University of Victoria

Department of Engineering and Computer Science

SENG440 Project - Technical Report

Summer 2021

RSA Cryptography Optimization for Embedded Systems

Waltvin Lee

V00894034

Bachelor of Software Engineering (BSEng)

waltvin@uvic.ca

10th August 2021

1 Introduction

This report will cover the final project for the SENG440 – Embedded Systems course offered in the University of Victoria. The project involves implementing and further optimizing the RSA Encryption and Decryption algorithm for an embedded system with a goal for using RSA private and public keys of size 95-bits.

1.1 Objectives

The objective of the project is to first implement the RSA encryption and decryption algorithm for an embedded system for 95-bit RSA keys. Given that the targeted embedded system is running a 32-bit processor, a custom routine that allows for 96-bit operations is required to be written to allow the processor to work efficiently with three 32-bit sized integers.

After implementing the RSA encryption and decryption algorithm for 95-bit RSA keys, optimization should be done to the Montgomery Modular Multiplication algorithm used in the encryption and decryption process to achieve lower cycle counts and faster runtime. The optimization that will be done will mostly be software optimizations. Hardware-assisted solutions will also be analyzed but are not implemented.

1.2 Embedded System Hardware and Processor

A Virtual Machine running Fedora is emulated with QEMU. QEMU allows us to emulate this Virtual Machine with the ARMv7 processor architecture. The code is compiled with the gcc compiler on the Virtual Machine itself. This Virtual Machine will then be used as the machine to test and execute the software.

2 Theoretical Background

2.1 RSA (Rivest–Shamir–Adleman) Public-key Cryptosystem

RSA is a public-key cryptosystem that is used for secure data transmission. In a public-key cryptosystem, the public key is used to encrypt and the private key is used to decrypt information. By publishing the public key and keeping the private key secret, it ensures that anyone can encrypt data with the public key, but only the party that holds the private key can decrypt the encrypted data. If both parties generate RSA pairs of keys and exchange their public keys, then both parties can send encrypted data to the other that only that other could decrypt.

2.1.1 RSA Key Generation

In RSA, the public key is a pair represented as PQ and E, while the private key is represented D. These keys PQ, E, and D are often very large in size, typically 1024, 2048, or 4096 bits to ensure security. The steps to generating these keys can be simplified as 3 steps.

1. PQ from the public key pair is a product of two prime numbers, therefore choose two unique prime numbers P and Q.
2. Find a value of E such that E > 1, E < PQ, and E and (P – 1)(Q – 1) are relatively prime.
3. Find a value of D such that (DE – 1) is evenly divisible by (P – 1)(Q – 1)

By performing these steps to obtain the keys, it will result in no known easy methods of calculating D, P, or Q given only the public key pair PQ and E [1]. However, if the RSA keys are too small, for example 256-bit keys, it can be easily cracked within minutes even with an inefficient method, therefore it is suggested that RSA keys should be at minimum 512-bits.

2.1.2 RSA Encryption and Decryption

After generating the keys, we can then use the keys to perform encryption and decryption steps. We first would require a plaintext that is going to be encrypted. This plaintext is not allowed to be larger than PQ, else RSA encryption would not work.

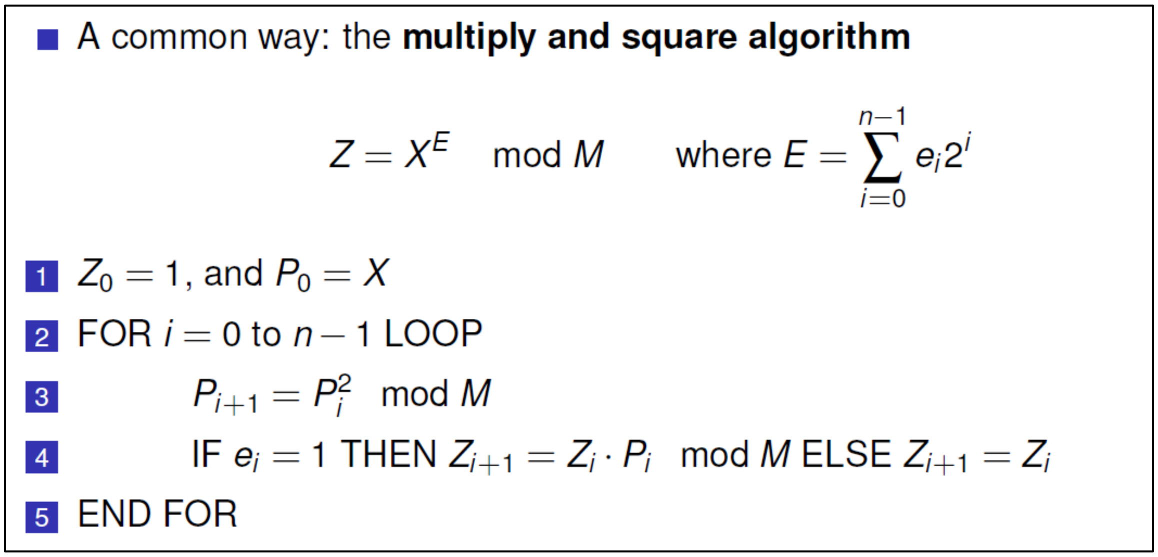
To encrypt a plaintext, the public key pair E and PQ from the public key pair is required.
The encryption formula for RSA is $C = T^{E} \; mod \; PQ$, where C = ciphertext, T = plaintext.

