating the need for time-consuming emulation. The other factor that leads to speedup is the parallelization we can achieve in hardware. Our processor must operate on `r` (if needed) and `b` from Equation 1 in series, as it can only execute one instruction at a time. However, our hardware can simply use 2 `MulRem` modules (only using the one for `r` conditionally, as Algorithm 1 suggests), allowing for the simultaneous execution of any required `mulrem` operations to achieve speedup.

Finally, our Montgomery accelerator coupled to the processor outperformed all other systems, with a 20.85x speedup in encryption and 34.82x speedup in decryption compared to our naive processor system, and even a 21.19x speedup in encryption and 21.94x speedup in decryption compared to our Montgomery processor system. This is partially because it inherits the same benefits from custom hardware as the naive `ProcXcel` system does, but also because the `mulrem` operations it executes are now extremely fast due to clever algorithms that overlap the multiplication and reduction stages of the algorithm [4, 2]. Our naive accelerator must rely on iterative 64-bit division to compute a `mulrem` operation (with the general-purpose processors only being slower from emulating the same operation), but operating in the *N-residue* form allows our pipelined stages in the `MontMulRem` module to perform the equivalent operation in just 4 cycles. This pipelining is also due in part to the shorter critical path inside the `AddRed` modules due to early computation

of the LSB [2].

While our 32-bit RSA implementation is a good proof-of-concept, it will not stand up to modern recommendations for RSA security [3]. When extrapolating this to more bits, we know that conversion in and out of Montgomery form (a constant factor) will quickly pale in comparison with the actual computation of the modular exponentiation (which is $O(N^2)$ in the worst case; we have to loop over our exponent's N bits, and perform either an N-bit mulrem operation or have N `AddRed` modules). Therefore, we can expect the benefit from a faster `mulrem` operation (a better constant factor) in our Montgomery implementations (at the cost of an increasingly irrelevant conversion in and out) to scale better, leading to greater speedup for larger numbers of bits.

5.2 Area

With just our RTL designs, it is hard to get a quantitative sense of the comparisons between designs in terms of the area and energy that they use. However, with our ASIC flows, we can now harden our design to obtain these concrete metrics, allowing us to have full knowledge of the tradeoffs between our designs. One important metric we can obtain is the area for our different designs, including our standalone processor (with trivial overhead due to a "null" accelerator begin attached to include the accelerator connections), our processor with our naive accelerator (`ProcXcelNaive`), and our processor with our Montgomery accelerator (`ProcXcelMont`). Additionally, we can push our accelerators through individually, to get information on their area in isolation as well. These results are summarized in Tables 2a and 2b, with the amoeba plots that visualize the area in Figures 15 and 16 (with the `MulRem` and `MontMulRem` units highlighted for our accelerators).

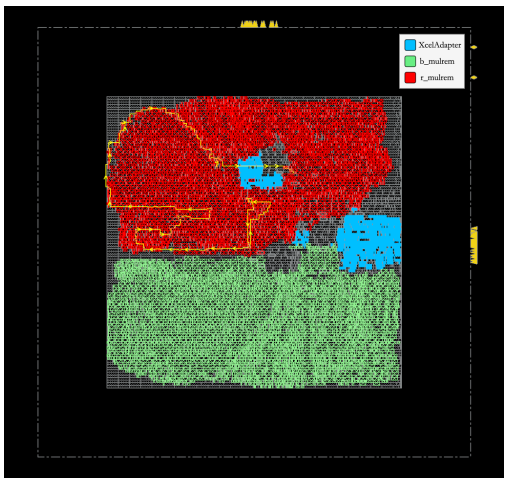
	RSAXcel	RSAMontXcel
Area (μm^2)	17512.376	75283.586
Average Area of a `MulRem`/`MontMulRem` module (μm^2)	7776.377	13846.736

