Instituto Tecnológico
y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática
ESPECIALIDAD EN SISTEMAS EMBEBIDOS

Montgomery Algorithm Implementation on an Embedded System for a 256-bit input size

---

TRABAJO RECEPCIONAL que para obtener el GRADO de
Especialista en Sistemas Embebidos

Presenta: **ADRIANA ARIZAGA JASSO**

Asesor **LUIS JULIÁN DOMÍNGUEZ PÉREZ**

Tlaquepaque, Jalisco. Julio 2021.

Montgomery Algorithm Implementation on an Embedded System for a 256-bit input size

Arizaga Jasso, Adriana
Embedded Systems
Specialization ITESO
Tlaquepaque, Jal. México
aajasso@iteso.mx

Domínguez Pérez, Luis Julián
Department of Electronics,
Systems and Informatics ITESO
Tlaquepaque, Jal. México
luisjdominguezp@iteso.mx

***Abstract*—** The Montgomery multiplication is a leading method to compute modular multiplications faster over large prime fields. Numerous algorithms in number theory use Montgomery multiplication computations. This fast data processing makes it appealing to cryptosystem analysis. The objective of this work is to implement the Montgomery algorithm on an embedded system. For this application, the following 256-bit arithmetic functions were executed in the MCUXpresso IDE software: adder, subtraction, multiplication, and Barret reduction. The obtained results in the FRDM-K64F board show the Montgomery form values, and the product out of the Montgomery domain. The operations computed in the embedded board also demonstrate that the applied algorithms are congruent with the values obtained in C programming, Python, and the FRDM-K64F board.

Keywords*— *Montgomery, Barret Reduction, Modular Arithmetic

I. Introduction

Electronic transactions and Internet security have become an essential part of daily life. To secure digital communications, cryptographic algorithms convert a plaintext message into an encrypted ciphertext. Since quantum computers and some mathematical algorithms can also solve encrypted data sent across the internet, encrypted data and public keys could be cracked, and cryptosystem keys revealed [4]. For this reason, algorithms that provide either quantum or classical security are needed, like the SIKE (Supersingular Isogeny Key-Encapsulation) algorithm. SIKE employs the Montgomery multiplier since it is the most efficient method for performing multiplications with large numbers. We have implemented the Montgomery Algorithm for 256-bit inputs in an embedded board (FRDM-K64F) from NXP. This implementation can be used in an embedded device for cryptosecurity applications. To be able to work with larger numbers than the standard library allows, groups of four arrays are used in every module of the program. The selected board FRDM-K64F has an ARM Cortex-M4 Core running up to 120MHz. It handles 16-bit ADCs, a DAC, and a variety of peripherical and interfaces, also hardware encryption, supporting CRC, DES, 3DES, AES, MD5, SHA-1 and SHA-256 algorithms. It works on a voltage range from 1.71 to 3.6V

II. Mathematical Background

A. *Modular Arithmetic*

The finite field ($\mathbb{F}_p$) arithmetic for a prime number (p) is the modular arithmetic Mod p. Let $a, b \in \mathbb{F}_p$, the addition in the $\mathbb{F}_p$ is defined as $(a + b)\ mod\ p$. The output carry of the previous addition needs to be added, then $c_i = (a_i + b_i + carry)$ is calculated while the operation in the subtraction implements borrows as inputs. The integer multiplication $t = (a \cdot b)$ is carried out word by word according to the school-book method referred in the "Fig.1" at a computational cost of n^2 and $(n-1)^2$ [5].

$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & & b_3 & b_2 & b_1 & b_0 \\
\multicolumn{8}{c}{\text{---------------------------------}} \\
 & & & & pp_{03} & pp_{02} & pp_{01} & pp_{00} \\
 & & & pp_{13} & pp_{12} & pp_{11} & pp_{10} & \\
 & & pp_{23} & pp_{22} & pp_{21} & pp_{20} & & \\
 & pp_{33} & pp_{32} & pp_{31} & pp_{30} & & & \\
\hline
r_7 & r_6 & r_5 & r_4 & r_3 & r_2 & r_1 & r_0
\end{array}
$$

Fig. 1. School book method.

