SENG 440 - Embedded Systems

Summer 2021

University of Victoria

Final Report - Team 12

128-Bit RSA Cryptography Implementation

August 6th, 2021

David Bishop	V00884250
Benjamin Austin	V00892013

Table of Contents

1. Introduction 1
    1.1 Contributions 1
    1.2 Platform Information and Building Instructions 2
    1.3 Project Requirements 2
        1.3.1 Hardware Requirements 2
        1.3.2 Specification Requirements 2
2. Background 3
    2.1 RSA Cryptography 3
    2.2 Solution Approaches 5
        2.2.1 Table of Powers 5
        2.2.2 Modular Exponentiation 6
        2.2.3 Montgomery Modular Multiplication 6
3. Design 7
    3.1 Modelling 7
    3.2 Original Design and Methodology 7
4. Optimizations 8
    4.1 NEON Intrinsics 8
    4.2 General Function Optimizations 8
    4.3 Optimizing the MMM Algorithm 9
    4.4 Predicate Operations 9
    4.5 Realizations from Reviewing Original Pseudocode 9
    4.6 Loop Unrolling and Pipelining 10
5. Proposed Hardware Assist 10
    5.1 Selected Area 10
    5.2 Proposed Hardware Logic 11
    5.3 Inline Assembly 12
    5.4 Performance Benefit Estimation 12
6. Performance 13
7. Discussion 15
    7.1 Analysis of Performance Increases 15
    7.2 Challenges 16
8. Conclusion 16
9. References 18

List of Figures and Tables

Figure 1: RSA Encryption / Communication / Decryption Process 6
Figure 2: RSA Encryption / Decryption Logical Flow 7
Figure 3: UML diagram of implemented RSA encrypt and decrypt algorithm 9
Figure 4: Selected area for hardware assist proposal 13
Figure 5: High-level circuit design of proposed hardware assist 13
Figure 6: Inline assembly call to proposed hardware assist 14
Figure 7: Pure software assembly 14
Figure 8: Hardware assisted assembly 14
Figure 9: Charted execution time decreases over development iteration 17
Table 1: Performance tracking over development iterations 16

1. Introduction

For our term project, we were tasked with researching, implementing, and optimizing the Rivest-Shamir-Adleman (RSA) Cryptography algorithm on a 32-bit ARM processor, by analyzing and improving the implementation's assembly-level code to reduce operations and leverage optimization techniques to identify bottlenecks and increase overall performance.

Within the report, the background section will present relevant theory and information behind RSA Cryptography. In the design section, the modelling of our solution, original solution, and optimizations to the solution will be discussed and justified. The discussion section will analyze and review how effective our optimizations were to the original implementation in terms of performance benefits on the 32-bit processor. Finally, conclusions and future work sections will summarize the results presented in previous sections, and present limitations and potential areas for improvement in the future if a larger timeline for the project was feasible.

1.1 Contributions

For the project, each members approximate contributions to the overall final deliverable are listed as follows:

David Bishop

- High level C code implementation
- Report writing & editing
- Software optimizations
- Performance tracking
- Test bench design

- Performance tracking

Benjamin Austin
- High level C code implementation
- Some software optimizations
- Hardware assist design
- Report writing & editing
- Diagram design & creation
- Test bench design
- Performance tracking

1.2 Platform Information and Building Instructions

For our project, a Raspberry Pi 3 B+ was used as our development and test machine. The Raspberry Pi that was used runs Debian 10.8 on a ARMv7 rev4 (v7l) processor. For compilation, the included Raspbian gcc 8.3.0 compiler was used. To build the project code as tested on the machine, the following commands can be used to generate an executable and associated assembly code:

Executable: *gcc -Wall -static -mfloat-abi=hard -mfpu=neon -flax-vector-conversions ./src/main.c -o main*
Assembly: *gcc -Wall -static -mfloat-abi=hard -mfpu=neon -flax-vector-conversions -S ./src/main.c*

To compile the code into an executable, assembly, and an optimized version use *make*. Finally, to execute the code against the test bench, simply run *./test.sh*

1.3 Project Requirements

For our project, there were several requirements that were presented which affected our design process and results. An explanation of the specification and requirements follows below.

1.3.1 Hardware Requirements

- Develop the solution for use with a 32-bit operating system.
- Implement solution for use on the ARM processor.

1.3.2 Specification Requirements

- Implement RSA Cryptography based on Modular Exponentiation and Montgomery Modular Multiplication Algorithm.
- Implement RSA encrypt, decrypt and modulo keys with a maximum bit length of 128 bits.

- Apply software optimization techniques presented in class in combination with a test bench to evaluate and improve the performance of the implementation.
- Identify an area of the program that could benefit from a proposed hardware assist and estimate the performance benefit of the proposed hardware.

2. Background

In this section, a background of RSA Cryptography, its applications, and methods of approaching the algorithm will be discussed.

