Centro de Investigación y de Estudios Avanzados
Del Instituto Politécnico Nacional

Laboratorio de Tecnologías de la Información

Reporte Técnico:

Arquitecturas hardware compactas del algoritmo Montgomery para multiplicación en campos finitos $\mathbb{GF}(p^k)$

Miguel Morales Sandoval, Arturo Díaz Pérez

CINVESTAV TAMAULIPAS. LABORATORIO DE TECNOLOGÍAS DE LA INFORMACIÓN.
Parque Científico y Tecnológico TECNOTAM – Km. 5.5 carretera Cd. Victoria-Soto La Marina.
C.P. 87130 Cd. Victoria, Tamps.

Cd. Victoria, Tamaulipas, México. Julio, 2014

Resumen:

En este documento se describe el funcionamiento del algoritmo Montgomery para multiplicación modular en campos finitos $\mathbb{GF}(p^k)$, $k = 1$, y los diversos enfoques en la literatura para implementarlo en FPGAs. A continuación, se describe una arquitectura hardware, la cual implementa una versión digito-serial del método Montgomery para realizar la multiplicación modular iterativamente, explotando el paralelismo en las instrucciones. Los resultados de implementación en los FPGA comerciales Spartan3E y Virtex5 muestran que un multiplicador compacto para campos finitos $\mathbb{GF}(p)$ es posible, y puede usarse como un módulo dedicado para acelerar operaciones aritméticas en esquemas de cifrado de llave pública en sistemas embebidos y de cómputo móvil.

Este trabajo es apoyado por el Consejo Nacional de Ciencia y Tecnológia de México (CONACyT), a través del proyecto FORDECYT 2011-01-174509.

PALABRAS CLAVE: Campos finitos, Criptografía, Multiplicación modular, Montgomery

Autor correspondiente: Miguel Morales-Sandoval <mmorales@tamps.cinvestav.mx>

Fecha de envío: 6 de Julio, 2014

Este documento debería ser citado como: M. Morales-Sandoval, Arturo Díaz Pérez. Arquitecturas hardware compactas del algoritmo Montgomery para multiplicación en campos finitos $\mathbb{GF}(p^k)$. Reporte Técnico Número 1, 38 págs. CINVESTAV-Tamaulipas, 2014.

Lugar y fecha de publicación: Ciudad Victoria, Tamaulipas, MÉXICO. Julio 6, 2014

Contenido

1 Idea general del algoritmo Montgomery

El algoritmo Montgomery fue propuesto en 1963 por el matemático P. Montgomery [Montgomery, 1985]. Aunque más bien, este algoritmo se enfoca en realizar la reducción modular a través de operaciones menos costosas que el método tradicional, el cual consiste en hacer la división y tomar el residuo como resultado. Sin embargo, se le da el nombre de Multiplicación Montgomery cuando la reducción se hace con el método propuesto por Montgomery. La reducción modular puede hacerse usando cualquier otro método, como el ya citado método tradicional o el método Barret [Knezevic et al., 2010].

Un enfoque para calcular $c \bmod p$ es realizar restas sucesivas de p sobre c hasta que éste sea un número menor que p. Cuando p es muy grande, el numero de restas puede ser muy grande por lo que este enfoque resultaría ineficiente. Si se pudieran saber cuantas restas son necesarias, el problema sería mas simple, pero precisamente eso es lo que se hace en el método escolar para calcular el residuo: se calcula el número de restas de p a c a través de una división entera, y después se hace esas restas a c: el número de restas es c/p, el numero a restar es $p * (c/p)$ (que no es c, a menos que la división c/p sea exacta, lo que no ocurre ya que p es un número primo) y entonces el residuo sería $c - p * (c/p)$.

Ejemplo. Calcular $23 \bmod 7$. En este caso, $c = 23$ y $p = 7$, $c/p = 3$, $p * c/p = 21$, $c - p * c/p = 2$, por lo que $23 \bmod 7 = 2$.