The Barret reduction algorithm is defined by formula (1).

$$Q = \lfloor (t/b^{k-1}) \cdot \mu \cdot (1/b^{k+1}) \rfloor \quad (1)$$

The Barret Reduction algorithm resolves $t\ mod\ p$ (for a positive integer t and a modulus p) where p is a prime number such that $|t| \approx 2|p|$ [5].

Where:

$$\mu = b^{2k}/p \quad (2)$$

$$k = \lceil log_b p \rceil + 1 \quad (3)$$

$$t = Qp + R \quad (4)$$

Where 0≤R<p, the quotient Q= $\left\lfloor \frac{t}{p} \right\rfloor$ can be written as,

$$Q = \left[\frac{t}{b^{k-1}}\right] \cdot \left(\frac{b^{2k}}{p}\right) \cdot \left(\frac{1}{b^{k+1}}\right) = \left[\frac{t}{b^{k-1}} \cdot \mu \cdot \left(\frac{1}{b^{k+1}}\right)\right] \quad (5)$$

The remainder is obtained as:

$$r = (t - q \cdot p) mod\ b^{k+1} \quad (6)$$

The Montgomery multiplication is executed using $\mathbb{F}_p$ elements in the Montgomery representation which requires the modular reduction. Cryptographic implementations require at least 112 bits of security level, though a 128-bit security level is preferred. This leads to use prime numbers that are larger than 256 bits. The Barret, and Montgomery algorithms employ the modular reduction. The Montgomery multiplier is one of the most efficient methods for large number multiplication as it reduces the number of calculations [5].

The Montgomery product is defined by the equation (7), which can be used to compute the field multiplication equation (8).

$$MontPr(\tilde{a}, \tilde{b}) = \tilde{a} \cdot \tilde{b} \cdot r^{-1}\ mod\ p \quad (7)$$

$$c = a \cdot b\ mod\ p \quad (8)$$

III. Methodology

In this work, every implemented module was tested separately, and the result was compared with a Python implementation. After verifying that the independent modules worked accordingly, they were placed together inside the main code and tested as a whole. The code execution time was measured using the Debug Watch and Trace module included in the Cortex-M ARM processor. The original programming was performed in a C X86_64 architecture Linux compiler, then it was converted to the embedded software board and finally downloaded to the NXP FRDM-K64 board. Considering the prime number in the Montgomery multiplication is 256-bit size, groups of four arrays unsigned integer 64-bits were employed. For the adder section from "Fig. 2", both t and $u0$ inputs are compared with the maximum possible array input value `0XFFFFFFFFFFFFFFFF`. If either input has the maximum value, a carry is generated and this carry is added to the result from adding $t + u0$.

Fig. 2. 256 bits Adder

```
Inputs:  t[4],u0[4]
Outputs: carry,r[3],r[2],r[1],r[0]
1: r[0]=t[0]+u0[0]
2: if(t[0]==0XFFFFFFFFFFFFFFFF | u0[0]==
   0XFFFFFFFFFFFFFFFF)
3: carry=1
4: else
5: carry=0
6: for (i=1; i<=3; i++)
7: r[i]=t[i]+u0[i]+carry
8: if (t[i]== 0XFFFFFFFFFFFFFFFF |
   u0[i]==0XFFFFFFFFFFFFFFFF)
9: carry=1
10: else carry=0
```

Fig. 2. 256-bits adder. If any input has the maximum value, a carry is generated.

The subtractor implementation stated in the "Fig. 3" performs a comparison if the second operator is higher than the first operator, the carry represented as `0xFFFFFFFFFFFFFFFF` is added to the adjacent more significant array number value.

Fig. 3. Subtractor 256-bits

```
Inputs: u [4], p [4]
Outputs: c_t[3],c_t[2],c_t[1],c_t[0]
1: for (i=0; i<=3; i++)
2: c_t[i]=u[i]-p[i]
3: for (i=0; i<4; i++)
4: if (p [0]> u [0])
5: c_t[i+1] =(c_t[i+1]
   +0xFFFFFFFFFFFFFFFF)
```

Fig. 3. Subtraction 256-bits. If the second term is larger than the first term, a carry is generated.