To decrypt a plaintext, the private key D as well as PQ from the public key pair is required.
The decryption formula for RSA is $T = C^{D} \; mod \; PQ$, where C = ciphertext, T = plaintext.

The encryption and decryption process involves modular exponentiation, and this is a computationally intensive process involving many arithmetic operations. Given that RSA keys ideally should be large integers as well, doing this modular exponentiation normally will result in a very slow process. The multiply and square algorithm together with the Montgomery Modular Multiplication algorithm can be used to fasten this modular exponentiation process [1].

2.1.3 Modular Exponentiation

The multiply and square algorithm is used to fasten the modular exponentiation process. This algorithm however involves a significant amount of modular multiplication operations by a loop. Therefore, the Montgomery Modular Multiplication algorithm helps us perform modular multiplication efficiently. Figure 1 below shows the pseudocode given and used for the multiply and square algorithm to achieve modular exponentiation [1].


■ A common way: the **multiply and square algorithm**

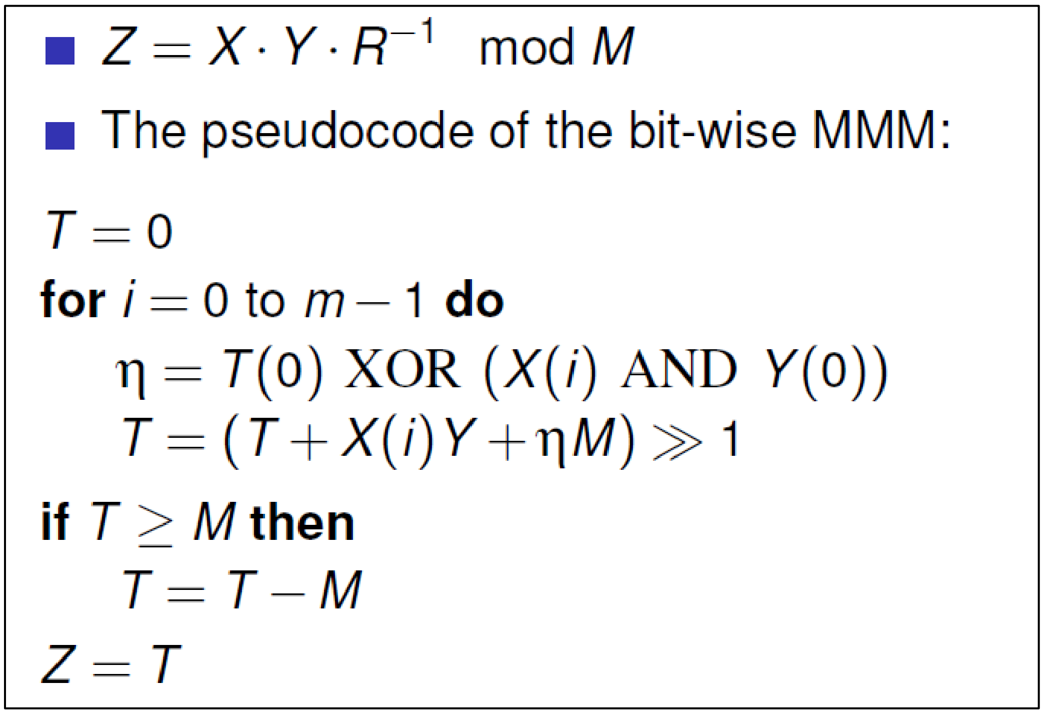
$$Z = X^E \mod M \qquad \text{where } E = \sum_{i=0}^{n-1} e_i 2^i$$

```
1 $Z_0 = 1$, and $P_0 = X$
2 FOR $i = 0$ to $n - 1$ LOOP
3         $P_{i+1} = P_i^2 \mod M$
4         IF $e_i = 1$ THEN $Z_{i+1} = Z_i \cdot P_i \mod M$ ELSE $Z_{i+1} = Z_i$
5 END FOR
```

Figure 1: Screenshot of modular exponentiation pseudocode from the course slides that were referred to for this project [1]

2.1.4 Montgomery Modular Multiplication

Montgomery Modular Multiplication (MMM) is used to optimize this modular multiplication process. MMM calculates $Z = X \cdot Y \cdot R^{-1}\ mod\ M$ efficiently. Figure 2 below shows the pseudocode that was given and referred to for the project [1]. Some optimizations covered in Section 4 of this report will heavily refer to operations and variables in the pseudocode in this figure.