(a) Areas of our RSA accelerators

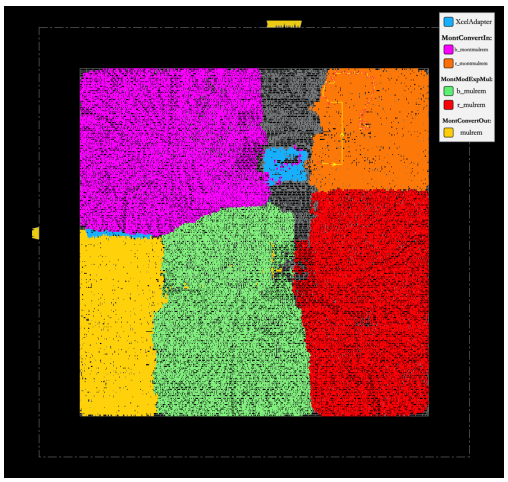
	ProcXcelNull	ProcXcelNaive	ProcXcelMont
Area (μm^2)	21434.812	38658.046	96137.986

(b) Areas of our RSA systems

Table 2: Areas of our different RSA modules



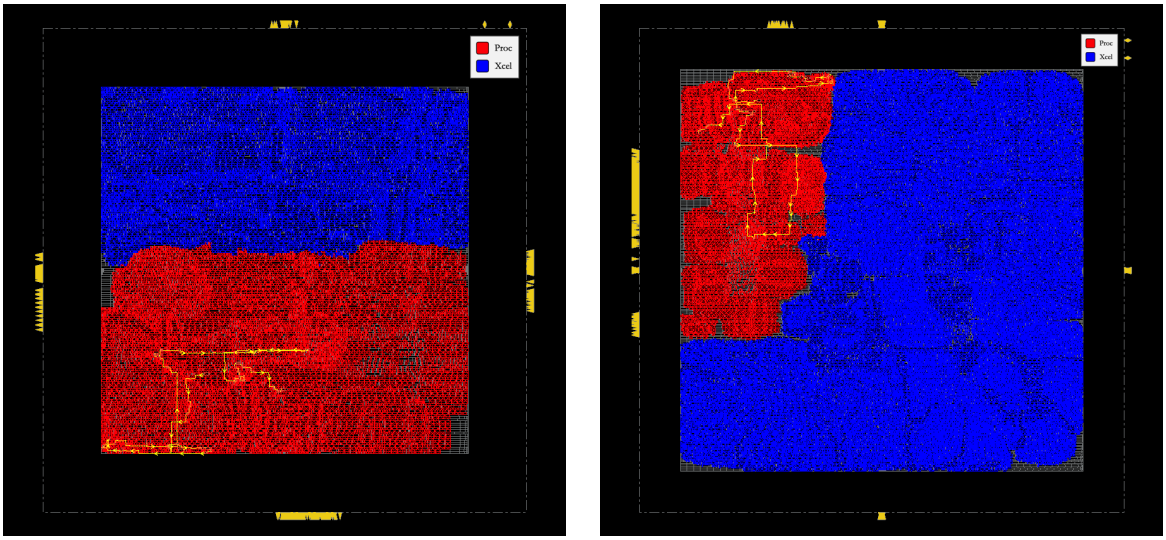
(a) Amoeba plot of our `RSAXcel` accelerator module



(b) Amoeba plot of our `RSAMontXcel` accelerator module

Figure 15: Amoeba plots of our accelerators

Comparing our accelerators in Figure 15, we can see that our `RSAMontXcel` accelerator is substantially larger than our `RSAXcel` accelerator, being 4.30x larger. This is due to the implementation of the



(a) Amoeba plot of our `ProcXcelNaive` system (b) Amoeba plot of our `ProcXcelMont` system