The one-digit multiplication code defined in "Fig. 4" is used in the multiplication code, as the School Book Method states. A carry that initializes with cero is added to this partial multiplication in the variable x. The variable N is equal to 16 due every array size is equal to 4, for this reason the multiplication product doubles to N^2. Where $N = 16$ for the leftShifting and Addition codes described below. The variable $temp$ is equal to x `AND 0XFFFFFFFF`. This takes the 64-bits lowest part half for the $temp$ variable. The carry is 32-bits shifted right. If carry is different to zero, the $temp$ highest significant bit becomes equal to carry.

Fig. 4. onedigitmultiplication

```
Inputs:  dataA[16],uint64 dataB
Output:  dataP[16],
1: for (n=0; n < N; n++)
2: x = (dataA[n] · number) +
   carryOverflow
3: temp[n] = x & 0XFFFFFFFF
4: carryOverflow = (x >> 32) & 0xFFFFFFFF
5: if (carryOverflow!= 0)
6: temp [16] = carryOverflow
```

Fig. 4. Onedigitmultiplication represents the operation of the multiplication between the two terms digit by digit.

In the leftShifting code from the "Fig. 5", the multiplication intermediate results are left shifted as part of the School Book Method multiplication process. The numbers are shifted to represent the units, the tens, and so on.

Fig. 5. leftShifting

```
Inputs: dataA[16],dataB[16]
Output: dataP[16]
1: for (i=N-1; i>= number; i--)
2: dataA[i] = dataA [i- number]
3: while (i >= 0)
4: dataA[i--] = 0
```

Fig. 5. The purpose of the LeftShifting multiplication algorithm is to shift the partial result for place it, according to units, tens, or hundreds.

In the addition code declared in "Fig. 6", *sum* is equal to the partial result addition plus the generated carry. If sum is higher than the maximum number an overflow is generated.

Fig. 6 Addition

```
Inputs:  dataA[16],dataB[16]
Output: dataC[16]
1: for (j=0; j<N; j++)
2: sum = dataA[j] + dataB[j] +
   overflowCarry
3: if (sum > 0xFFFFFFFF)
4: overflowCarry = (sum >> 32) &
   0xFFFFFFFF
5: else overflowCarry = 0
6: dataC[j] = sum & 0XFFFFFFFF
7: if (carry)
8: print "overflow"
```

Fig. 6. The Addition 256-bits adds the multiplication partial results to obtain the final product.

"Fig. 7" refers to the 256-bits multiplier code, where every input array goes thru the previously codes 3,4, and 5 (onedigitmultiplication, leftShifting and addition).

Fig. 7. Multiplier 256 bits

```
Inputs: A[16],B[16]
Output: C[16]
1: for (i = 0; i < 16; i++)
2: onedigitmultiplication (A,P,B[i])
3: leftShifting(P,i)
4: addition (C,P,C)
```

Fig. 7. Multiplier 256-bits. This part of code calls the previous functions that are needed by the entire multiplication process.

The Barret reduction code from "Fig. 8" employs the positive integers inputs: t,p and b. Where $t \approx 2|p|$, μ is calculated by (2), and b is selected as a power of two. [5]. Then formulas (4) and (5) are computed. While the Barret result is higher or equal to the prime number array, the result is then equal to this computed result minus the prime number p. For this study, the prime number p, the t value, and b values were defined as shown in the Fig. 8 code.

Fig. 8 Barret Reduction

```
Inputs: A[4],p[4]={0x21000000000013a7,
0x21000000000013a7,0xa344d80000000061,
0x252364820000001b},
t[4]={0x3d27f7a581a1b7aa,
0x2b0953390ec8b4f3,0x4778503e454a0a70,
0xa84a439ff8959361},
b[4]={0x10,0x4,0x8,0x10}
Output: r [4]
1: for (i=0; i<4; i++)
2: miu[i]=pow(b[i],2·k) /p[i]
3: q[i]=(t[i]/pow(b[i], (k1))
   ·(miu[i]/pow(b[i],(k+1))))
4: r[i] = remainder(t[i], pow(b[i],
   (k+1)))- (q[i] · remainder(p[i],
   pow(b[i], (k+1))))
5: if (r[i]<0)
6: r[i]=(r[i]+ pow(b[i],(k+1)))
7: while (r[i]>=p[i])
8: r[i]=(r[i]-p[i])
```

Fig. 8. This code represents the execution of the Barret Reduction algorithm.