2.1 RSA Cryptography

With the world increasingly shifting towards reliance on sensitive data transmitted over the internet, individuals and corporations are increasingly susceptible to being targeted by malicious attackers attempting to intercept their sensitive data as it travels around the world in the form of data packets. With more devices that send and receive sensitive information being introduced to the internet each day, the importance of privacy is becoming very apparent. A common method of securing one's information as it travels the networks of the world is to encrypt it during the transmission period. Encryption renders your data useless to malicious attackers by shuffling and modifying the original contents.

Rivest-Shamir-Adleman (RSA) Cryptography is a public-key cryptosystem that was first published in 1977 [1]. Public-key cryptography is a cryptographic system that uses a pair of keys: public keys, which may be known to others, and private keys, which may remain in the sole knowledge of the owner. The public key is used to encrypt the message before transmission, whereas the private key is used to decrypt the message upon arrival. The layout of this communication can be observed in Figure 1.

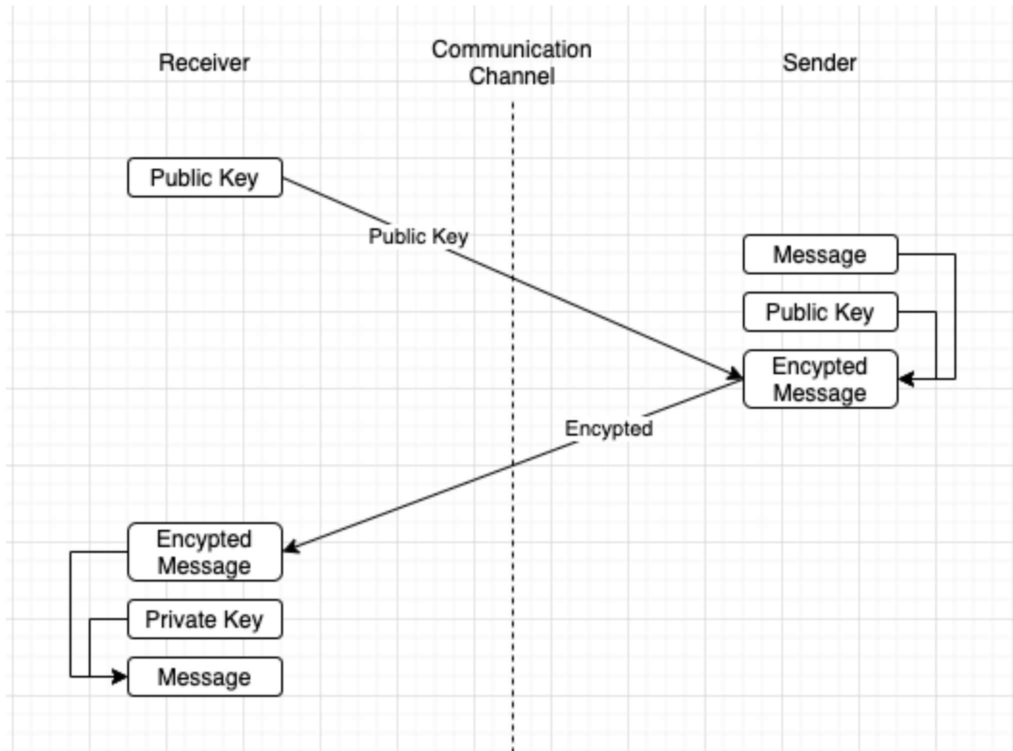

Figure 1: RSA Encryption / Communication / Decryption Process

RSA encryption primarily relies on choosing the public and private keys properly, and the efficient computing of large exponents. To choose the public and private keys, the following steps are taken:

1. Select variables *P* and *Q,* such that both are large prime numbers.
2. Select *E* such that $E > 1$ and $E < PQ$
    a. *E* and *(P - 1)(Q - 1)* are relatively prime.
    b. *E* is odd
3. Compute *D* such that *(DE - 1)* is evenly divisible by *(P - 1)(Q - 1)*. To accomplish this, an integer *X* must be chosen which causes $D = \frac{(X(P-1)(Q-1)-1)}{E}$ to be an integer.

After the successful calculation of these values, the public key is produced with the pair (*PQ, E)*. The private key is *D*. The encryption function based on these values is as follows, with *C* being the ciphertext, and *T* being the plaintext.

$$C = T^{E} mod\ (PQ)$$

The decryption function is as follows, with *C* being the ciphertext, and *T* being the plaintext.

$$T = C^{D} mod\ (PQ)$$

As we can see from the previous equations, the real challenge of the two functions is the calculation of computing the large exponential values is an efficient time range, as with large values of *E* and *D,* both computations become very complex. A UML diagram of the encrypt and decrypt process is shown in Figure 2 below.