El método Montgomery usa un enfoque parecido al del ejemplo anterior, solo que en lugar de calcular cuanto restar a c, se calcula cuanto sumarle, a fin de obtener un valor de c que pueda ser dividido fácilmente, típicamente haciendo algún tipo de corrimiento el cual no implica un costo significativo. El método Montgomery usa el hecho que $c \bmod p = c + (N \times p) \bmod p$. No importa cuantas veces le sumemos p a c el resultado siempre será el mismo. La idea es sumarle un número tal que $(c + N \times p)$ pueda dividirse fácilmente por un factor r. En el método Montgomery, todas las operaciones módulo y división se realizan respecto a un factor r el cual es escogido para que dichas operaciones se realicen fácilmente. El factor r es mayor que p y realtivamente primo a él. Típicamente, $r = 2^m$, para algún entero positivo m. Así, la

reducción modulo r consiste en descartar bits mas significactivos de $(c + N \times p)$ mientras que la división por r consiste en descartar bits menos significativos de $(c + N \times p)$.

Dado que $gcd(r, p) = 1$, existen dos números r^{-1} y p', $0 < r^{-1} < p$ y $0 < p' < r$ tales que $r \times r^{-1} - p \times p' = 1$. Para realizar una reducción Montgomery es necesario que estos valores **esten previamente calculados**. Para realizar una multiplicación de dos números a, b con reducción módulo p es necesario que a, b se transformen a una representación diferente, es decir, es necesario transformar a a, b al dominio de Montgomery. Esta transformación se realiza como se muestra en las ecuaciones 1 y 2.

$$a' = a \times r \bmod p \tag{1}$$

$$b' = b \times r \bmod p \tag{2}$$

Mientras que a es un número "normal", a' lo llamaremos un número Montgomery. Así, la reducción Montgomery recibe como entrada un número z' y devuelve como resultado $z' \times r^{-1} \bmod p$.

$$z'r^{-1} = z'rr^{-1}/r = z'(pp' + 1)/r = (z'pp' + z')/r = (z'p'p + z')/r \tag{3}$$

La reducción Montgomery se lista formalmente en el algoritmo 1. La última expresión de la ecuación anterior se ejecuta en dos pasos diferentes del algoritmo 1. Primero se calcula $z' * p'$ el el paso 1. Después, en el paso 2 se realiza el cálculo final. En este paso $(z'p'p+z')$ es fácilmente divisible por r (i.e. $z'p'p+z' \bmod r = 0$, o dicho de otra forma, $z'p'p$ es el inverso aditivo de z' módulo r). Suponiendo que $z' < rn$, el valor $a = (z'p'p + z')/r$ será menor que $2p$, por lo que si $a > p$, la operación módulo p puede calcularse con una simple resta para si tener $z'r^{-1} \bmod p$ calculado.

Supongamos que tenemos dos números Montgomery y queremos realizar la multiplicación entre ellos. Esta multiplicación la podemos hacer en el dominio de Montgomery usando el algoritmo 2.

La reducción Montgomery en la multiplicación de a', b' permite que el valor resultante siga siendo un número Montgomery. En el algoritmo 2, $t' = a' \times b' = (a \times r \bmod p) \times (b \times r \bmod p) = (a \times b) \times r^2 \bmod p$. Cuando se calcula z' lo que se obtiene es $t' \times r^{-1} = (a \times b) \times r \bmod p)$.