Figure 16: Amoeba plots of our processor-accelerator systems

`MontMulRem` module, compared to the size of our naive accelerator's `MulRem` module. The latter implements the `mulrem` operation using a optimized multiplier (our synthesis tool chose an unsigned Booth-Encoded Radix8 multiplier from the Synopsys DesignWare component library) and iterative divider, allowing for relatively little area to be used ($7776.377\mu m^2$, on average). However, unrolling the operation combinationally for the 32 `AddRed` units in our `MontMulRem` module takes up significantly more area ($13846.736\mu m^2$, on average). This is augmented by the fact that our `RSAXcel` module only needs 2 `MulRem` units in the main `ModExp` module, whereas our `RSAMontXcel` module needs 3 more for a total of 5; 2 extra for conversion into *N-residue* form in the `MontConvertIn` module, and one more for conversion out in the `MontConvertOut` module. This leads to an area overhead of $2(13846.736\mu m^2 - 7776.377\mu m^2) = 12140.718\mu m^2$ for the enlarged size of the initial 2 `MulRem` modules, plus $3(13846.736\mu m^2) = 41540.208\mu m^2$ for the additional 3 `MontMulRem` modules needed for conversion, for a total of $53680.926\mu m^2$ of extra area from these modules. This accounts for the large majority of the extra $57726.21\mu m^2$ of area that our `RSAMontXcel` module uses compared to our `RSAXcel` module.

This area analysis translates directly over to our RSA systems, with our accelerators coupled to processors. our `ProcXcelNull` system served as the system to run our baseline implementations, having a "null" accelerator just to maintain the accelerator connections. Relative to this, our `ProcXcelNaive` system (processor with a `RSAXcel` accelerator) had an area overhead of $17223.234\mu m^2$, only slightly less than what our `RSAXcel` occupied in isolation (with the slight decrease due to the fact that our null accelerator still took up some area in the `ProcXcelNull` implementation). Similarly, our `ProcXcelMont` system (processor with a `RSAMontXcel` accelerator) had an area overhead relative to the `ProcXcelNull` system of $74703.174\mu m^2$, slightly less than the area of our `RSAMontXcel` in isolation. Combined with our previous data, this allows us to analyze the tradeoff between area and energy in our RSA systems, visualized in the plots in Figure 17.

Here, we can see the tradeoff we have to make as designers between are and energy. The least area designs are our baseline designs with just the processor and null accelerator (with the most performant design depending slightly on the application, but assuming that $0.98 \sim 1$, we can generally say that the Montgomery version is faster due to the speedup on decryption). However, as we want to obtain faster performance with custom accelerator hardware, we must also sacrifice area. This comes in different degrees; our `ProcXcelNaive` system has 1.8x for fairly sizeable performance gains relative to both baseline designs. We can obtain even faster performance using our `ProcXcelMontgomery` module, but it comes at the cost of a system with 4.5x the area of our baseline. With the exception of our baselines (which change performance with the algorithm), no system completely dominates another; any could be a valid choice, depending on