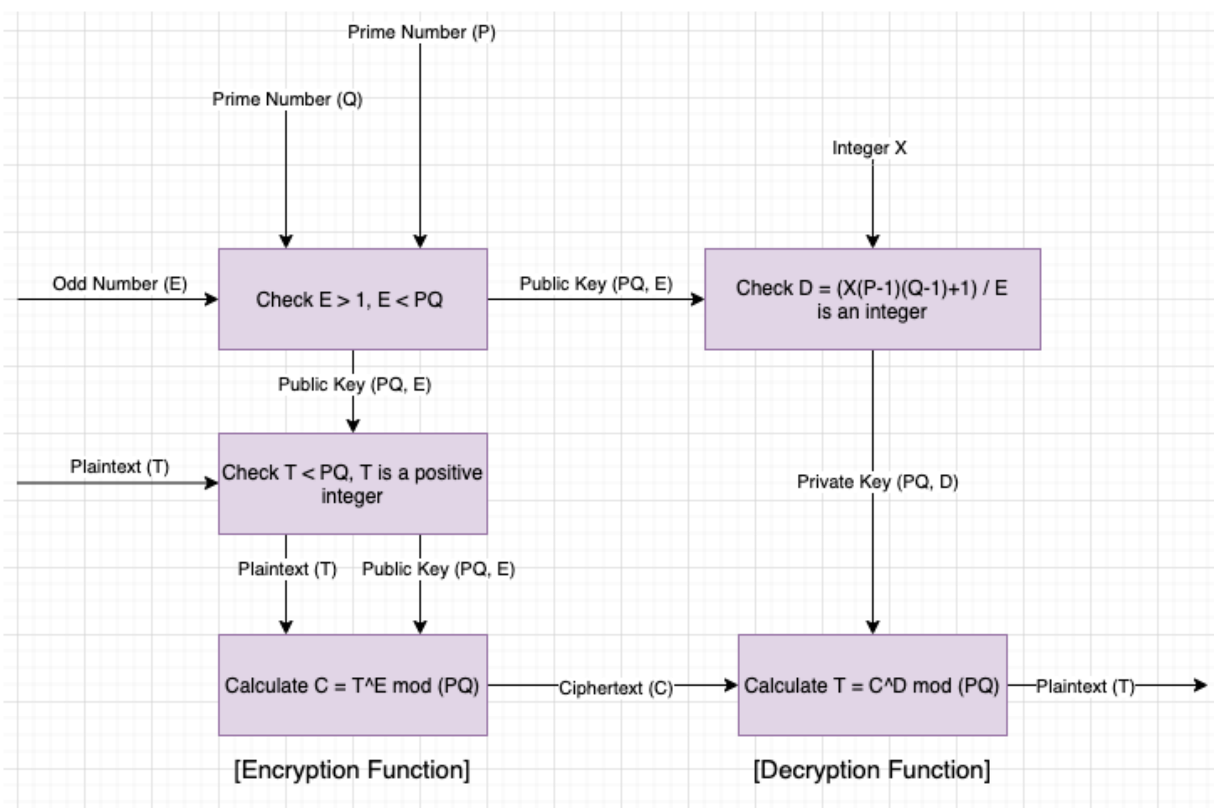

Figure 2: RSA Encryption / Decryption Logical Flow

2.2 Solution Approaches

In this section, different approaches to computing the equations associated with RSA will be presented and compared.

2.2.1 Table of Powers

While the table of powers method can be computed before the encryption / decryption is set to take place, the sheer amount of space that this method requires instantly disqualifies it from being a valid option for our project. The table of powers method is very similar to the Modular Exponentiation method, with the difference being that the values are pre-computed in a table. This method requires less real-time computation, but at the cost of an unacceptable amount of memory being used.

2.2.2 Modular Exponentiation

Modular Exponentiation (ME) is an approach that improves upon the table of powers method by computing the value $c = b^e (mod\ m)$. c is computed iteratively by applying the modular exponential algorithm, while at each stage, the modulo function is performed to keep any intermediate variables within the integer range of m. Another method is to allow the intermediate variables to grow and perform the modulo operation as a single final step. The first method is more desirable because it keeps the multiplication functions down to a practical bit width. This algorithm can be seen below [2].

```
Z_0 = 1
P_0 = X
FOR ( i = 0 to n - 1 )
        P_{i+1} = P_i^2 mod M
        IF ( e_i = 1 )
                Z_{i+1} = Z_i P_i mod M
        ELSE
                Z_{i+1} = Z_i
END FOR
```

2.2.3 Montgomery Modular Multiplication

Due to the expensive nature of the modular operation in programming languages it is unwanted in most optimized systems. Unfortunately, as shown above in, lines three and four of the pseudo code include the modular operation. Montgomery Modular Multiplication (MMM) is an efficient algorithm to compute $Z = X * Y * R^{-1} mod\ M$. The pseudo code for this algorithm is shown below [2].

```
T = 0
FOR ( i = 0 to m - 1 )
        η = T(0) XOR (X(i) AND Y(0))
        T = (T + X(i)Y + ηM) >> 1
END FOR
IF ( T ≥ M )
        T = T − M
Z = T
```

The main performance benefit of this approach comes from the fact that any expensive mathematical operators, such as division and modulo, are performed by using powers of two. Consequently, any division or modulo computations can be transformed into bitwise operations

[3]. However, due to the scalar of R^{-1} in the computed value from MMM the input values X and Y need to be pre scaled with R^2 to have the result Z be scaled with R as well. To remove the scale at the end of the process, Z and one must be put into MMM to finally remove the scalar of R. This process is shown below [2].

$$\overline{X} = X \cdot R^2 \cdot R^{-1} mod\ M = X \cdot R\ mod\ M$$

$$\overline{Y} = Y \cdot R^2 \cdot R^{-1} mod\ M = Y \cdot R\ mod\ M$$

$$\overline{Z} = (X \cdot R) \cdot (Y \cdot R) \cdot R^{-1} mod\ M = X \cdot Y \cdot R\ mod\ M$$

3. Design

For our implementation, several steps were taken to translate the provided pseudocode into an executable program, while meeting the previously defined specification requirements for the project.

