SENG 440 Project Report

Embedded System Implementation of RSA Cryptography

July 30th 2014

Simon Diemert, V00723337
Scott Low, V00725620
Geoff Gollmer, V00730142

1.0 Introduction

1.1 Document Purpose and Scope

The purpose of this report is to present an embedded system implementation of the standard RSA encryption and decryption algorithm. The process of moving the implementation from purely software, through to firmware and finally to hardware is described. This document is not meant to be an exhaustive review on all possible implementations of RSA and only describes the lookup table, modular exponentiation and Montgomery modular exponentiation approaches in a suitable amount of mathematical detail.

1.2 Background

RSA encryption provides a means of encrypting data using a set of two encryption keys (public and private) which is for all intents and purposes is unbreakable, based on our current mathematical and computational abilities [1]. In practice, RSA is widely used to secure communications between two hosts on a network. A review of the details of RSA is given in Section 2.0 of this report.

Due to the nature of the mathematics involved in RSA, encryption/decryption of data is computationally expensive. In an effort to produce faster methods of encryption/decryption, several algorithms have been developed in attempt to reduce the number of clock cycles required complete one encryption/decryption operation. Techniques that have been developed include Modular Exponentiation [2] and Montgomery Multiplication [2], both of which are described in section 2.0. However, these techniques are algorithms that are usually implemented in software. To improve the efficiency, moving portions of these algorithms into hardware will help reduce the number clock cycles required to encrypt/decrypt a message.

1.3 Project Objectives

The major conceptual objective of this project is to demonstrate that embedded system operations can be optimized by moving the implementation closer to hardware, or in some cases directly into hardware. As a result, this project has five objectives:

1. Implement various RSA encryption/decryption algorithms in the C programming language.
2. Optimize the most efficient C algorithm using the software optimization techniques presented in class.
3. Optimize ARM assembly instructions produced by ARM-GCC compiler.
4. Determine major the bottlenecks in Software/Firmware implementation.
5. Implement slower firmware operation(s) in hardware using VHDL.

1.4 Approaches

As previously mentioned, there are several approaches to reducing the strength (number of clock cycles required) for encryption and decryption, this section describes five approaches that were considered. For the purposes of this project only the fifth, the combination of Modular Exponentiation and Montgomery Multiplication, was fully implemented, the remainder of this report will detail this implementation.

1.4.1 Stock RSA

The standard or ‘stock’ RSA approach uses both the modulus and multiplication operations quite heavily and relies on large numbers to ensure communications are secure (see section 2.0). As a result, this is not feasible to implement on an embedded processor with small amount of computation resources. This stock algorithm was implemented using the Python programming language, which has constructs for handling large integers, to demonstrate its viability but was not further pursued as it is not feasible for an embedded system.

1.4.2 Look Up Table

The look up table approach uses a pre-computed table of values used in the encryption and decryption process. This reduces the cost of the encryption and decryption by requiring some of the more expensive operations to only be computed once, rather than repeatedly. Unfortunately, this approach requires a table to be computed for every possible combination of encryption key and input to encryption/decryption operation. Thus the strength of the RSA operations is only reduced if the look up table is computed ahead of time. For this project the look up table method was implemented in the C programming language and then not pursued further.

1.4.3 Modular Exponentiation

The modular exponentiation approach improves upon the look up table method by doing away with the requirement to compute a table ahead of time. As a result, the value $c \equiv b^e \ (mod\ m)$ is computed iteratively by applying the modular exponentiation algorithm (see section 1.2.2.3) and repeatedly taking the modulus of the resulting product to ensure that the value of b^e does not exceed the modulus. There is another approach where the value of b^e is computed entirely before the modulus is taken, but the risk of this approach is that the resulting exponentiation can overflow the data type being used. For the purpose of this project, the modular exponentiation method was implemented in the C programming language and was not pursued further due to the observable optimizations present in the Montgomery Multiplication approach.

1.4.4 Montgomery Multiplication

The Montgomery multiplication approach uses an augmented version of modular exponentiation to compute $c \equiv b^e \ (mod\ m)$. In this approach, however, Montgomery Multiplication (see section 2.3.2.3) is used in place of regular multiplication to compute the result of any mathematical operation of the form (A * B) mod m. The main speedup in this approach comes from the fact that any complex mathematical operators (such as modulus and division) are all performed using powers of two. As a result, any modulo or division operations can be reduced to bit-wise ands or right shifts respectively.