- $Z = X \cdot Y \cdot R^{-1} \mod M$
- The pseudocode of the bit-wise MMM:

```
$T = 0$
**for** $i = 0$ to $m - 1$ **do**
    $\eta = T(0)\ \mathrm{XOR}\ (X(i)\ \mathrm{AND}\ Y(0))$
    $T = (T + X(i)Y + \eta M) \gg 1$
**if** $T \geq M$ **then**
    $T = T - M$
$Z = T$
```

Figure 2: Screenshot of MMM Pseudocode from the course slides that were referred to for this project [1]

Notice that for MMM, there is R^{-1} being multiplied together, where $R = 2^{(bit-wdith\ of\ M)}$. To achieve modular multiplication without R^{-1}, the operands X and Y each needs to be pre-scaled up by performing MMM with the multiplication operand R^2, giving $\bar{X} = X \cdot R\ mod\ M$ and $\bar{Y} = Y \cdot R\ mod\ M$.

The result then will also be scaled up by R, as $\bar{Z} = (X \cdot R) \cdot (Y \cdot R) \cdot R^{-1} \mod M = X \cdot Y \cdot R \mod M$. Then, the final step is to perform MMM again with multiplication operands $\bar{Z}$ and 1, resulting in $Z = \bar{Z} \cdot 1 \cdot R^{-1} \mod M = X \cdot Y \mod M$ [1].

3 95-Bit Length Strategy

Given that the project's goal is to target 95-bit integers for public key pair E and PQ, and also private key D, a custom struct to hold 96-bits and its routines is created.

3.1 ***uint96_t*** – 96-bit Struct and its Operations

Given that the ARMv7 processor is a 32-bit processor, with the goal to work with 95-bit unsigned integers, a custom struct called ***uint96_t*** was created. This is a struct that consists of three 32-bit unsigned integers, making up to 96-bit in total. Custom routines will have to be implemented for this ***uint96_t*** type. For RSA Encryption and Decryption, only the following 96-bit operations functions are required.

- Addition – ***add_uint96(uint96_t, uint96_t)***
- Subtraction – ***sub_uint96(uint96_t, uint96_t)***
- Larger than or Equals Comparison - ***cmp_uint96(uint96_t, uint96_t)***
- Right-Shift by variable **-** ***rshift_uint96(uint96_t, int)***

The MMM optimization covered in Section 4.2 narrows the right-shift operation to only requiring right-shifts by 1 or 32, and Section 4.4 replaces the right-shift by variable function ***rshift_uint96(uint96_t, int)*** to two right-shift by 1 or 32 functions ***rshift1_uint96(uint96_t)*** and ***rshift32_uint96(uint96_t)***.

In the MMM pseudocode, there is multiplication required but not included in the above 96-bit operations. This is because operator strength reduction can be made to avoid the expensive multiplication step and the effort of writing a routine for 96-bit multiplication. This optimization is covered in Section 4.1. Other 96-bit operations that are relatively simple such as obtaining the i-th bit or 0-th bit for ***X(0), X(i),*** or ***Y(0)*** are performed in-line.

3.2 Overflow of T In Montgomery Modular Multiplication

In MMM, there is a possibility of an overflow during the summation procedure ***(T + X(i)Y + nM)***. Based on testing with 95-bit values, the maximum overflow will only be 1. However, when testing with 96-bit values, the maximum overflow reaches as high as 3.

An initial approach to solving the overflow was to create an additional carry flag, as it was thought that the maximum overflow would only be 1. However, it was unsure as to what the maximum overflow would be, or whether there was a certain test case with 95-bit values which results in an overflow being larger than 1. Therefore, the carry flag was changed to an ***uint32_t*** type that serves as an additional 32-bit unsigned integer to extend the original 96-bit T to 128-bit.

4 Software Optimizations

4.1 V0 – MMM, Avoiding Multiplication Operation

When initially beginning to implement a multiplication operation for 96-bit integers, it was realized that the multiplication operands ***X(i)*** and ***n*** can only be 0 or 1. Therefore, the multiplication operation for 96-bit integers is not required at all, and this would avoid expensive 96-bit multiplication operation calls.

The replacement to handle multiplying 0 or 1 is simple, if conditions were used to check if it's a 0 or 1, if 0, the result will be 0, else the result will be the other operand. By doing this, we have successfully performed operator strength reduction.

4.2 V1 – MMM, Avoiding loop counter ***i*** bit shifts in ***rshift_uint96***

The for loop in the MMM algorithm uses an ***int i*** counter. This ***int i*** is accessed frequently by the ***rshift_uint96*** function to obtain ***X(i)*** by performing a right shift by ***i***-bit. This is problematic because if ***int i*** that ranges from 0 to 96 is used as an argument for the ***rshift_uint96*** function, the compiler seems to not in-line the function, and the function routine will need to be called at the assembly level, and this means there will be overhead with branching and inefficient CPU pipelining.