3.1 Modelling

As a starting point, figure 3, a UML diagram of the RSA encryption and decryption algorithm was created to represent the logical flow of the encryption and decryption process.

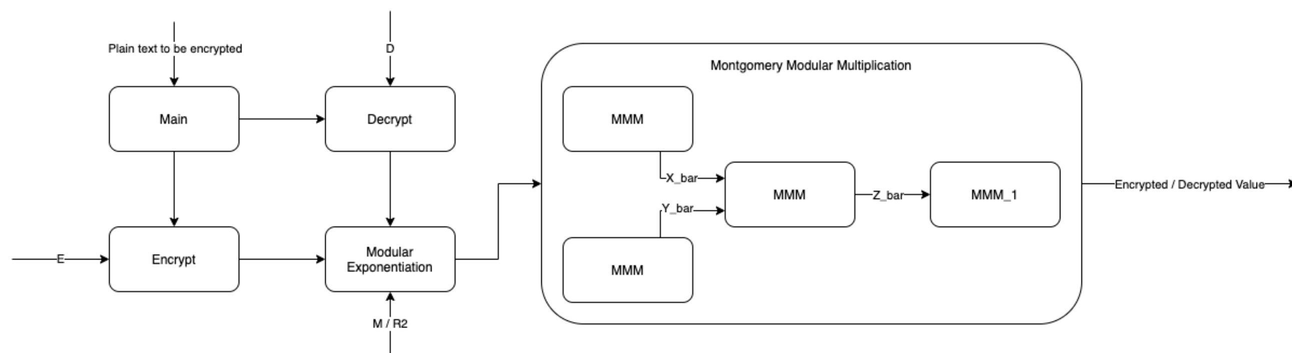

Figure 3: UML diagram of implemented RSA encrypt and decrypt algorithm.

3.2 Original Design and Methodology

After researching and analyzing the different methods of approaching the solution, Modular Exponentiation with Montgomery Modular Multiplication was determined to be the most optimal solution, with the most areas for pure software improvement.

To implement the Modular Exponentiation, Montgomery Modular Multiplication, and other associated functionality, the provided lecture slides on RSA cryptography [2] were referenced as a guide. Firstly, the basic implementation was written on our personal 64-bit machines with standard *long long int* data types as a proof of concept, and to verify simple functionality of the encryption and decryption algorithms when tested with known inputs and outputs.

Next, we were required to adapt the code to accommodate support of 128-bit encryption, decryption, and modular keys plus all relevant values associated with the algorithms. At first, we considered using a struct to hold the two halves of the 128-bit integers in the form of two 64-bit *unsigned long long int* data types. After implementing a test of this in C, we analyzed the assembly code and concluded that using a struct resulted in too much processing overhead and excess memory use, which would be detrimental when performing many operations on the two halves of the 128-bit integer. From here, we decided to represent the 128-bit integers by using a two-long array consisting of two 64-bit *unsigned long long int* data types to represent the upper and lower halves of the integer. This method of representation was used for the first steps of the optimization process, but later replaced by NEON intrinsic, discussed in section 4.1.

4. Optimizations

To optimize the code, several software optimization techniques were applied to key locations in the code to help the performance of the implementation.

4.1 NEON Intrinsics

NEON Intrinsics is a set of intrinsic functions known to the gcc compiler used for ARM processors to optimize vector operations through large 128-bit *neon registers* that are separate from the main registers of the CPU. Many of these operations on the vectors can be executed in parallel to improve efficiency and execution speed.

NEON's vector operations replaced our first design choice to have an array of two 64-bit *unsigned long long integers*. This improved performance of the software significantly because of the parallelism from NEON's vector operations and reducing the number of load and store operations in the assembly code.

4.2 General Function Optimizations

When optimizing software, in c there are specific keywords that affect how the compiler compiles the code to make the assembly more efficient. The *register* keyword is applied to variables to notify the compiler this variable should be kept in a register as much as possible. This is useful with variables used in loops because they are accessed often and would add overhead to the loop if the iteration variable is moved in and out of registers each loop.

Next, the *static inline* keywords are applied to simpler functions to notify the compiler that the code within the function can be placed at the point the function is called. This is applied to most of the wrapper functions our team wrote to operate on the NEON vectors to increase performance of the software.