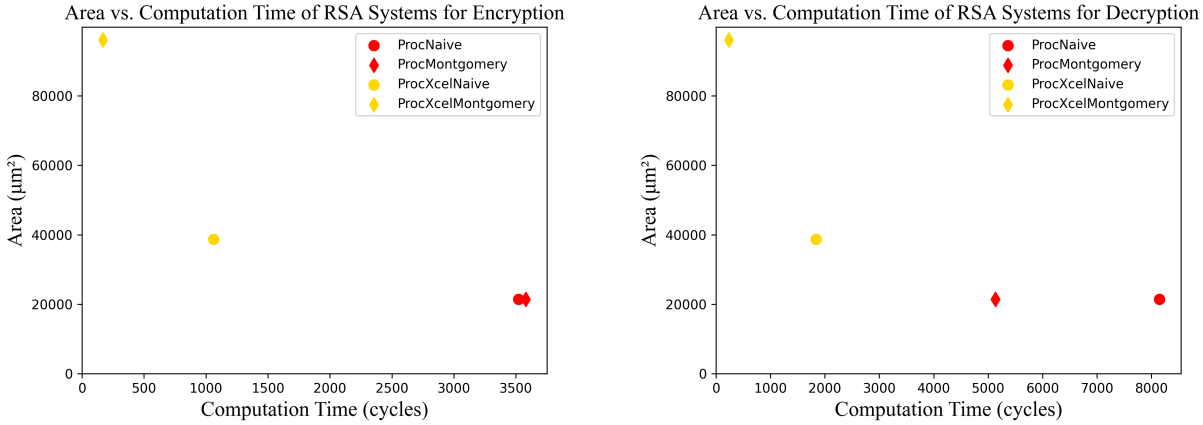


Figure 17: Area vs. Performance plots for our RSA systems

the constraints that are imposed on our system (they all lie along a pareto-optimal curve).

Similarly to with performance, this would change for scaling up the number of bits. Our processors wouldn't largely change due to the emulation of N-bit operations. Our `RSAXcel`'s area would grow with N, as it needs an N-bit multiplier and `div_rem` unit. However, our `RSAMontXcel` would grow in area proportional to N^2; it needs N `AddRed` modules in the `MontMulRem` unit, each of which need an $(N+2)$-bit adder. Therefore, as we scale up our number of bits, our accelerator implementations will grow larger; `RSAXcel` will grow with N, and `RSAMontXcel` will grow with N^2.

5.3 Critical Path

In addition to area, a useful metric our ASIC flows can tell us it the **critical path**, the longest register-to-register path in our design, and the one that limits how fast our design can perform. While our designs were constrained with the shortest clock period at 3ns (and not optimized beyond that), the critical path is still useful in knowing which part of our design is limiting our speed, and which one we'd have to optimize first in order to have a faster system. For our accelerator designs, these paths can be additionally viewed as the paths highlighted in yellow in Figure 15, as well as the paths for the overall systems in Figure 16.

The majority of our designs (`RSAXcel`, as well as all of our processor-accelerator systems) incorporate a combinational 64-bit multiplier followed by a `div_rem` unit. For our accelerator, this is part of the `MulRem` module, whereas for our processor systems, this path can be found as a bypass path from the result of the multiplier in the `X` stage back to the input of the `div_rem` unit in the `D` stage. For these designs, this is the critical path (specifically inside the `r_mulrem` instantiation for our accelerator); the combinational 64-bit multiplier is very large, and takes a while for the operands to pass through (with the end of the path being in the `div_rem` unit due to the logic done to store the inputs, as they must be muxed with other intermediate values that are used during operation). Note that when our accelerator is coupled with the processor, the multiplier→`div_rem` path in the processor is longer than the one in the accelerator, as it must also go through bypass muxing.

The only module for which this isn't the critical path is the `RSAMontXcel` module. This is by design; we modified the Montgomery algorithm to strategically overlap the multiply and reduction phases, breaking them up into smaller portions that only use adders in the `MontMulRem` module [2]. However, this is still the bulk of our computation; we had to pipeline the stages in the `MontMulRem` module to meet timing, so it makes sense that our critical path for the overall accelerator is in one of these stages (specifically, the first stage of the `r_montmulrem` instantiation in the `MontConvertIn` module). While this is hard to compare to our naive implementations in terms of speed (as our Montgomery version optimized the computation at an algorithmic level), it tells us that we can achieve even more speedup at a hardware level if we were able to execute this combinational computation faster.

5.4 Energy

Lastly, an important part of our designs is the power and overall energy consumed during operation. Similar to with performance, I also calculated the energy used for our "dummy" algorithm (which varied across different hardware), and subtracted it from our reported values to obtain our "normalized" energy (just the energy used in the actual computation). These results are summarized in Table 3.