Therefore, an approach was taken to avoid using the ***int i*** loop counter as the argument for the right shifts. This is done by right shifting ***X*** by 1 for every loop iteration and getting the 0-th bit every time ***X(i)*** is required. After this change, there was a significant reduction to the cycle count and runtime as now only right shifts by constants 1 or 32 is required by the optimized algorithm. The reason why right shifts of 32 is required will be covered in the following section.

4.3 V1 – MMM, Storing relevant 32-bit chunks of ***uint96_t***

Since ***uint96_t*** consists of three ***uint32_t*** variables, it will naturally require three registers to be stored. An approach that was taken was to try to only keep the relevant 32-bit chunks (high 32-bit, mid 32-bit, or low 32-bit) of the ***uint96_t*** variable into a ***uint32_t*** variable and use that 32-bit chunk instead. This was done to the MMM algorithm for the variable ***X,*** since only the ***i***-th bit is required from ***X,*** thus only the relevant 32-bit chunk containing the ***i***-th bit is kept in the ***uint32_t*** type**.**

This is done to hopefully help the compiler reduce the register usage of ***X*** from three registers to one register when performing the loop iterations. This change has also resulted in reducing the number of 96-bit right shift operations. Instead of shifting ***uint96_t*** by 1 per loop iteration, we are now only shifting only ***uint32_t*** by 1 per loop iteration. Only for the 32nd, 64th, and 96th loop iteration, shifting ***uint96_t*** by 32 is done.

Since ***Y(0)*** is used in every loop iteration and never changes, ***Y(0)*** is also stored into a ***uint32_t*** type and later added with a register keyword to hopefully hint the compiler that it is a good idea to store ***Y(0)*** in a register.

Code snippets 1 and 2 below will show the MMM function before and after performing optimizations.

```c
uint96_t mmm(uint96_t X, uint96_t Y, uint96_t M, uint32_t numBits)
{
    uint96_t T_low = {0};
    uint32_t T_high = 0;

    int m = numBits;

    for (int i = 0; i < m; i++)
    {
        uint32_t n = (T_low.value[2] & 1) ^ ((rshift_uint96(X, i).value[2] & 1) & (Y.value[2]
& 1));

        if (rshift_uint96(X, i).value[2] & 1){
            T_low = add_uint96(T_low, Y);

            if (cmp_uint96(T_low, Y)){
                T_high += 1;
            }
        }

       if (n){
            T_low = add_uint96(T_low, M);

            if (cmp_uint96(T_low, M)){
                T_high += 1;
            }
        }
        T_low = rshift_uint96(T_low, 1);
        T_low.value[0] += T_high << 31;
        T_high >>= 1;

    }

    if (T_high & 1 || !cmp_uint96(T_low, M)){
        T_low = sub_uint96(T_low, M);
    }

    return T_low;
}
```

Code Snippet 1: 96-bit MMM algorithm before optimization

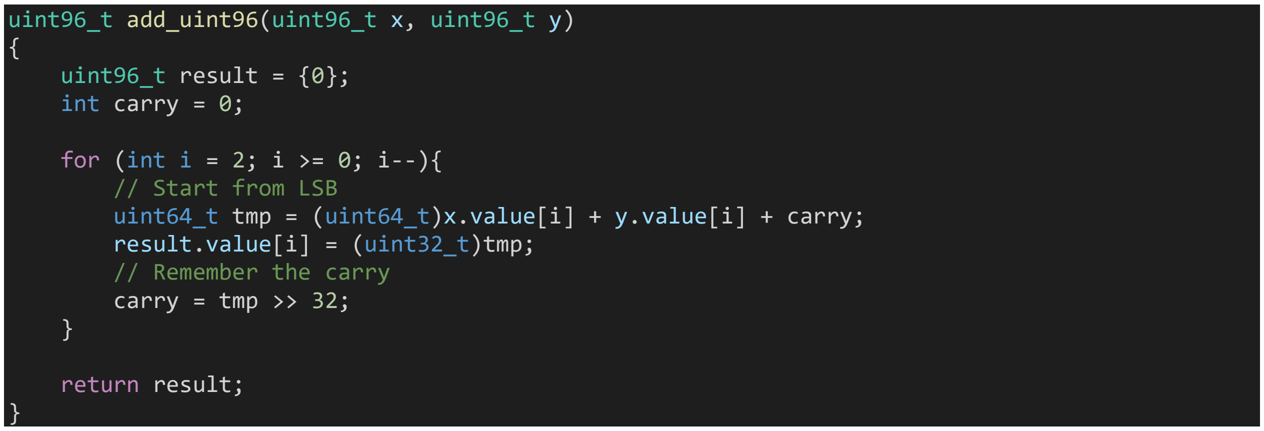
```c
uint96_t mmm(uint96_t X, uint96_t Y, uint96_t M, int numBits)
{
    register uint96_t T_low = {0};
    register uint32_t T_high = 0;
    register uint32_t n;
    register uint32_t X_32local;
    register uint32_t Y_0 = Y.low & 1;

    for (int i = 0; i < numBits; i++)
    {
        if(!(i % 32)){
            X_32local = X.low;
            X = rshift32_uint96(X);
        }
        n = (T_low.low & 1) ^ ((X_32local & 1) & Y_0);

        if (X_32local & 1){
            T_low = add_uint96(T_low, Y);

            if (cmp_uint96(T_low, Y)){
                T_high += 1;
            }
        }

        if (n){
            T_low = add_uint96(T_low, M);

            if (cmp_uint96(T_low, M)){
                T_high += 1;
            }
        }

        T_low = rshift1_uint96(T_low);
        T_low.high += T_high << 31;
        T_high >>= 1;

        X_32local >>= 1;
    }
    if (T_high & 1 || !cmp_uint96(T_low, M)){
        T_low = sub_uint96(T_low, M);
    }
    return T_low;
}
```