El algoritmo 2 puede utilizarse para realizar la transformación de $a \rightarrow a'$ haciendo la invocación

Algorithm 1 Algoritmo para reducción Montgomery: *REDUCE(z')*

Entrada: z'
Salida: $z' \times r^{-1} \bmod p$
1: $q \leftarrow (z' \bmod r) \times p' \bmod r$
2: $a \leftarrow (z' + qp)/r$
3: **if** $a > p$ **then**
4: $\quad a \leftarrow a - p$
5: **end if**
6: **return** a

Algorithm 2 Algoritmo multiplicación Montgomery: *MM(a', b')*

Entrada: $a' = a \times r \bmod p$
Entrada: $b' = b \times r \bmod p$
Salida: $a' \times b' \times r^{-1} \bmod p$
1: $t' \leftarrow a' \times b'$
2: $z' \leftarrow$ *REDUCE(t')*
3: **return** z'

MM(a, r^2). De igual forma, se puede realizar la conversión $a' \rightarrow a$ haciendo MM$(a', 1)$. En ambos casos, debe realizarse una multiplicación seguida de una reducción Montgomery. Para el primer caso, la multiplicación resulta trivial si r es una potencia de 2, y para el segundo la multiplciación, evidentemente, es simbólica. La siguiente secuencia de pasos calcula $c = a \times b \bmod p$ usando llamadas a función MM del algoritmo 2.

1. a' = MM(a, r^2)
2. b' = MM(b, r^2)
3. t' = MM(a', b')
4. c = MM$(t', 1)$

Es evidente que el cálculo de c usando el algoritmo Montgomery resulta mucho mas costoso que hacerlo con el método clásico. Sin embargo, cuando el número de multiplicaciones es muy grande, el ahorro en tiempo de cómputo es notoria. Esto es lo que ocurre en una operación de exponenciación con reducción modular, la cual es una operación típica en esquemas criptográficos, como RSA o el protocolo Diffie-Hellman para intercambio de llaves, donde es necesario realizar exponenciaciones de la forma $x^n \bmod p$, siendo n un número grande, suficientemente grande para resistir un ataque de fuerza bruta (i.e. un número de mas de 80 bits). En la práctica, los pasos 1, 2 y 4 se calculan solo al principio y al final de una exponenciación.

Tabla 1: Cálculo de $c = (61 \times 5) \text{ mód } 79 = 305 \text{ mód } 79 = 68$, usando el método Montgomery.

Conversión de a a un número Montgomery	$a' = \text{MM}(61, 100^2)$	1. $t' = 61 \times 10000 = 610000$ 2. $z' = REDUCE(610000)$ $q = (610000 \text{ mód } 100) \times 81 \text{ mód } 100$ $= 0$ $a = (610000 + 0 \times 79)/100$ $= 6100 = 61 \times 100 \text{ mód } 79 = 17$
Conversión de b a un número Montgomery	$b' = \text{MM}(5, 100^2)$	1. $t' = 5 \times 10000 = 50000$ 2. $z' = REDUCE(50000)$ $q = (50000 \text{ mód } 100) \times 81 \text{ mód } 100$ $= 0$ $a = (50000 + 0 \times 79)/100$ $= 500 = 5 \times 100 \text{ mód } 79 = 26$
Multiplicación y reducción Montgomery	$t' = \text{MM}(a', b')$	1. $t' = 17 \times 26 = 442$ 2. $z' = REDUCE(442)$ $q = (442 \text{ mód } 100) \times 81 \text{ mód } 100$ $= 42 \times 81 \text{ mód } 100$ $= 3402 \text{ mód } 100 = 2$ $a = (442 + 2 \times 79)/100 = 600/100 = 6$
Conversión del número Montgomery a su representación normal	$c = \text{MM}(t', 1)$	1. $t' = 6 \times 1 = 6$ 2. $z' = REDUCE(6)$ $q = (6 \text{ mód } 100) \times 81 \text{ mód } 100$ $= 6 \times 81 \text{ mód } 100$ $= 486 \text{ mód } 100 = 86$ $a = (6 + 86 \times 79)/100$ $= 6800/100 = 68$

En la tabla 1 se muestra un ejemplo del algoritmo Montgomery para multiplicación de dos números con reducción modular. Sean $a = 61, b = 5, p = 79$. Entonces, sabemos que $c = a \times b \text{ mód } p = (61 \times 5) \text{ mód } 79 = 305 \text{ mód } 79 = 68$.