	Proc (Naive)	Proc (Montgomery)	ProcXcel (Naive)	ProcXcel (Montgomery)
Encryption - Power (mW)	3.46	5.33	4.55	6.93
Encryption - Energy (nJ)	40.28	62.87	19.31	10.85
Encryption - Energy *(Dummy)* (nJ)	4.78	4.78	7.78	7.99
Encryption - Energy *(Normalized)* (nJ)	**35.50**	**58.09**	**11.53**	**2.86**
Decryption - Power (mW)	3.38	5.71	4.20	7.08
Decryption - Energy (nJ)	86.17	94.03	27.63	12.47
Decryption - Energy *(Dummy)* (nJ)	4.81	4.81	7.80	8.01
Decryption - Energy *(Normalized)* (nJ)	**81.37**	**89.22**	**19.82**	**4.46**

Table 3: Power and Energy Usage of our different RSA systems

Beginning with our baseline designs, we can see that our Montgomery algorithm uses significantly more power than our baseline. This is due largely to the underlying microarchitecture; our naive algorithm uses `div` and `rem` operations, both of which require many cycles to pass through our iterative `div_rem` unit. During this time, the rest of the hardware isn't doing anything (as our processor is stalled waiting for the operation to complete), resulting in only a small fraction of the hardware being used during the computation. However, in modifying our algorithm to avoid `div` and `rem` operations in favor of `mul` operations, our Montgomery algorithm utilizes the entire processor more of the time, leading to it consuming energy more often and leading to an overall power increase. Additionally, when comparing encryption to decryption, our naive algorithm uses less power on average for decryption (as more `mulrem` operations are performed, leading to more time iterating in the `div_rem` unit instead of engaging the entire processor). However, our Montgomery algorithm consumes more power in decryption, as the increased number of `mulrem` operations mean more time engaging the entire processor.

We can predict that this power consumption will result in more energy overall being consumed by our Montgomery algorithm for encryption, as it is slower and uses more power. Interestingly, however, the slight speedup in performance in decryption for the Montgomery algorithm didn't outweigh the larger power consumed, leading to the Montgomery algorithm also using more energy for decryption. This has interesting implications for which one we might choose for a given task, and how we can change our choice at runtime; on the same hardware, if we want a (generally) faster algorithm, we'd use our Montgomery algorithm, but if we wanted a more energy-efficient computation, we'd use our naive algorithm.

Looking at our accelerated `ProcXcelNaive` system, we can see that it used more power than our naive processor algorithm, but less power than our Montgomery processor system. Similar to the naive algorithm, it only uses a portion of the hardware at any given time, as it similarly iterated in the `div_rem` unit for division and remainder computations (leading to less power usage than the Montgomery algorithm in our processor). However, it still engaged more hardware on average than the naive algorithm in our processor, as `mulrem` computations (the main arithmetic operation) were performed in parallel for r and b in Algorithm1, leading to more overall power usage. It additionally had more power usage in encryption than decryption, for the same reasons as our naive algorithm. However, despite these increases in power, the significant gain in performance dominated, leading to a substantial overall decrease in energy used.

Transition to our `ProcXcelMont` system, we can see that it consumes the most power of all the designs, due to its continuous usage of the extensive hardware in the combinational stages in the `MontMulRem` units. This had a similar impact to our Montgomery algorithm in our processor, where more power was used in decryption compared to encryption due to the increased usage of our `MontMulRem` modules. However, our `ProcXcelMont` module additionally used more power because these `MontMulRem` modules were engaged in parallel, similar to our naive accelerator system. In fact, if you multiply the power overhead from improving the algorithm to our Montgomery algorithm (1.31x for encryption, 1.69x for decryption) by the power overhead for introducing custom hardware in our naive accelerator implementation (1.54x for encryption,

1.24 for decryption), you get an overall overhead of 2.02x for encryption and 2.10x for decryption for both an improved algorithm and custom hardware. This is astoundingly close to the actual overhead of our `ProcXcelMont` system when compared to our naive baseline (2x for encryption, 2.10x for decryption), indicating that the overhead likely comes from the combined efforts of the overhead from our individual systems. However, despite all of this increase in energy, the large increase in performance still dominates, resulting in an overall dramatic decrease in energy, making it the most energy efficient design (using 12.4x less energy for encryption and 18.2x less energy for decryption when compared to our naive baseline).

