RSA Algorithm in x86-64

Christopher Arredondo-Fallas
chrisarrefall@gmail.com,
Computer Architecture - Computer Engineering
Costa Rica Institute of technology

***Abstract*—In the following document the implementation of the RSA algorithm using x86-64 assembly will be discussed. You will find information on the various algorithms that were used to manipulate multi-register arithmetic or arbitrary-precision arithmetic, we will discuss the implementation of the $addition$, $subtraction$, $multiplication$, $division$ and how based on those algorithms, we calculated RSA public and private keys. Further more we will be analyzing performance and comparing division algorithms.**

***Keywords*—RSA, keys, encryption, arithmetic, algorithm, register, euclidean.**

I. Introduction

RSA(Rivest–Shamir–Adleman) is an algorithm used by modern computers to encrypt and decrypt messages. It is an asymmetric cryptographic algorithm. Asymmetric means that there are two different keys. This is also called public key cryptography, because one of the keys can be given to anyone. The other key must be kept private. This algorithm is based on special operations done with two prime numbers p and q. For this project to be able to use more than 64 bits, we uses arbitrary-precision arithmetic. Arbitrary-precision arithmetic in computer science is dividing numbers in various registers and operating on them. In this project we implemented the 4 basic arithmetic operations($+$, $-$, $\times$, $\div$) with arbitrary-precision.

II. Arbitrary-precision algorithms

A. Addition

For addition, a simple manual school algorithm is used in which you carry every time a sum is greater than 9. An example would be:

$$123456 + 78$$

- initial carry = 0
- 56 + 78 + 0 carry = 134 = 34 with 1 carry
- 34 + 00 + 1 carry = 35 = 35 with 0 carry
- 12 + 00 + 0 carry = 12 = 12 with 0 carry

In x86-64 it would look like:

```
_sum:
    mov rax, [b]
    add [a], rax
    mov rax, [b+8]
    adc [a+8], rax
    mov rax, [b+16]
    adc [a+16], rax
    mov rax, [b+24]
    adc [a+24], rax
    ret
```

Fig. 1. In this code, we move the data in b in to the 64 bit register rax, after that we operate using sub, followed by that we repeat using the next 64 bit space of memory but instead of using sub we use a special command called sbb that subtracts but with the borrow that will affect the next operation.

B. Subtraction

For subtraction, we use a similar algorithm to addition but instead of carry we look for a borrow. An example would be:

```
_sub:
    mov rax, [b]
    sub [a], rbx
    mov rax, [b+8]
    sbb [a+8], rcx
    mov rax, [b+16]
    sbb [a+16], rcx
    mov rax, [b+24]
    sbb [a+24], rcx
    ret
```

Fig. 2. In this code, we move the data in b in to the 64 bit register rax, after that we operate using add, followed by that we repeat using the next 64 bit space of memory but instead of using add we use a special command called adc that adds but with the carry generated from the previous operation.

C. Multiplication

For multiplication, the algorithm changes a bit, and can me summarized in the following figure.

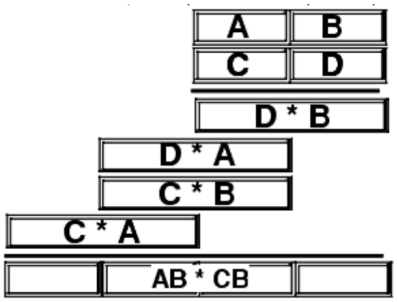

Fig. 3. Multiplication algorithm.

In x86-64 it would look like:

```
_mul:
    mov rax, [a]
    mov rbx, [b]
    mul rbx
    mov [c], rax
    mov rcx, rdx

    mov rax, [a+8]
    mul rbx
    add rax, rcx
    adc rdx, 0
    mov [c+8], rax
    mov rcx, rdx
          .
          .
          .
```

Fig. 4. In this code, we move the data in a in to the 64 bit register rax and the data in b in to the 64 bit register rbx, after that we operate using mul, followed by that we save the value in c not forgetting to save the overflow in rcx(overflow is saved by default in rdx). We repeat the same process using the next memory segment of b and finally the next memory segment of a.

D. Division

For the division algorithm, is got trickier. There are two implementations of division in the code, one using the instruction div and the other one based on shifts and subtractions. Later on we will be discussing a comparison of performance between both algorithms. The first algorithm mentioned before is using a 64 bit denominator and a 256 bit numerator, and is implemented in the following way:

```
_div64:
    mov rbx, [b]

    mov rdx, 0
    mov rax, [a+24]
    div rbx
    mov [c+24], rax

    mov rax, [a+16]
    div rbx
    mov [c+16], rax

    mov rax, [a+8]
    div rbx
    mov [c+8], rax

    mov rax, [a]
    div rbx
    mov [c], rax

    mov [r], rdx

    ret
```

Fig. 5. In this algorithm, we basically do a manual long division, in which we start by dividing the first segment of the 256 bit number by the 64 bit denominator, we then move to the next segment until we get to the last part of the numerator. Finally the remainder is saved in the rdx register

The other algorithm is a lot bigger but is based of the following pseudo-code:

```
Quotient := Dividend;

Remainder := 0;

for i:= 1 to NumberBits do

    Remainder:Quotient := Remainder:Quotient SHL 1;

    if Remainder >= Divisor then
        Remainder := Remainder - Divisor;
        Quotient := Quotient + 1;
    endif

endfor
```

Fig. 6. This algorithm is fairly simple, but compared to the normal algorithm implemented previously, it is more complex and more inefficient. Although it has a finite number of iterations, it being the amount of bits we are using, in our case it had 256 iterations because we were using 256 bits for the numerator and denominator.