Next, our team reorganized for loops to reduce expensive comparison instructions. This is done because in the standard for loop, an integer *i* is initialized to zero and comparing its value to some number *num* where it stops looping if *i* is greater than or equal to *num*. The comparison if i is less than num is a costly operation in assembly. Instead, we rearrange the loop to count down from *num* and instead loop until *i* is zero. A comparison of *i* and zero is found to be much more efficient and is a significant performance enhancer.

Lastly, for a short time our team's software had the encryption, decryption and modular keys as global variables. At each use of the keys the assembly code would have to access the main memory to retrieve the value. Instead at each iteration of ME, we adjusted this to make a local copy of the keys at each function call to reduce memory loads and keep the values in registers. Fortunately, our team was also able to remove the global variables and instead keep the values as macros and make a new vector at each call to encrypt or decrypt.

4.3 Optimizing the MMM Algorithm

As discussed before in section 2.2.3, the MMM algorithm must be executed to pre-scale and de-scale the operands and result. When de-scaling the resulting number must be put into MMM with a constant 1. Our team found that we can simplify the function significantly because we can anticipate the bits of X to simplify operations. Therefore, our team created a new method *MMM_1* to perform MMM but with a constant 1.

4.4 Predicate Operations

While predicate operations are more efficient then branching. Unfortunately, due to our use of NEON Intrinsics to represent 128-bit integers the vector operations do not support predicate operations, therefore, in many cases the assembly code must use branches.

4.5 Realizations from Reviewing Original Pseudocode

To start the project our team understood the pseudo code from the MMM slides to mean that for each instance of the modulo operator the MMM algorithm must run four times to pre-scale, multiply and descale the result. After rereading the slides, we realized that we can simply pre-scale X and Y operands to the ME function, execute a MMM function in place of each modulo operation and descale Z as the final step. This can be seen in Table 1 as the difference between

iteration 7 and iteration 8 is a factor of almost 3x faster. This does not decrease instruction count by much, however, reduces redundancy and makes the resulting software much more efficient.

4.6 Loop Unrolling and Pipelining

Due to the linearity of the ME and MMM functions, our team found that the loop unrolling and pipelining optimizations they did not increase performance of the software. In this case if our team attempts to loop unroll the software, we found that when compiled to assembly many more branch operations and loops were created, this caused the software to be slower in encrypt and decrypt averages. Additionally, our team found that the software could not be pipelined properly. This is due to the specific sequence of bit checks that need to be done and the order of operations through the function that limited the possibility of rearranging operations to increase linearity.

5. Proposed Hardware Assist

As part of the optimization process of the project, there was a requirement to suggest a hardware assist and estimate its performance benefit to the respective section of the code. For the purposes of this requirement, all assists were designed and implemented assuming a 32-bit context for simplicity.

5.1 Selected Area

For our proposed hardware assist, we determined that we would focus on attempting to increase the performance of a section of code that is in a loop, and that has simple input and output formats that are easily adaptable to the supported 2 x 32-bit inputs and single 32-bit output for the User-Defined Unit that replaces the Arithmetic Logic Unit. We selected the area inside of the loop in the *MMM_1* method. This method is used to remove the scale factor of R in the MMM process, by computing MMM with a scale factor of 1. The specific loop inside this method that we selected iterates over the bit length of M. The inner portion of this loop consists of performing a logical *AND* on Z, potentially performing an addition, and finally performing a bitwise right shift of 1 before returning the new value of Z.

For the purpose of the hardware assist proposal, the logic of this loop was moved to a new method and re-written to support 32-bit arguments for Y, M, and Z, although Y is not used in this demonstration, as it is not required by any section of the inner loop body. Related calls to helper methods were removed and replaced with their respective logic. The creation of this demonstration method can be seen below in figure 4.

```
//desribes one loop iteration inside MMM_1 in a 32-bit context
void MMM_1_ITERATION(register unsigned int Y, register unsigned int M) {
    register unsigned int Z;
    if(Z & 1) {
        Z = (Z + M);
    }
    Z = (Z >> 1);
}
```

Figure 4: Selected area for hardware assist proposal.

5.2 Proposed Hardware Logic

To replace the pure-software implementation of this loop body, a solution in hardware was drafted and described using the following high-level circuit diagram shown in figure 5.