Code Snippet 2: Final v4 96-bit MMM algorithm after all optimizations

4.4 V2 – uint96_t, Loop Unrolling and Removal of Internal Array Memory Access

Initially, ***uint96_t*** was a struct class which consists of an array of three ***uint32_t*** elements, giving a total of up to 96-bits to work with. This array however caused the register keyword to be unusable for ***uint96_t*** types if the 32-bit chunks are indexed internally in a loop, because the compiler cannot store arrays in registers if the address of the array is fetched and are forced to be stored in memory.

For example, for the addition function below in Code Snippet 3, the variable result will not be allowed to be stored in a register, and this will significantly increase the number of load and store operations, as this function is frequently used. The reason is because the internal array indexing for ***uint96_t*** types in the for loop requires accessing the address of the array. It would make sense in this case for arrays to be stored in memory and not in registers, as they require contiguous memory storage. Adding a register keyword for ***result*** for example will cause a compilation error stating **Error : "address of register variable 'result' requested"**.



```
uint96_t add_uint96(uint96_t x, uint96_t y)
{
    uint96_t result = {0};
    int carry = 0;

    for (int i = 2; i >= 0; i--){
        // Start from LSB
        uint64_t tmp = (uint64_t)x.value[i] + y.value[i] + carry;
        result.value[i] = (uint32_t)tmp;
        // Remember the carry
        carry = tmp >> 32;
    }

    return result;
}
```

*Code Snippet 3: **add_uint96** function before loop unrolling and **uint96_t** modification*

If we perform loop unrolling however, it seems that the compiler is then able to store the array in registers, as the registers are not indexed internally in a loop based on loop counter ***i***, instead indexed by constants 0, 1 or 2.

However, to prevent this from happening in the future, the ***uint96_t*** struct was changed, replacing the array to 3 different ***uint32_t*** type variables called low, mid, and high, representing the low 0-31 bits, mid 32-64 bits and high 65-96 bits. Loop unrolling is then also performed in ***add_uint96*** and ***sub_uint96***. Code Snippet 4 below shows this optimization change to ***add_uint96***.

```
uint96_t add_uint96(uint96_t x, uint96_t y)
{
    uint96_t result = {0};
    int carry = 0;

    // Optimization: Loop Unrolling
    // Start from LSB
    uint64_t tmp = (uint64_t)x.low + y.low + carry;
    result.low = (uint32_t)tmp;
    // Remember the carry
    carry = tmp1 >> 32;

    tmp = (uint64_t)x.mid + y.mid + carry;
    result.mid = (uint32_t)tmp;
    // Remember the carry
    carry = tmp2 >> 32;

    tmp = (uint64_t)x.high + y.high + carry;
    result.high = (uint32_t)tmp;

    return result;
}
```

*Code Snippet 4: **add_uint96** function after loop unrolling and **uint96_t** modification*

The compiler can now decide to store the three ***uint32_t*** variables in registers, and that the register keyword can now be used without errors.

4.5 V2neon – uint96_t, Neon Operations

An attempt to use Neon operations was made. The ***add_uint96*** function was changed to use Neon operations to perform addition internally. This is done by storing the 96-bit ***uint96_t*** operands into an array of four ***uint32_t*** elements, and then loading the array of 4 ***uint32_t*** elements to ***uint32x4_t***. The Neon addition is then used to perform the addition between the two ***uint32x4_t*** operands, and further handling of overflow is required. Each lane of the ***uint32x4_t*** result is then extracted to a ***uint96_t*** which is then returned.

This approach caused an increase to the cycle count and longer runtime. This is probably because of the overhead of loading and storing to and from Neon datatypes or registers are too costly. Therefore, Neon operations are not further pursued.

A further consideration is to replace all ***uint96_t*** types with ***uint32x4_t*** in the entire program. This approach allows to entirely work in Neon datatypes or registers. However, this will waste 32-bits or one register for each 96-bit integer as Neon registers are 64-bit or 128-bit sized, and this consideration was not further pursued.