For the purpose of this project, a Montgomery Multiplier and a modular exponentiation algorithm using the Montgomery multiplier were implemented in the C programming language. Since this was the most efficiently of the three approaches tried in this project, the algorithm was optimized as much as possible before being compiled to ARM assembly code. This assembly code was then further optimized and its execution time was measured. Finally, the C code was translated to VHDL and a logic circuit to emulate Montgomery multiplication in hardware was created and analyzed to determine the speedup achieved.

2.0 Theoretical Foundation

1.2.1 RSA Encryption

1.2.1.1 Purpose of RSA

RSA is a public-key cryptosystem that is widely used for secure data transmission. First described in 1977 by Leonard Adleman, Ron Rivest and Adi Shamir, RSA (Rivest, Shamir, Adleman) uses the product of two large prime numbers as a modulo when creating both public and private encryption/decryption keys. As a result, its security stems from the fact that the factorization of a product of two large large prime numbers remains a problem in the FNP complexity class.

1.2.1.2 Basic Concepts

RSA involves both a private key (kept secret and used for decryption) and a public key (known by everyone and used for encryption). The keys are generated via the following process:

- Pick two unique, large prime numbers. Call them p and q.
- Compute the integer n = pq. This will be used as the modulus for both the public and private keys.
- Euler's totient function is used to calculate $\Phi(n) = n - (p + q - 1)$
- An integer e is then chosen such that $1 < e < \Phi(n)$ and e and $\Phi(n)$ are coprime (in other words, $\gcd(e, \Phi(n)) = 1$). This is the public key exponent.
- Determine d as $d \equiv e^{-1} \pmod{\Phi(n)}$. This is the private key exponent and should be kept secret.

The public key for RSA is given by the tuple (n, e). To encrypt a message, m, to ciphertext, c, a public key, (n, e), and the formula $c \equiv m^e\ (mod\ n)$ are used. Then, to decrypt, the formula $m \equiv c^d\ (mod\ n)$ is used where d is the private key exponent as outlined above.

1.2.2 Modular Exponentiation

1.2.2.1 Purpose

As mentioned in the previous section, one of the main computational operations that makes up the encryption and decryption process of the RSA algorithm is modular exponentiation. This occurs when some message, m, is converted to ciphertext, c, by using the public key, e, and the specially selected integer n using the formula $c \equiv m^e\ (mod\ n)$. Likewise, when decrypting, the previous formula becomes $m \equiv c^d\ (mod\ n)$ where d is the receiver's private key. For large messages, many encryption and decryption cycles could occur. As a result, it is important for this process to be as streamlined as possible.

1.2.2.2 Mathematical Foundation

The modular exponentiation approach uses two fundamental mathematical identities to compute $c \equiv b^e\ (mod\ m)$. These are namely the fact that $b^{2n} = (b^n)^2$ and $b^{2n+1} = b^{2n} * b$. As a result, the resulting product, b^e, can be computed by continuously dividing the exponent, e, by 2, and applying the aforementioned identities to a running product that keeps track of the overall result. After each time the exponent is divided by two, the modulus by m is applied. Although more expensive than the alternative (which is to compute the entire product b^e before taking the modulo at the end), this ensures that the product never overflows the data type being used.

1.2.2.3 Algorithm

Pseudocode for the modular exponentiation algorithm is as follows. Assume that the algorithm is trying to determine c, where $c \equiv b^e \ (mod\ m)$

```
var result = $b^0$ = 1

while e is not 0, do:
    if e is not evenly divisible by 2:
      result = (result * base) mod m.
      //This is equivalent to applying the identify $b^{2n+1}$=$b^{2n}$*b.
      base = ($base^2$) mod m. This is equivalent to applying the
identify $b^{2n}$ = $(b^n)^2$
  divide exp by 2 (using integer division)
return result
```

1.2.3 Montgomery Multiplication

1.2.3.1 Purpose

As mentioned in section 1.2.2.1, modular exponentiation is a crucial factor which contributes to the execution time and efficiency of the RSA encryption algorithm. The mod m operation that occurs at each step in the modular exponentiation algorithm (as shown in section 1.2.2.3), however, is particularly slow since modulo is an expensive operation to perform in software. As a result, Montgomery multiplication defines a faster way to calculate A*B mod m that does away with expensive modulo operations in favour of more efficient bitwise ands and right shifts. As a result, Montgomery multiplication allows the two identities ($b^{2n} = (b^n)^2$ and $b^{2n+1} = b^{2n} * b$) to be computed in a more efficient manner, thus reducing the overall execution of the modular exponentiation algorithm.