Finally, along with our performance data, this allows us to understand the tradeoff between energy and performance, visualized in the plots in Figure 18. Here, we can see that for decryption, there is the tradeoff mentioned previously between our baseline systems; our Montgomery algorithm is faster, but uses more power and ultimately energy. However, for the rest of the systems, we can see that even with the increasing power, our performance still dominates, leading to lesser energy usage being directly correlated with faster performance. This leads our `ProcXcelMont` system to dominate in this field; if all you care about is energy and performance (with no consideration of area), than the `ProcXcelMont` system will always be the correct choice.

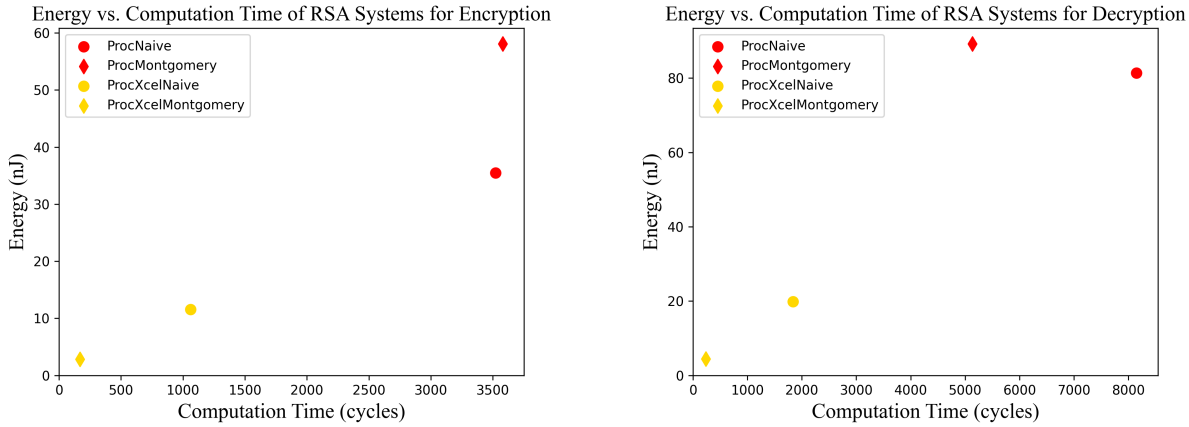


Figure 18: Energy vs. Performance plots for our RSA systems

Because of this, we can make some concrete conclusions from our different RSA implementations. Without implementing any extra hardware, our baseline designs demonstrate how we can still experience tradeoffs based solely on our choice of algorithm; our Montgomery baseline version performed better on average (up to 1.59x with decryption), at the cost of using more energy. When introducing custom hardware for this purpose, we can see how we can get large increases in performance, dominating the power consumption to lead to additional decreases in energy, at the cost of the increase in are required for the hardware. This highlights the tradeoffs between our two designs. If all we care about is performance and energy, with no regard for area, then we should always choose our `ProcXcelMont` system; it was able to achieve a average of 27.84x the performance versus our naive baseline (and 21.57x compared to our Montgomery baseline) for encryption, as well as using a fraction of the energy (averaging 6.8% of the energy compared to our naive baseline), at the cost of 4.49x the area. However, if we want a more moderate approach, taking into consideration the area of our system, then we might choose out `ProcXcelNaive` system; for only 1.8x the area of our baseline, we still get an average of 3.88x speedup versus our naive baseline (and 3.08x compared to our Montgomery baseline), as well as only 28.4% of the energy compared to our naive baseline. This shows how there is a wide spectrum of available options, and choosing the best one will rely heavily on the overall constraints and importance of parameters for our system in our specific application.