Por simplicidad, sea $r = 100$. Se puede demostrar que $100 \times 64 - 79 \times 81 = 1$. De aquí que $r^{-1} = 64, p' = 81$. Usando el método Montgomery, los cálculos serían los que se muestran en la tabla 1.

Como se observa en la columna 3, se deben realizar 3 multiplicaciones de números enteros, dos reducciones módulo r y una división módulo r, las cuales como ya se ha comentado son relativamente simples de hacer.

Tabla 2: Bases de representación númerica típicas y notación númerica posicional.

Base	Digitos	Ejemplo
2	$\{0, 1\}$	$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
10	$\{0, 1 \cdots 8, 9\}$	$6731 = 6 \times 10^3 + 7 \times 10^2 + 3 \times 10^1 + 1 \times 10^0$
16	$\{0, 1 \cdots 14 = E, 15 = F\}$	$10EF = 1 \times 16^3 + 0 \times 16^2 + E \times 16^1 + F \times 16^0$

2 Implementación del algoritmo Montgomery

La implementación del algoritmo Montgomery en computadoras digitales requiere un claro entendimiento de la representación de los números y las propiedades asociadas a dicha representación. Cualquier número a puede representarse en una base dada, tambien conocida como **radix**. Los sistemas de numeración basan su funcionamiento de acuerdo a un esquema posicional respecto a la base usada. Así, los números de un campo F necesitan tener una base de representación, de la cual se determinan los digitos que permiten representar cualquier número perteneciente a F. En la tabla 2 se describen las bases más comunmente utilizadas para la representación de números enteros.

A partir de la tabla **??**, se puede ver como un número puede expresarse como una combinación lineal de potencias de la base usada en su representación numérca. Así, las siguientes expresiones son equivalentes:

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \tag{4}$$

$$1101 = (11) \times 2^2 + (01) \tag{5}$$

$$1101 = 1 \times 2^3 + (10) \times 2^1 + 1 \times 2^0 \tag{6}$$

$$1101 = (110) \times 2^1 + 1 \times 2^0 \tag{7}$$

Debido a que las computadoras representan a los números como cadenas de n bits, es muy común describir a dichos números mediante la expresión $a = \sum_{i=0}^{n-1} \alpha_i 2^i$, siendo la base 2, y $\alpha \in \{0, 1\}$. La expresión puede generalizarse para cualquier base β con la expresión $a = \sum_{i=0}^{m-1} \alpha_i \beta^i$, $\alpha_i \in \{0, 1, \cdots, \beta - 2, \beta - 1\}$.

Sean $x = \sum_{i=0}^{m-1} x_i \beta^i$, $y = \sum_{i=0}^{m-1} y_i \beta^i$. Sea $r = \beta^m$. En una multiplicación Montgomery $MM(x, y) =$

$x \times y \times r^{-1}$, el resultado será $A = x \times y \times \beta^{-m} = \sum_{i=0}^{m-1} a_i \beta^i$.

$$
\begin{aligned}
A =& x \times y \times \beta^{-m} \\
=& \left(\sum_{i=0}^{m-1} x_i \beta^i \right) \times y \times \beta^{-m} \\
=& \left(\sum_{i=0}^{m-1} x_i \beta^i \times y \right) \times \beta^{-m} \\
=& \left(x_0 \beta^0 \times y \right) \times \beta^{-m} + \\
=& + \left(x_1 \beta^1 \times y \right) \times \beta^{-m} + \\
& + \cdots \\
& + \left(x_i \beta^i \times y \right) \times \beta^{-m} + \\
& + \cdots \\
& + \left(x_{m-1} \beta^{m-1} \times y \right) \times \beta^{-m} \\
=& \sum_{i=0}^{m-1} REDUCE(x_i \beta^i \times y).
\end{aligned}
\tag{8}
$$