1.2.3.2 Mathematical Foundation

An normal optimized modular multiplier will examine the most significant bit of the running project and subtract the modulo when the product outgrows the size of the modulo. This is fundamentally different from a Montgomery modular multiplier, which will examine the least significant bit and if overflow is going to occur, add the modulo to the product (which does nothing to change the congruency of the product) before right shifting it each iteration. In effect, this eliminates the requirement for long carry chains to determine whether or not a readjustment for the integer range of the modulo is necessary.

Due to the right shifts performed, there is an inherent scale factor of 2^{-n} present in the running product. To eliminate this, Montgomery defined *m-residue* format, a conversion factor that defines an input A as Ar = A * 2^n mod M. Converting both the multiplier and the multiplicand to *m-residue* format is necessary, but also creates an extra scale factor of 2^n to be present in the *m-residue* version of the product, S, since it would be defined as S = Ar * Br * 2^n mod M. A multiplication of S in a Montgomery multiplier by the integer 1 (not in its *m-residue* representation) will reintroduce the 2^{-n} factor, cancelling out the extra factor and producing the correct, unscaled result.

As mentioned previously, a Montgomery multiplier can be used in the modular exponentiation algorithm given in section 1.2.2.3 above to reduce the operator strength of the algorithm, and as a result, reduce its execution time.

1.2.3.3 Algorithm

The bitwise pseudocode for a Montgomery multiplier has been given below (taken directly from Lesson 103: RSA Cryptography class slides) [3]

```
T = 0
for i = 0 to m - 1 do
    n = T + X(i)Y(0)
    T = (T + X(i)Y + nM)/2
end for
if T >= M then
    T = T - M
end if
return T
```

Assuming that the above pseudocode defines a function called Montprod, then a modular exponentiation algorithm using a Montgomery multiplier can be defined as follows [2]:

```
MontExp(X, E, M)
    Nr = $2^{2n}$ mod M
    $Z_0$ = Montprod(1, Nr, M)
    $P_0$ = Montprod(X, Nr, M)
    for i = 0 to n-1 do
        $P_{i+1}$ = Montprod($P_i$, $P_i$, M)
        if($e_i$ = 1) then
            $Z_{i+1}$ = Montprod($Z_i$, $P_i$, M)
        else
            $Z_{i+1}$ = $Z_i$
        end if
    end for
    $Z_n$ = Montprod(1, $Z_n$, M)
    return Z
```

Note that Nr has been defined as 2^{2n} where n is the number of bits in each of the inputs to the MontExp function due to the fact that number is guaranteed to be both greater than and relatively prime to the modulo M. Furthermore, since Nr is a power of two, division and remainder operations become shifts and bitwise masks respectively (as shown above).

To enhance readability, an activity diagram of the Montgomery multiplication algorithm has been included in Figure 1 below.