4.6 V3 – rshift_uint96, Handling right shifts by 1 and 32

rshift_uint96 no longer requires an operand of ***int i*** to specify the number of bits to shift, as the MMM optimization from Section 4.2 resulted in only requiring right shifts by 1 or 32 for ***uint96_t***. Therefore, an approach to rewrite the function of ***rshift_uint96*** to two separate functions ***rshift1_uint96*** and ***rshift32_uint96*** to handle specifically right shifting by 1 or 32 is done.

The idea was to remove the handling of right shifts which are not by 1 or 32, as extra if branches are required for handling. Recall that ***uint96_t*** consists of three ***uint32_t*** variables, thus shifts of range 0-31, 32-63, and 64-96 will require different implementations and thus require the need for if branches.

This optimization step however only benefited when the code is compiled with -O0. This is because with -O0, the compiler does not perform in-lining for ***rshift_uint96*** and does not optimize the if branches away even if they are called with constants 1 or 32.

4.7 V4 - Register Keywords

The register keyword is added to any variables that are frequently accessed, especially if the variable is accessed in every iteration of a loop. This is to give the compiler a hint or helping hand in identifying which variables are more suited to be stored in registers.

The register keyword gave the greatest benefit when the software is compiled with no optimization flags. This is probably because the compiler is hesitant in using registers when optimization is set to level 0, and this keyword pushes the compiler to use registers. In higher levels of optimization flags, not much of an improvement is seen as the compiler is smart in deciding which variables should be in registers.

4.8 V5 - Loop Unrolling in MMM algorithm

The project goal is to work with 95-bit integers. Given that this number of bits is already known, the loop in the MMM algorithm will always be 95 iterations. Therefore, loop unrolling was attempted. The lowest divisor of 95 is 5, thus the internal body of the loop is copied and pasted to a total of 5 in the loop, the ***int i*** counter is changed appropriately to add constants 0 to 4, and the counter of the loop is incremented by 5 each loop iteration.

This however resulted in an increase in approximately 33.75% of cycle counts, and thus it was no longer further pursued. This is probably because the loop in MMM algorithm have very tight dependencies, and not much parallelization could be done even with loop unrolling. Loop unrolling by 5 times also increased the assembly size by 411 lines, and this may have resulted in more instruction cache misses.

Note that this version of optimization is not compared to the other versions in Section 5, as this version only works with RSA keys of bit size that are a factor of 5. The general test cases used by all other versions included other bit sized integers such as 96-bits and 48-bits.

5 Timing Comparisons

For measuring the cycle count and time required to compare between optimizations, the ***clock()*** function from ***time.h*** is used as a rough estimation. The optimization flag -O1 was most used for this project when performing and analyzing the optimizations steps in Section 4. This is because based on testing, the compiler seems to be hesitant in using registers past R4 with -O0. With the strongest optimization -O3, most optimizations were already performed by the compiler. Therefore, as a learning practice for this course project to better understand optimization techniques and what the compiler does, the optimization flag of -O1 is used.

Five runs of the program is performed and the average of this is compared. Each run consists of encrypting and decrypting five test cases. three of the test cases are 95-bit keys, and two other test cases are 48-bit and 96-bit keys. Table 1 below shows the testing results of each optimization version/iteration.

Version	**Optimizations Performed**	**Cycle Count Averages**		
		-O0	**-O1**	**-O3**
v0	Initial Operator Strength Reduction on multiplication	43017.8	27207.2	6116.6
v1	+ MMM Optimization	31139.2	12256.2	4931.2
v2	+ Loop Unrolling for ***uint96_t*** add and sub operations + Avoid internal array indexing in ***uint96_t***	29255.0	4022.2	5023.0
v2neon	+ Neon Addition (Not pursued / Removed for next version)	33277.4	13961.6	8141.6
v3	+ Rewriting rshift_uint96 to two functions handling only right shifts by 1 or 32	27711.8	3925.6	5251.8
v4	+ Register Keyword	21813.4	4254.2	4830.0

Table 1: Cycle Count Averages of Optimization Steps

From the results, MMM optimization that includes storing relevant 32-bit chunks of ***uint96_t*** into local 32-bit variables and avoiding using loop counter ***i*** as shift operand seems to provide positive results for all situations. Neon Addition on the other hand reduced the performance.