References

[1] Dan Boneh. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the AMS*, 46:203–213, 1999.

[2] Ç. K. Koç. RSA Hardware Implementation. *RSA Laboratories*, page 9, August 1995.

[3] Q. H. Dang E. B. Barker. Recommendation for Key Management Part 3: Application-Specific Key Management Guidance. *National Institute of Standards and Technology*, January 2015.

[4] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation (AMS)*, 44:519–521, 1985.

This is the defining paper for 'Montgomery multiplication', so-named for the author of the paper. In it, Montgomery describes a fast algorithm for computing the multiplication of two integers under a modulus while never having to divide by said modulus. Given that divisions are usually expensive in hardware (from Weste and Harris[1], page 708: 'Division and modulus in hardware is so costly that it may not be synthesizable.'), this algorithm can provide a quick and low-area way to implement the multiplication-under-modulus that is required for many algorithms in RSA, such as the modular exponentiation mentioned in [6]. It also provides comments on how one might go about its implementation in hardware, which will certainly prove useful when doing so for our accelerator. Lastly, it mentions that Montgomery multiplication may be inefficient when the values change frequently, as computing the residues introduce overhead. However, with our modular exponentiation algorithm [5], we iteratively multiply by the same value, lending our application to be well-suited for Montgomery multiplication

[5] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

This is the defining paper for the RSA algorithm. In it, the founders discuss the algorithm in depth, including how public and private keys are generated, as well as how they are used for encrypting and decrypting messages. Additionally, they discuss the security and mathematics behind the algorithm, and provide a proof for the security of the algorithm (noting that despite factoring being a valid way to crack the encryption, the time scale required to do so for their recommended length of keys makes it infeasible). Later discussion also provides remarks on how they imagined others might go about attempting to crack the encryption, and how/why it would fail. Lastly, they discuss algorithms for implementing the arithmetic required for RSA operations, including modular exponentiation for encryption and decryption, as well as generation of large primes and the critical parameters required for key generation. In doing so, they provide users with the tools to create implementations specific to their system, allowing us to adapt their overall procedure to a hardware-based implementation.

[6] Bruce Schneier. *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., 1996.

This book contains a general overview of cryptographic techniques and algorithms. Many of these algorithms are directly applicable to RSA, such as Euler's Totient, the extended Euclidean algorithm (specifically when used to compute the modular multiplicative inverse), and modular exponentiation (exponentiation under a modulo). While it discusses these algorithms at a high-level, it also provides a framework for each, allowing for adaptation to both software and hardware. Beyond this, it also discusses the real-world implications of cryptography, which is useful for understanding the applications and impacts of working within the field. This includes past chips (and their performance) that have been produced for RSA encryption/decryption, some of the struggles and attacks that RSA faces in the field, and some tricks and tips for having a fast implementation that is still secure.

[7] Editors Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017.

[8] Jianqin Zhou, Jun Hu, and Ping Chen. Extended Euclid algorithm and its application in RSA. In *The 2nd International Conference on Information Science and Engineering*, pages 2079–2081, 2010.

[1] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed, Addison Wesley, 2011.

This paper gives a detailed analysis of the extended Euclidean algorithm, which is used to find not only the greatest common denominator (GCD) of two values (as does the 'normal' Euclidean algorithm), but provides the coefficients that, when applied to the values in a linear combination, result in the GCD. It usefully proves that the magnitude of these coefficients will never be more than the magnitude of the original operands at any point in the process, ensuring that for our hardware implementation, we will never encounter overflow. Lastly, it discusses the implications of the algorithm for RSA. While it notes that the algorithm can be used for RSA key generation (specifically for finding d), it can also be used in encryption and decryption, specifically when finding the coefficients R^{-1} and N' involved in Montgomery multiplication when calculating residues. [4]