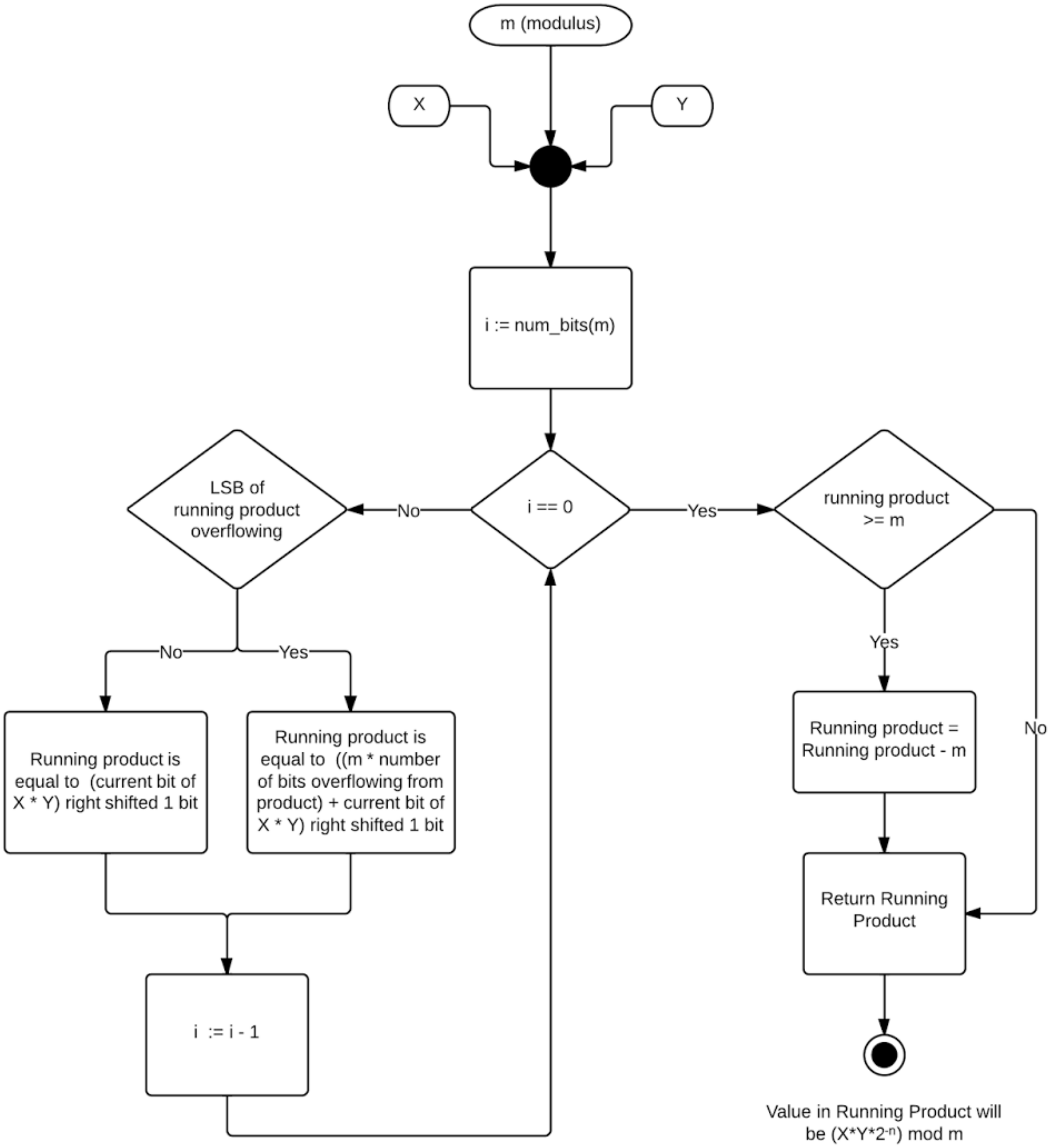

m (modulus)
X
Y
i := num_bits(m)
i == 0
No
Yes
LSB of running product overflowing
No
Yes
Running product is equal to (current bit of X * Y) right shifted 1 bit
Running product is equal to ((m * number of bits overflowing from product) + current bit of X * Y) right shifted 1 bit
i := i - 1
running product >= m
Yes
No
Running product = Running product - m
Return Running Product
Value in Running Product will be (X*Y*2-n) mod m

Figure 1: Montgomery multiplication activity diagram

3.0 Software Implementation

The following section discusses the various RSA-encryption/decryption algorithms that were implemented in the C programming language as part of this project.

3.1 Look Up Table

The code for the implementation of the look up table RSA encryption/decryption algorithm can be found in section A.1 of Appendix A below. Due to the fact that a lookup table must be computed before any actual encryption/decryption can begin, this algorithm is much slower than others explored in this project. Furthermore, the algorithm requires several function calls, which in turn increases the number of branches required to perform encryption/decryption. As a result of these observations, it was decided that the look up table implementation would not be optimized further, as even a more optimized version would be slower and require more memory than other alternatives.

3.2 Modular Exponentiation

The code for the modular exponentiation approach for RSA encryption/decryption can be found in section A.2 of Appendix A below. This algorithm has a lower execution time than the lookup table approach due to the fact that it is able to compute a modular product dynamically rather than requiring a precomputed table. This fact also means that the algorithm overall requires less memory to operate. A key issue with this implementation of modular exponentiation, however, is the `% divisor` operation that occurs twice in the algorithm. Since there is no guarantee that divisor will be a power of two, there is no way to perform operator strength reduction and use bitwise masks in place of the modulo operation. Since modulo is typically performed via repeated subtraction, the bottleneck in this algorithm will occur at this point.