"Fig. 9" [5] page 5-8, shows the pseudocode to compute the Montgomey domain result. In step 1 where $t = \tilde{a} \cdot \tilde{b}$, $\tilde{b} = D \bmod p$, and $D = B \cdot rr$. Also, $\tilde{a} = C \bmod p$, and $C = A \cdot rr$. To avoid an expensive operation, the precomputation of p between rr was done. Since this algorithm works only in the Montgomery space, a transformation of the numbers into that space is needed before the multiplication process begins.

Fig. 9 Montgomery

```
Inputs: A[4]={}, B[4]={},
p[4]={0x21000000000013a7,
0x21000000000013a7,0xa344d80000000061,0x2
52364820000001b};
rr[4]={0x1000,0x1000,0x1000,0x1000}
p_betw_rr [4] = {0x2,0x0ffdf00000013a72,
0xe58be000000274eb, 0xdf5f2bb280027afe}
Outputs:c_t(Montgomery Product Result)
1: t=ã·b̃
2: q0= t·b
3: q= q0 mod rr
4: u0 = q*( p_betw_rr)
5: u = t+u0
6: if u[]>p[]
7: return c_t= u-p
8: else
9: return c_t=u
```

Fig. 9. The application of the Montgomery code obtains the Montgomery domain result.

IV. Results

"Fig. 10" represents the adding operation with maximum input values.

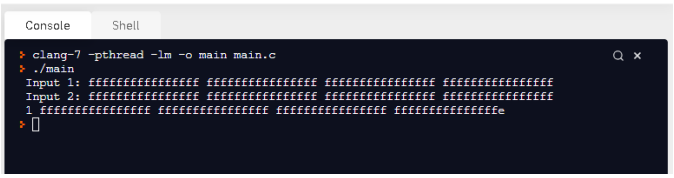

Fig. 10. 256-b Adder test in C compiler.

"Fig.11" displays the adder operation in Python.

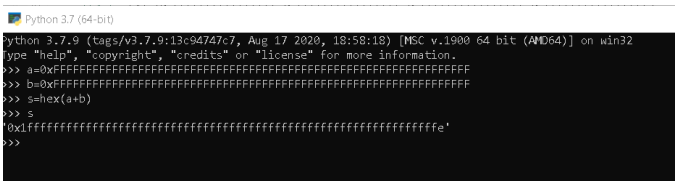

Fig. 11. Adder tested in Python.

"Fig. 12" shows the adder result in the FRDM-K64F board with an execution time of 512 CPU clock cycles.

Fig. 12. Adder computation result on embedded board.

"Fig. 13" displays the subtraction operation result in C compiler.

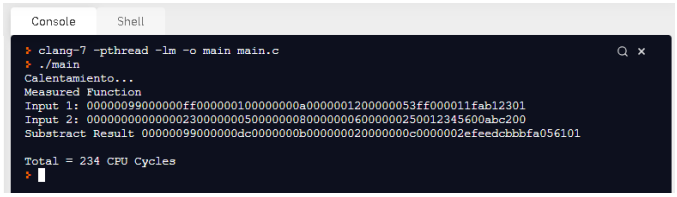

Fig. 13. Subtractor computation in C compiler.

"Fig. 14" shows the subtractor operation in Python.

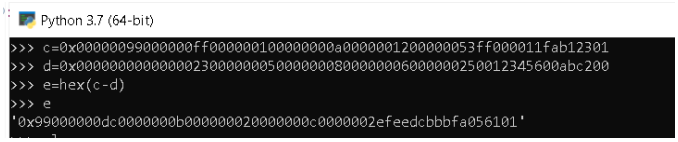

Fig. 14. Subtractor test in Python.

"Fig. 15" represents the subtraction result in the FRDM-K64F board, with an execution time of 127 clock cycles.