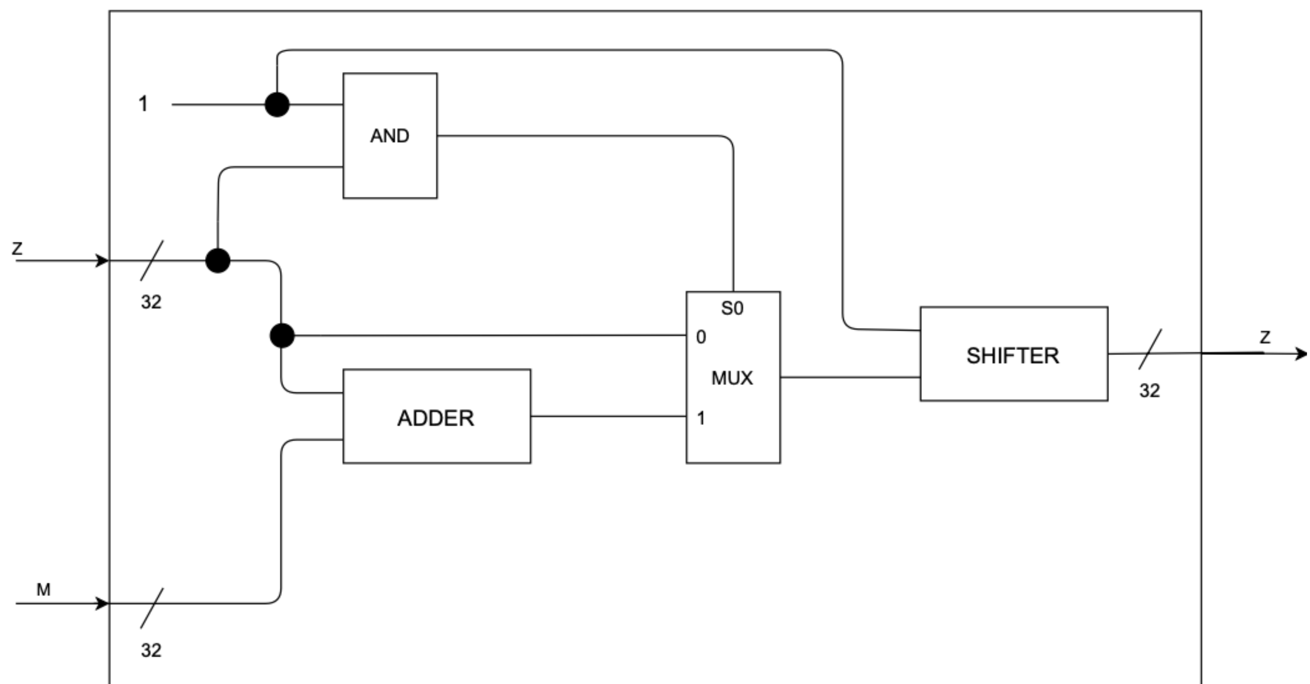

Figure 5: High-level circuit design of proposed hardware assist.

All components of this system are 32-bit components and have been abstracted for simplicity of explanation. For performance estimations, we can assume that the AND requires 1/32 of a clock cycle, the ADDER requires 1 clock cycle, the MUX requires 4/32 of a cycle, and the SHIFTER requires no clock cycles, as it is akin to re-wiring, since we are shifting a constant amount a single time. For a safe estimation, we can also add some overhead for any register operations or

other environment variables and can predict that the hardware assist will take 3 clock cycles to complete.

5.3 Inline Assembly

To integrate the proposed hardware assist into the code, inline assembly can be used to simulate the call of the unit, which will be referred to as *mmm_1_hw*. The inline assembly is shown below in figure 6.

```
//desribes one loop iteration inside MMM_1 in a 32-bit context
void MMM_1_ITERATION(register unsigned int Y, register unsigned int M) {
    register unsigned int Z;

    __asm__ __volatile__ (
    "mmm_1_hw\t%1, %2, %0\n"
    : "=r" (Z)
    : "r" (Y), "r" (M)
    );
}
```

Figure 6: Inline assembly call to proposed hardware assist.

5.4 Performance Benefit Estimation

To estimate the performance benefit of the proposed hardware implementation, we can first compare the generated assembly for the pure software implementation and the inline assembly implementation of the single-iteration example method of the loop body. The generated assembly for the pure software implementation can be seen in figure 7, while the generated assembly for the proposed implementation with inline assembly can be seen in figure 8.

```
    mov r2, r1      // 1 Clock Cycle
    add fp, sp, #4  // 1 Clock Cycle
    and r3, r4, #1  // 1 Clock Cycle
    cmp r3, #0      // 1 Clock Cycle
    beq .L60        // 3 Clock Cycle
    add r4, r4, r2  // 1 Clock Cycle
.L60:
    lsr r4, r4, #1  // 1 Clock Cycle
```

Figure 7: Pure software assembly

```
    add fp, sp, #4              // 1 Clock Cycle
    mov r3, r1                  // 1 Clock Cycle
    .syntax divided
@ 100 "./src/RSA.h" 1
    mmm_1_hw    r4, r3, r3 -3c  // 3 Clock Cycle

@ 0 "" 2
```

Figure 8: Hardware assisted assembly

As we can see above, the relevant assembly portion for the pure software implementation typically contains 9 clock cycles, depending on if the branch is taken, whereas the hardware implementation with inline assembly contains 5 clock cycles. In terms of purely comparing the

amount of clock cycles, we can see that this proposed hardware solution provides almost a 2x speedup compared to the pure software implementation. While this demonstration was performed in a 32-bit context, we can assume that the speedup will translate to a 128-bit implementation, making it worthwhile to further investigate the use of a hardware assist for this specific area of the RSA algorithm.

6. Performance

To investigate performance benefits of optimization practices, and to keep track of the code performance throughout the development process, performance was measured as often as possible for each major change. The main performance metric of concern during the term was performance, as in embedded systems, when working with limited computational resources, speed is one of the most important factors and metrics for the software.