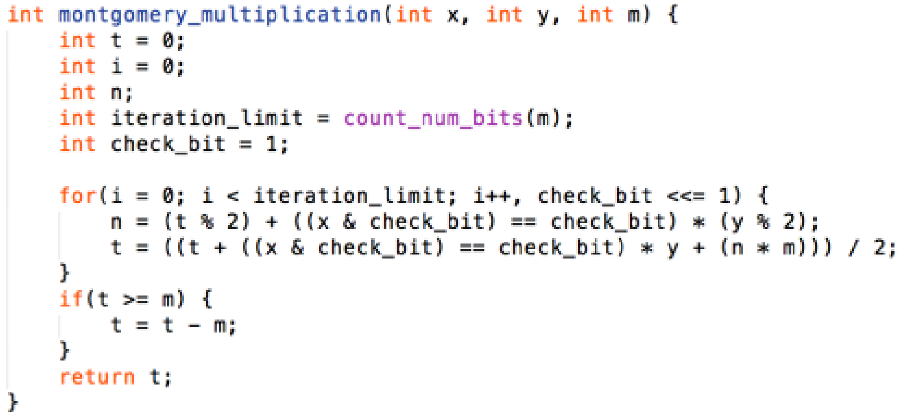
Although the version of the algorithm in section A.2 of Appendix A could be optimized, it was decided to leave the code unoptimized in favour of exploring and optimizing the Montgomery modular exponentiation algorithm since its implementation avoids the use of the complex modulo operations present in modular exponentiation.

3.3 Montgomery Modular Exponentiation

The unoptimized code for the Montgomery modular exponentiation approach for RSA encryption/decryption can be found in section A.3 of Appendix A below. As mentioned in the previous section, this algorithm avoids complex operators such as modulo and division by ensuring that the modulus and divisor are always powers of two. As a result, these operations can be converted into bitwise masks and right shifts accordingly. Due to these speedups, it was decided to further explore and optimize this algorithm using the techniques learned in class.

The optimized version of the algorithm can be found in section A.4 of Appendix A. Several techniques, including operator strength reduction, initialization/exit condition optimization of loops, local declaration of static variables and suggestion of registers have all been performed.

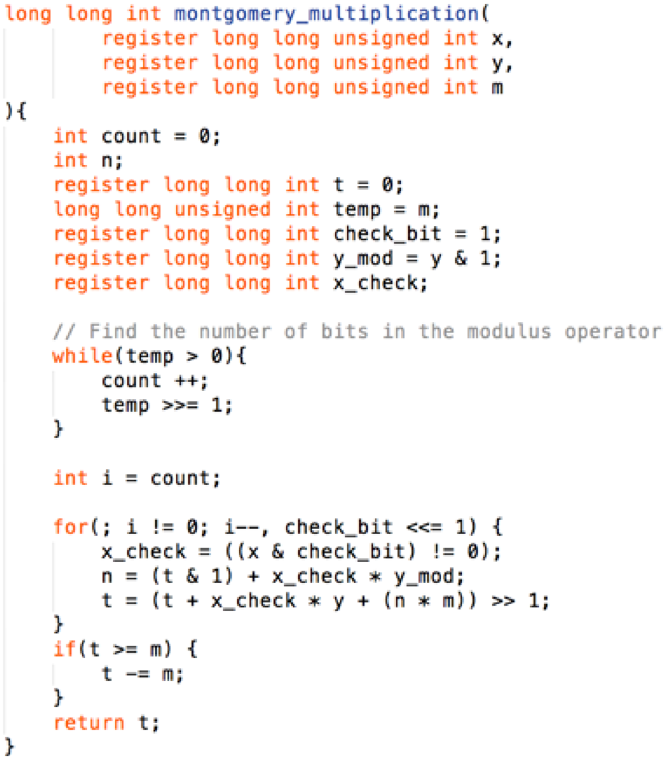
For example, consider the differences between the Montgomery multiplication algorithms shows in Figure 1 and Figure 2 below. The loop initialization/exit condition optimization has been performed by initializing i outside of the loop and decrementing it and comparing it with zero each time the loop iterates. This is much more efficient than incrementing and performing a less than comparison due to the fact that an equality test with zero can be performed in a single clock cycle in ARM. Furthermore, operator strength reduction has been performed by converting all modulo and division operations to bitwise masks and shifts. (Note that n % m is equivalent to n & (m - 1) when m is a power of two). Finally, some extra calculations, such as y % 2, t % 2 and x & check_bit were computed more than once per each iteration of the loop. As a result, these operations have all been stored in local variables so that they only have to be computed once per loop iteration.



```
int montgomery_multiplication(int x, int y, int m) {
    int t = 0;
    int i = 0;
    int n;
    int iteration_limit = count_num_bits(m);
    int check_bit = 1;

    for(i = 0; i < iteration_limit; i++, check_bit <<= 1) {
        n = (t % 2) + ((x & check_bit) == check_bit) * (y % 2);
        t = ((t + ((x & check_bit) == check_bit) * y + (n * m))) / 2;
    }
    if(t >= m) {
        t = t - m;
    }
    return t;
}
```