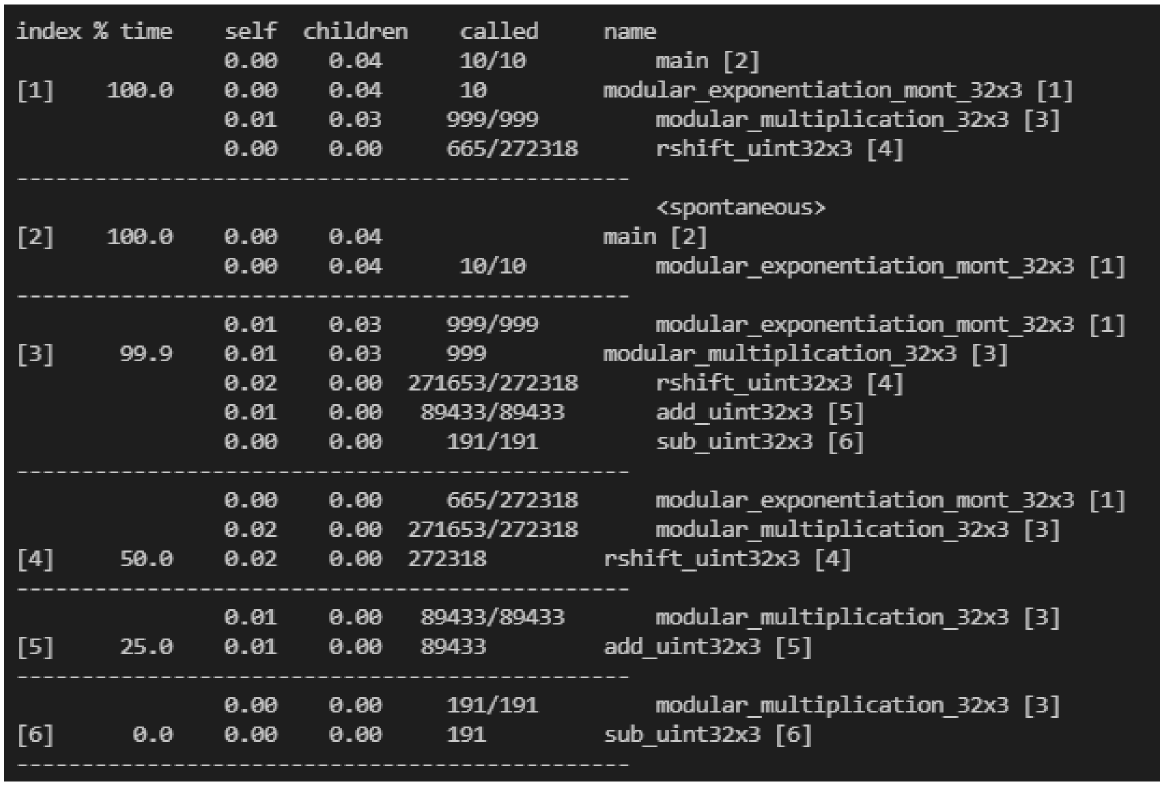
Other optimizations seem to provide improvements in some optimization levels but remain the same for others. This is expected as some optimization level may already include the optimization that was manually performed. For example, loop unrolling is optimized by the compiler using -O3, but not optimized when using -O1, therefore it makes sense that performing loop unrolling improved -O1's performance but not -O3.

An interesting observation is that after performing manual loop unrolling, -O1's performance seems to be better than -O3. An explanation for this is probably because -O3 performed some optimizations that were counterproductive whereas -O1 did not.

6 Gprof Results with -O1 Optimization Flag

Using gprof will allow us to observe which functions are being in-lined between each optimization version iterations. We want to avoid functions being called and branched, because calling functions in the assembly level leads to overhead with branching and inefficient CPU pipelining. If a function is not shown by gprof, it means that the function has been in-lined by the compiler at the assembly level. The -O1 flag is used for this comparison of manual optimization steps.