In order to analyze the performance of the code, a test suite was created to feed a set of predefined inputs to the code in question. For our project, the test suite consisted of encrypting and decrypting a collection of 2000 randomly generated integers in the range [1, 1000]. The test suite runs the encryption and decryption of these 2000 identical integers to collect a large sample size of execution times, measured with the *clock()* library function. Partway through the development process, the test suite was modified to use a set of 2000 randomly generated ASCII strings of length 15, along with increasing the pre-set values for M, D, and E. This was done to increase the workload on the program to collect more detailed results over a longer runtime and exercise the full 128-bit capability of the algorithms.

The *main* executable handled the file reading & writing, along with the timings, and remained un-optimized and un-changed throughout the course of the project's development, therefore we could simply dismiss the runtime of this portion of program as an overhead and could focus on the performance optimizations of the code that was directly related to the RSA algorithm. The resulting times were written to a file and analyzed to collect the total loop time, average, minimum and maximum encryption and decryption times for the test suite. From these results, we were able to document our performance increases throughout the development of the project, and record them in a table, as seen in Table 1. For the performance analysis, we took a sum of the average encryption and average decryption for each iteration to get a "round-trip" value for the program. All performance increase percentage calculations were based on this value.

Iteration	**Inputs**	**Description**	**Average Encrypt & Decrypt Combined (s)**	**Performance % Increase Compared to Iteration 0**	**Lines of Assembly**
0	2000 Random length 15 ASCII strings	Initial test of 128-bit implementation on Raspberry Pi.	0.047478	N/A	1118
1	2000 Random length 15 ASCII strings	Moved constants for M, R^2, E and D to RSA.h and used local variables in all methods	0.036677	22.7495%	1211
2	2000 Random length 15 ASCII strings	Implement NEON Intrinsics	0.029349	38.184%	1028
3	2000 Random length 15 ASCII strings	Adjustments to *register* integers	0.027993	41.0401%	938
4	2000 Random length 15 ASCII strings	Combine MMM and ME methods	0.023279	50.9689%	920
5	2000 Random length 15 ASCII strings	Split MMM into MMM and MMM_1	0.016904	64.3961%	984
6	2000 Random length 15 ASCII strings	Loop optimizations and function inlining	0.015477	67.4017%	927
7	2000 Random length 15 ASCII strings	General optimizations	0.015452	67.4544%	915
8	2000 Random length 15 ASCII strings	Re-interpret provided pseudocode and update MMM logic	0.005510	88.3946%	903

Table 1: Performance tracking over development iterations.

Additionally, a line chart of the encrypt / decrypt round trip time per iteration was plotted and can be seen in figure 9 below.

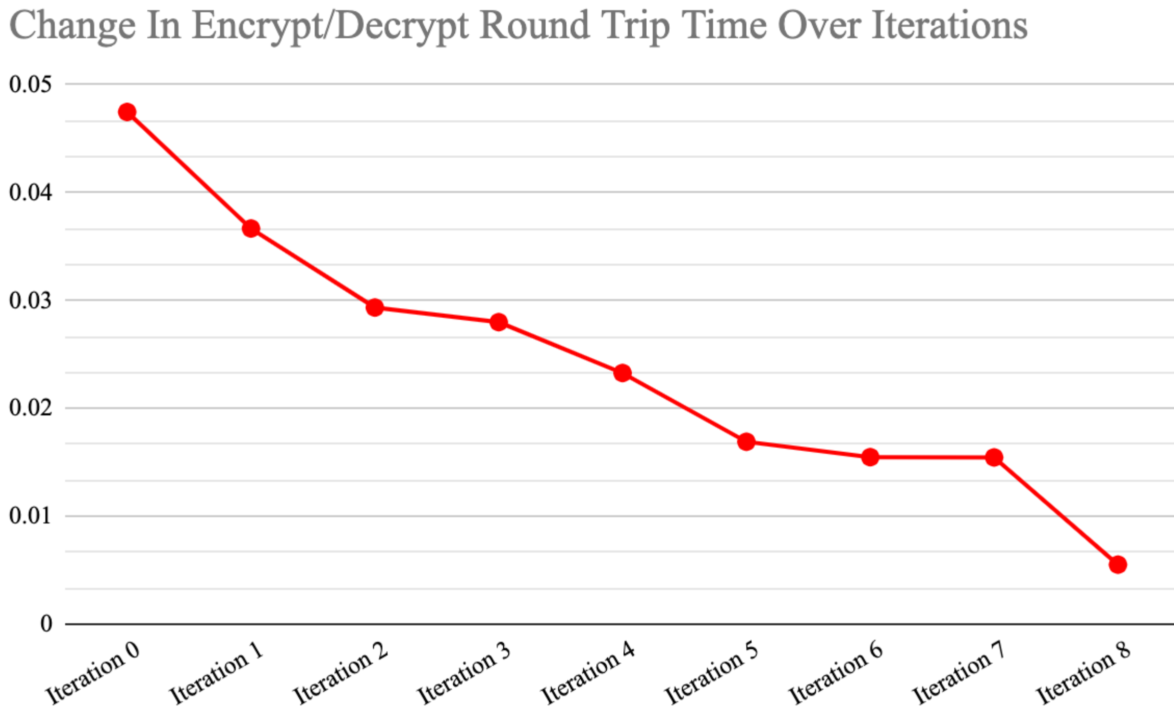

Figure 9: Charted execution time decreases over development iterations.