Fig. 15. Subtractor in embedded board.

"Fig. 16" shows the multiplication operation in C compiler.

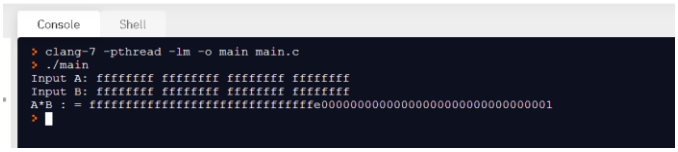

Fig. 16. Multiplication test in C compiler.

"Fig. 17", corresponds to the multiplication operation in Python.

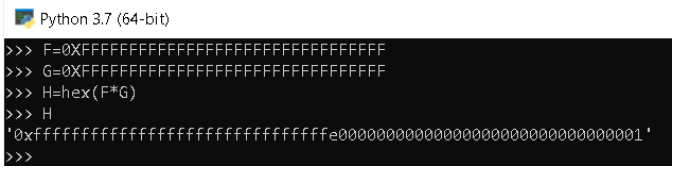

Fig. 17. Multiplication test in Python.

"Fig. 18" represents the Montgomery result on the FRDM-K64F board. The program contains the previous presented modules: adder, subtraction, and Barret.

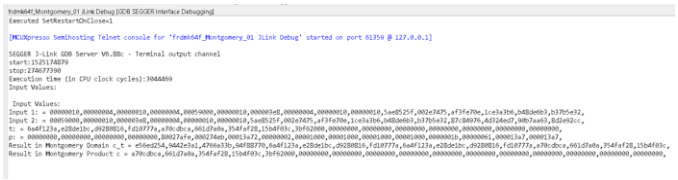

Fig. 18 Montgomery result in the embedded board.

V. Conclusions

In this work, the 256-bits Montgomery algorithm was implemented on the NXP FRDM-K64F board using the NXP software MCUXpresso IDE v11.3.0. For this purpose, the 256-bits arithmetic functions were executed in a C compiler: adder, subtract, multiplication and Barret reduction. Since the C compiler does not include the libraries for larger numbers that 64 bits, the input number was split in arrays, without the need of using external libraries. For the 256-bits adder implementation, a comparation to check if the inputs had the maximum value number was performed, thus a carry was assigned to the operation. Montgomery devices offers a competent performance when used in complex cryptography for Internet of Things systems and can be implemented in present-day cryptography. As a further work we would like to test this Montgomery algorithm in the SIKE algorithm.

References

[1] D. Jao, "Supersingular Isogeny Key Encapsulation", *Sike.org*, 2021. [Online]. Available: https://sike.org/files/SIDH-spec.pdf. [Accessed: 08- Jul- 2021].

[2] W. Diffie and M. Hellman, "New directions in cryptography," in IEEE Transactions on Information Theory, vol. 22, no. 6, pp. 644-654, November 1976, doi: 10.1109/TIT.1976.1055638.

[3] K. Ruohonen "Mathematical Cryptology", September 2014

[4] D. B. Roy, T. Fritzmann and G. Sigl, "Efficient Hardware/Software Co-Design for Post-Quantum Crypto Algorithm SIKE on ARM and RISC-V based Microcontrollers," *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1-9

[5] Marc Joye., Nadia El Mrabet. and El Mrabet., *Guide to Pairing-Based Cryptography*.

[6] L. Washington, *Elliptic curves*. Boca Raton: Chapman and Hall/CRC, 2008.

[7] S. Nagaich and Y. C. Goswami, "Shor's Algorithm for Quantum Numbers Using MATLAB Simulator," 2015 Fifth International Conference on Advanced Computing & Communication Technologies, 2015, pp. 165-168, doi: 10.1109/ACCT.2015.16.

[8] M. Mihailescu and S. Nita, *Pro Cryptography and Cryptanalysis*.

[9] "FRDM-K64F Platform|Freedom Development Board|Kinetis MCUs | NXP Semiconductors", *Nxp.com*, 2021. [Online]. Available: https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F. [Accessed: 14- Jul- 2021].

[10] M. Scott, *On the Deployment of curve based cryptography for the Internet of Things*. 2020.