Figure 2: Montgomery Multiplication Unoptimized



```
long long int montgomery_multiplication(
        register long long unsigned int x,
        register long long unsigned int y,
        register long long unsigned int m
){
    int count = 0;
    int n;
    register long long int t = 0;
    long long unsigned int temp = m;
    register long long int check_bit = 1;
    register long long int y_mod = y & 1;
    register long long int x_check;

    // Find the number of bits in the modulus operator
    while(temp > 0){
        count ++;
        temp >>= 1;
    }

    int i = count;

    for(; i != 0; i--, check_bit <<= 1) {
        x_check = ((x & check_bit) != 0);
        n = (t & 1) + x_check * y_mod;
        t = (t + x_check * y + (n * m)) >> 1;
    }
    if(t >= m) {
        t -= m;
    }
    return t;
}
```

Figure 3: Montgomery Multiplication Optimized

By performing these optimizations, the number of ARM assembly instructions required to implement the algorithm dropped from 198 to 123, approximately 37% reduction.

4.0 Assembly Optimization

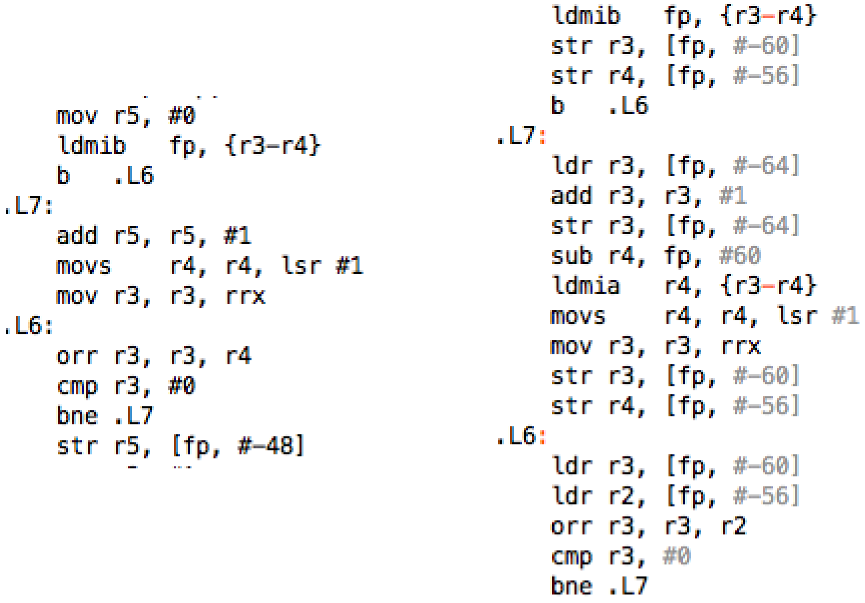
The assembly code was produced by compiling the optimized C algorithm (presented in Section 3.0) and outputting the assembly intermediate step via the following command:

```
arm-linux-gcc -static  montgomery_asm.s -o montgomery.c
```

The Montgomery Multiplication function within the optimized C code was the target for further optimization in assembly. The following types of inefficiencies were the primary targets for optimization:

1. Redundant or extra move operations which have no effect
2. Extra load operations for values that could easily be kept in registers between branch instructions
3. Extra load instructions that occur because only 2-3 registers are commonly used and must be swapped out
4. Use of only 2 registers for operations, instead of spreading the values out into multiple registers
5. Redundant initialization steps