7. Discussion

In this section, a discussion about the performance increases from optimization techniques will be discussed, along with an overview of issues and limitations that were encountered during the project.

7.1 Analysis of Performance Increases

Starting with an initial implementation with an average round trip encrypt/decrypt time of 0.047478 seconds, an optimized average round trip time of 0.005510 seconds was achieved through the application of several software optimization techniques such as the use of local variables and *register* keywords, loop optimization, function inlining, and NEON intrinsics, and finally realizing an initial misinterpretation of the provided pseudocode. This performance increase in time translates to an 88.4% performance increase over the course of the project development. Additionally, a proposed hardware assist was drafted, and estimated to provide almost a 2x speedup in its specific application location.

7.2 Challenges

During the initial development of the implementation and following optimization efforts, several main issues and difficulties were experienced.

Firstly, as we were working on a 32-bit ARM processor, implementing and optimizing code that dealt with 128-bit integers proved to be challenging for a few reasons. Firstly, we struggled with deciding how to represent these 128-bit integers efficiently. Different solutions such as structs were discussed and evaluated by experimenting with the associated assembly code to understand how the compiler translated our ideas into assembly and determine if they were efficient or not. For our initial iterations, we decided to represent these integers by using a pair of *unsigned long long int* variables, each carrying 64 bits of information, as a way of representing the upper and lower halves of our 128-bit integer.

Secondly, we initially found that implementing the arithmetic functionality for the algorithm, such as functions built to add and subtract, proved to be complicated and difficult when the variables that were being operated on were stored in two separate variables. While the ideal solution that does not employ NEON intrinsics may use four separate *long int* variables to represent the 128-bit integers, as they store 32 bits, we decided that using two 64-bit halves simplified the development of the arithmetic functions associated with the RSA algorithm. Eventually, we were able to use NEON intrinsics to represent the 128-bit integers efficiently.

Additionally, we found it difficult to conceptualize the proposal of a hardware assist for our implementation. Originally, we attempted to propose a hardware solution for a large portion of the system, but quickly realized that doing so would be unnecessarily challenging, especially for two members who have limited experience with circuit design and logic gates. Next, the implementation of a smaller area of the software was attempted, with the relevant 128-bit integers, but due to the input/output register constraints for the 32-bit architecture, we realized that this would also be extremely complicated to implement and propose as a solution. In order to convey our understanding of a proposed hardware assist and inline assembly, we decided to create a new method that contained a simple amount of logic, which was pulled from an existing loop body in the original code. This logic was represented in a 32-bit context for simplicity, and a resulting hardware implementation and inline assembly was designed and demonstrated as a proof of concept.

8. Conclusion

While the process of computing the correct exponentials to perform RSA Encryption is an expensive task for a 32-bit processor, we were successfully able to increase the algorithms' performance on a 32-bit ARM processor. After implementing an initial C implementation of Modular Exponentiation with Montgomery Modular Multiplication we were able to apply

various software optimization techniques presented in the course, such as NEON Intrinsics and loop optimizations. After applying these optimization techniques and running our test suite on the improved implementation, we were able to obtain an average execution time of 0.005510 seconds to encrypt and decrypt a length-15 ASCII string, compared to our initial execution time of 0.047478 seconds before the code was optimized. This translates to an 88.4% improvement through the application of software optimization techniques. Furthermore, we were able to identify and propose a hardware assist for a specific area of the code. The proposed hardware assists were estimated to provide almost a 2x speedup when compared with the pure software implementation of the logic. From this estimation, we can say that it would be worthwhile to further investigate the application of a hardware assist for this area of the program.

While successful, we believe that there are areas for further improvement of the software if project deadlines were not a constraint. For example, the investigation into potential performance benefits through hardware implementations of other components may be worthwhile, as well as further analysis of the C implementation and accompanying assembly instructions to identify areas which could benefit from further improvement. Overall, we believe that our efforts were a success as we were able to apply the techniques presented in this course to improve the performance of the software on a 32-bit ARM processor.

9. References

[1] University of Bristol, “Dr Clifford Cocks CB,” *Dr Clifford Cocks CB | Graduation | University of Bristol*, 06-Apr-2016. [Online]. Available: http://www.bristol.ac.uk/graduation/honorary-degrees/hondeg08/cocks.html. [Accessed: 23-Jul-2021].

[2] M. Sima. SENG 440. Class Lecture, Topic: “Lesson 108: RSA Cryptography” Department of Engineering, University of Victoria, Victoria, BC, May 14, 2021.

[3] “Montgomery multiplication,” *Montgomery Multiplication - Competitive Programming Algorithms*. [Online]. Available: https://cpalgorithms.com/algebra/montgomery_multiplication.html. [Accessed: 04-Aug-2021].