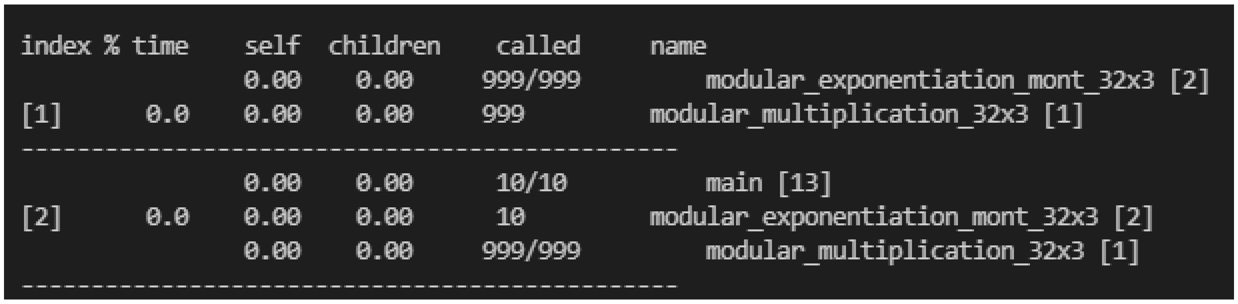
Looking at gprof Output 1 below, the initial code being compiled does not perform in-lining for ***rshift_uint96*** function calls. This is because currently the optimization that avoids passing the ***int i*** loop counter as an operand to ***rshift_uint96*** have not been performed (Section 4.2), which prevents the compiler from in-lining the function. Similarly, the initial code being compiled does not perform in-lining for ***add_uint96*** and ***sub_uint96*** as well, probably because loop unrolling optimization steps has not been performed for them (Section 4.4).



```
index % time    self  children    called     name
                0.00    0.04      10/10          main [2]
[1]    100.0    0.00    0.04      10         modular_exponentiation_mont_32x3 [1]
                0.01    0.03     999/999         modular_multiplication_32x3 [3]
                0.00    0.00     665/272318      rshift_uint32x3 [4]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    0.04                 main [2]
                0.00    0.04      10/10          modular_exponentiation_mont_32x3 [1]
-----------------------------------------------
                0.01    0.03     999/999         modular_exponentiation_mont_32x3 [1]
[3]     99.9    0.01    0.03     999         modular_multiplication_32x3 [3]
                0.02    0.00  271653/272318      rshift_uint32x3 [4]
                0.01    0.00   89433/89433       add_uint32x3 [5]
                0.00    0.00     191/191         sub_uint32x3 [6]
-----------------------------------------------
                0.00    0.00     665/272318      modular_exponentiation_mont_32x3 [1]
                0.02    0.00  271653/272318      modular_multiplication_32x3 [3]
[4]     50.0    0.02    0.00  272318         rshift_uint32x3 [4]
-----------------------------------------------
                0.01    0.00   89433/89433       modular_multiplication_32x3 [3]
[5]     25.0    0.01    0.00   89433         add_uint32x3 [5]
-----------------------------------------------
                0.00    0.00     191/191         modular_multiplication_32x3 [3]
[6]      0.0    0.00    0.00     191         sub_uint32x3 [6]
-----------------------------------------------
```

gprof Output 1: gprof Call Graph for Initial Code Version

After performing the optimizations in Section 4.2 to 4.4, gprof no longer reports the calling of these three functions, as shown in gprof Output 2 below. This is because ***rshift_uint96*** has been changed to only shift 1 or 32 in Section 4.2. Loop unrolling has been performed for ***add_uint96*** and ***sub_uint96*** in Section 4.4.



```
index % time    self  children    called     name
                0.00    0.00     999/999         modular_exponentiation_mont_32x3 [2]
[1]      0.0    0.00    0.00     999         modular_multiplication_32x3 [1]
-----------------------------------------------
                0.00    0.00      10/10          main [13]
[2]      0.0    0.00    0.00      10         modular_exponentiation_mont_32x3 [2]
                0.00    0.00     999/999         modular_multiplication_32x3 [1]
-----------------------------------------------
```

gprof Output 2: gprof Call Graph after Loop Unrolling on 96-bit Add & Sub, and avoiding 96-bit right shifts by loop counter i

7 Hardware Solution Optimization

Given that the project targets 95-bit keys, the most time-consuming processes are 96-bit operations. A large part of RSA encryption and decryption is performing 96-bit right shifts and 96-bit summation. On the other hand, the 96-bit subtraction operation is not used very often. This is known by studying the algorithm or observing at gprof's Output 1 from above. We can see that the ***rshift_uint96*** took 50% of total time and ***add_uint96*** took 25% of total time, while ***sub_uint96*** took 0%. Therefore, it is worthwhile to consider hardware-assisted solutions for these 2 operations.

Looking into the 96-bit right shift operations, given that we have already optimized MMM such that only right shifts by 1 and 32 is required, the instructions for both ***rshift1_uint96*** and ***rshift32_uint96*** are very simple and should not take more than 5-8 cycles. Given that the overhead of a hardware-assisted solution might be lower than 5-8 instructions, it is still not beneficial to consider a hardware-assisted solution for the 96-bit right shift operations, considering a hardware-assisted solution is limited to only two 32-bit operands and a 32-bit output and would require multiple hardware.

For 96-bit addition operations, the function is more complicated is estimated to have approximately 18 instructions. A hardware assisted solution may be useful assuming three hardware can be running in parallel, and the overhead of each hardware call is assumed to be around 5 instructions, but this will be difficult to implement given that there are tight dependencies among each instruction for the current implementation of 96-bit addition.

A further step can be done to look at larger operations that uses these smaller 96-bit operations but given the current code implementation and the tight dependencies in the MMM algorithm, it was extremely difficult to find larger chunks of code that could be put into functions, and even more so for functions with specifically two 32-bit operands and output one 32-bit result.

8 Conclusion

This project showed that it is important to understand and apply software optimizations techniques manually and not heavily rely on compiler's optimization. Although compilers today optimize very efficiently, manual software optimization has improved performance for even the strongest level of safe optimization that the gcc compiler can offer. It is also important to test the code's runtime after each optimization step because some optimization techniques seem beneficial in theory, but it is difficult to predict cache misses or how the compiler will compile the code, and these might result in a performance loss.

9 References

[1] M. Sima, Class Lecture, Topic: "Lesson 108: RSA Cryptography," University of Victoria, May. 2021.