The inefficiencies listed above are all a product of the gcc compiler attempting to ensure the operations are carried out safely in the correct order of execution for a general piece of C code. During manual optimization, the programmer can make changes based on assumptions related to the C code that a general purpose compiler cannot. An example illustrating the type of optimization techniques used can be seen in Figure 3 below.



```
    mov r5, #0
    ldmib   fp, {r3-r4}
    b   .L6
.L7:
    add r5, r5, #1
    movs    r4, r4, lsr #1
    mov r3, r3, rrx
.L6:
    orr r3, r3, r4
    cmp r3, #0
    bne .L7
    str r5, [fp, #-48]
```

```
    ldmib   fp, {r3-r4}
    str r3, [fp, #-60]
    str r4, [fp, #-56]
    b   .L6
.L7:
    ldr r3, [fp, #-64]
    add r3, r3, #1
    str r3, [fp, #-64]
    sub r4, fp, #60
    ldmia   r4, {r3-r4}
    movs    r4, r4, lsr #1
    mov r3, r3, rrx
    str r3, [fp, #-60]
    str r4, [fp, #-56]
.L6:
    ldr r3, [fp, #-60]
    ldr r2, [fp, #-56]
    orr r3, r3, r2
    cmp r3, #0
    bne .L7
```

Figure 4: Example of assembly optimization. Optimized code on the left, unoptimized version on the right.

The result of optimizing the Montgomery multiplication was approximately a 26% (125 instructions down to 92) increase in efficiency. This was measured based strictly on the number of assembly instructions that were removed from the original assembly code produced by the compiler. It represents the ‘best case’ input where each loop construct would only execute once, this would be caused by set of inputs to the Montgomery function where the modulus is 1.

Further analysis of the code, if executed with the ‘worst case’ inputs, showed that the increased in efficiency was approximately 24% which coincides with a difference of 1166 instructions being executed between the optimized and unoptimized code. The ‘worst case’ was considered to be a set of inputs that would use the most number instructions to accomplish the task, specifically a 64 bit modulus operand which would cause the most number of loop iterations. The increase in efficiency was computed by taking the number of instructions in sections of the code that would be repeated and multiplying the number loop iterations over those sections of code for both the optimized and unoptimized assembly programs.

5.0 Hardware Implementation

The following section outlines the strategy followed for implementing the presented algorithm in hardware, and the resulting statistics for the generated implementation.

5.1 Strategy

In order to speed up the algorithm, a hardware based implementation of Montgomery multiplication was created. The hardware implementation was based on the optimized C implementation and created using VHDL within Xilinx ISE software. The final implementation of hardware written in VHDL can be found in Appendix C. By implementing this component of the operation in hardware, the runtime of the Montgomery multiplication function can be reduced to a maximum of 43 clock cycles. However, the chip is able to process certain numbers faster than 43 cycles and emits a signal when the correct value has been calculated. The processor could potentially use this signal to detect when the function is completed and return control to the generated ARM code. Optionally, the ARM code can wait in an infinite loop until the signal is emitted.

The hardware implementation accepts the same arguments as the C function, in the form of three 64 bit busses. Those values are loaded into registers within the unit, so that the unit can continue processing if the input signals are changed. A single bit control signal informs the hardware unit to begin processing values, and the complete control signal is set off once the loop is completed. The multiplier is loaded into a working register, and processing begins.

The loop evident in the optimized code is simulated using a simple clock input. On the high edge of the clock, the output of the most recent loop is copied back into the register working register. In this way processing can occur for as many cycles as necessary. Each time the output is copied into the loop a copy of the modulus is down shifted, when the copy reaches a value of 0 the process is completed, and the result is loaded into an output register. Once the output has been loaded, the hardware component emits a signal indicating that it is ready for new input.

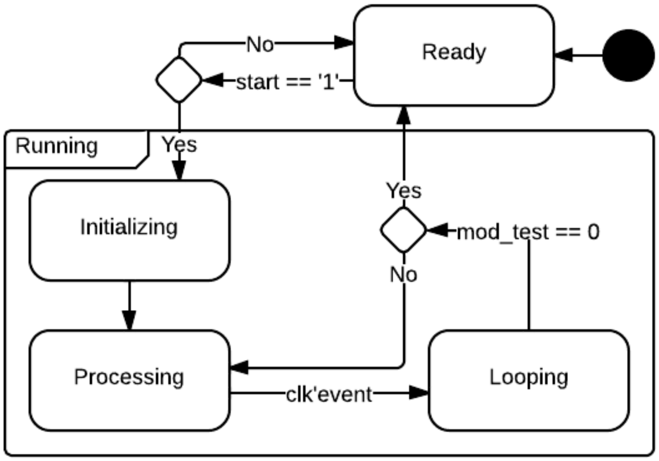


Figure 5: VHDL State Diagram

In order to execute the hardware from an ARM based processor, a number of implementations could be used. Instructions to set the modulus could be separated so that executing the hardware simply requires a set Montgomery modulo operation and a Montgomery multiplication operation that takes the multiplier and multiplicand as arguments. Otherwise, a preset relationship of register to arguments could exist, requiring each argument to be placed in a specific register before executing the hardware. This implementation would be faster given algorithms were designed around using it, but could be difficult to use correctly. Either option would still result in a significant speedup, as the hardware takes a less than two clock cycles per loop in hardware.