III. RSA SUBROUTINES

In RSA we have two sides of the story, calculating the private key and the public key.

A. Private Key

For the private key we will need to calculate a variable called d.

1) Calculating d: To calculate d we will need to first calculate e and phi for e we assumed $e = 65537$ the reason being that it is one of the most common keys used. And for phi we did the following operation:

$$phi = (p-1) * (q-1)$$

Finally to calculate d we did the following operation:

$$d = e^{-1} \ mod \ phi$$

As you can see calculating e^{-1} isn't a trivial operation for a computer, the solution to this problem is using an algorithm called the $extended\ euclidean\ algorithm$. The extended Euclidean algorithm is an extension to the Euclidean algorithm, and computes, in addition to the greatest common divisor of integers a and b, also the coefficients of Bézout's identity, which are integers x and y such that

$$ax + by = \gcd(a, b)$$

And to calculate $\gcd$ we used the following pseudo-code:

```
function gcd(a, b)
    while a ≠ b
        if a > b
            a := a − b;
        else
            b := b − a;
    return a;
```

Fig. 7. gcd algorithm pseudo-code.

Finally, based on the $extended\ euclidean\ algorithm$ we are able to calculate d by doing the following operation:

$$e \cdot x + phi \cdot y = \gcd(e, phi)$$

B. Public Key

For the public key, we will only have to calculate one variable, that would be n and in the case of e we assumed

$$e = 65537$$

. To calculate n it is as simple as doing:

$$n = p \cdot q$$

C. Encryption

After calculating every variable, we will proceed to calculate the encrypted text product of this algorithm. Taking m as a message given by the user and c the ciphered result it would be calculated with the following operation:

$$c = m^e \ mod \ n$$

As you can see we have an operation with the form of $x^y \ mod \ z$, again this is sort of complicated for a computer, because of the size x^y could get to. The solution to this problem was to implement the $modular\ exponentiation$ algorithm:

```
def power(x, y, z) :
    res = 1
    while (y > 0):

        yand = (y & 1)
        if (yand == 1) :
            resTx = (res * x)
            res = resTx % z
        y = y >> 1
        xTx = (x * x)
        x = xTx % z

    return res
```

Fig. 8. In the $modular\ exponentiation$ algorithm we basically start shifting y to the right, dividing the number in 2 until it reaches zero.

D. Decryption

For decryption, we use the same algorithms as encryption, the reason being that to obtain the message again we do the following operation:

$$m = c^d \ mod \ n$$

Because it has the same form as the encryption operation we are able re reuse the same algorithm to obtain the final message.

IV. Performance analysis

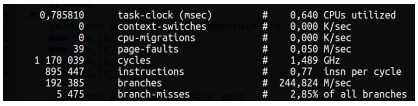

Fig. 9. Results using $perf$ with c calculation.

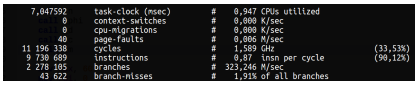

Fig. 10. Results using $perf$ with m calculation.

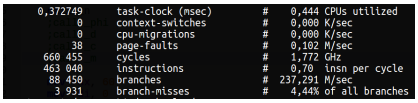

Fig. 11. Results using $perf$ with multiplication.

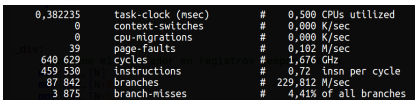

Fig. 12. Results using $perf$ with 64 bit denominator division.

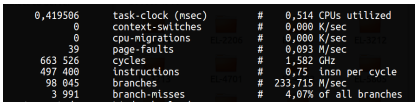

Fig. 13. Results using $perf$ with 256 bit denominator division.

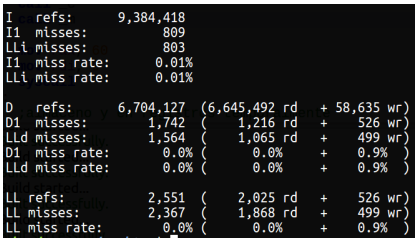

Fig. 14. Results using $valgrind$ cachetool.

Based on the previous performance tests, we can conclude that in the case of 64 bit denominator division and 256 denominator division we can see a difference in task-clock. Interestingly there were more instructions per cycle in the 256 bit denominator subroutine.

References

[1] 4.2.4 Extended Precision Multiplication . (2019). Plantation-productions.com. Retrieved 20 May 2019, from http://www.plantation-productions.com/Webster/www.artofasm.com/Linux/HTML/AdvancedArithmetica2.html