5.2 Optimizations

In hardware a number of simple optimizations are available that cannot be implemented in c. First and foremost is operator reduction. The variable n in C is actually always a value between 0 and 2. In c code this variable is calculated using a multiplication operation and an addition operation. In the VDHL implementation n was reduced to a 2 digit bus, and is calculated using 2 gates, an XOR gate and an AND gate. Which is a significant speedup. Additionally, many values do not need to be stored in registers in the VHDL implementation but can instead be available via busses for the duration of each loop.

5.3 Resulting Hardware

The following table documents the components required to assemble the generated hardware.

Hardware Component	Count
64-bit Adder	2
64-bit Subtractor	1
1-bit register	1
64-bit register	6 (384 1 bit registers)
64-bit comparator (Less than or equal)	1
1 bit 2-1 multiplexer	66 (1 64 bit split)
64-bit 2-1 multiplexer	5
1 bit xor	1

Table 1: Hardware implementation component breakdown.

The generated hardware was estimated to have a minimum clock period of 5.306ns in order to produce correct values.

5.4 Final Timing Estimations.

In order to estimate the optimization potential of a hardware chip, the clock speed of a 32 bit adder was calculated to provide an idea of timing for an ARM clock cycle. A 32 bit adder was calculated to have a clock speed of 3ns while our hardware implementation takes approximately 5.3ns per loop, with a maximum runtime of ~340ns or 114 ARM clock cycles compared to 3619 clock cycles in ARM, producing a 3175% speedup. With a 32 bit modulus the system runs in 57 ARM clock cycles (worst case).

6.0 Discussion

In order to measure the speedup of each method, CPU runtime was measured in terms of ARM instructions which we estimated as one clock cycles each. This measurement assumes that the number of instructions produced by a compiled C program directly relates to its runtime. However, this could be inaccurate, as a loop can often reduce the size of the compiled ARM code, while increasing the runtime of a program. It also failed to account for caching or memory misses

Measuring using clock cycles is a little more accurate, however, potential memory misses and instructions that can take more than one clock cycle are not considered. By measuring using clock cycles the best case scenario is always considered, but in actual runtime, this case will almost never occur.

Additionally, when measuring hardware, only the predicted execution time of a chip can be measured and used for conversions to clock cycles or other useful measurements. In order to determine the length of an ARM clock cycle, a 32 bit adder was constructed and tested in VHDL. This adder's runtime is the basis of all timing measurements in VHDL. If the generated adder differs from the add operation implemented in the ARM simulator, the measured optimization could be incorrect. Without proper timing details on the ARM implementation, and proper testing on actual hardware, the true speedup of a hardware implementation can only be based on the aforementioned estimations. Additionally, more hardware speedups could potentially exist on top of the generated chipset from our VHDL. Some registers, gates, or other chipsets may be unnecessary, allowing for reduced execution time.

7.0 Conclusion

In conclusion, an efficient embedded implementation of RSA was developed. Montgomery multiplication was chosen as the bottleneck in the execution and was optimized. This process involved optimizing C code using software optimization techniques before removing redundant instructions in assembly and finally moving the routine into hardware. Each step in the optimization reduced the complexity of the Montgomery multiplication routine.

For example, optimizing the original Montgomery multiplication routine in C using the software optimization techniques presented in class reduced the number of ARM instructions by approximately 37%. This optimized algorithm was then converted to ARM assembly and further optimized to achieve an increase in efficiency of approximately 24%. Finally, a VHDL implementation of Montgomery multiplication increased the efficiency of the optimized ARM assembly by approximately 3175%. In total, a speedup of 81.76 times was experienced between the unoptimized Montgomery multiplication algorithm in C and the VHDL implementation.

8.0 References

[1] R.L. Rivest, A. Shamir, L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. Cambridge, MA: MIT, 1978.

[2] J. Fry, M. Langhammer, “RSA & Public Key Cryptography in FPGAs”. San Jose, CA: Altera, 2005.

[3] M. Sima, “Lesson 103: RSA Cryptography”. Victoria, B.C., Canada: University of Victoria, 2013, p. 10095.