Analicemos con cuidado la ecuación 8. A primera vista, pareciera que solo se necesitarían m llamadas a la funcion REDUCE para calcular A iterativamente. Sin embargo, la suma acumulativa de cada término de la sumatoria en la ecuación 8 implicaria nuevamente una reducción, del término actual a reducir con el valor previamente calculado de A. En la función REDUCE($x_i \beta^i \times y$), los cálculos serian los siguientes:

$$
\begin{aligned}
q =& ((x_i \beta^i \times y) \bmod \beta^m) \times p' \bmod \beta^m \\
=& ((x_i \times y) \bmod \beta^m) \times p' \bmod \beta^m \\
=& ((x_i \times y) \bmod \beta) \times p' \bmod \beta \\
a =& ((x_i \beta^i \times y) + q \times p)/\beta^m \\
=& ((x_i \times y) + q \times p)/\beta
\end{aligned}
\tag{9}
$$

Así, A se calcularía en m iteraciones, siendo A_i el valor de A en la iteración i, resultado de la llamada a la función $A_i = REDUCE(A_{i-1} + x_i \beta^i \times y)$. Los cálculos que se hacen en la función REDUCE son:

$$q = (A_{i-1} + x_i \times y) \times p' \bmod \beta \tag{10}$$

$$a = ([A_{i-1} + x_i \times y] + q \times p)/\beta \tag{11}$$

2.1 Cálculo de p'

En el algoritmo Montgomery es necesario pre-calcular p' y r^{-1}, donde $r \times r^{-1} - p \times p' = 1(\text{mód}\beta^m)$. De esta igualdad se desprende que $-p \times p' \bmod \beta = 1$, y por tanto que p' es la inversa modulo β de $-p$ en base β, es decir $p' \bmod \beta = (-p)_\beta^{-1}$. Además se tiene que $-p = \beta - p_0$, siendo p_0 el primer dígito de p. Si suponemos que la base $\beta = 2^k$, para algun entero positivo k y que $p_0 = \beta - 1$. Entonces se tiene que:

$$\begin{aligned} p' \bmod \beta &= (-p)_\beta^{-1} \\ &= (\beta - p_0)_\beta^{-1} \\ &= 1. \end{aligned} \tag{12}$$

Por tanto, la ecuación 10 quedaria de la siguiente forma:

$$\begin{aligned} q &= (A_{i-1} + x_i \times y) \times p' \bmod \beta \\ &= (A_{i-1} + x_i \times y) \bmod \beta \\ &= ([a_0 + x_i \times y_0] \bmod \beta \end{aligned} \tag{13}$$

Al realizarse una operación módulo β, la ecuación 13 solo involucra el primer dígito de $(A_i + x_i \times y)$ y por tanto es el único que se utiliza en todo los cálculos. Así, el algoritmo iterativo que calcula la multiplicación Montgomery $a = x \times y \bmod p$ queda definido como se describe en el algoritmo 3.

2.2 Otro enfoque sobre Montgomery iterativo

Reconsidere la definición de una multiplicación Montgomery. Sea $z = x \times y$, entonces $A = REDUCE(z) = x \times y \times \beta^{-m}$, donde A se calcula de la siguiente forma.

Algorithm 3 Algoritmo iterativo para multiplicación Montgomery: *MM(*x, y*)*

Entrada: $a' = a \times \beta^m \bmod p = \sum_{i=0}^{m-1} x_i \beta^i$
Entrada: $b' = b \times \beta^m \bmod p \sum_{i=0}^{m-1} y_i \beta^i$
Salida: $a = x \times y \times \beta^{-m} \bmod p = \sum_{i=0}^{m-1} a_i \beta^i$
1: $A_0 \leftarrow 0$
2: **for** $i \leftarrow 0$ to $m-1$ **do**
3: $\quad q \leftarrow (a_0 + x_i \times y_0) \bmod \beta$
4: $\quad A_{i+1} \leftarrow ([A_i + x_i \times y] + q \times p)/\beta$
5: **end for**
6: **return** A_i

$$q = (z \bmod \beta^m) \times p' \bmod \beta^m \tag{14}$$

$$A = (z + q \times p)/\beta^m. \tag{15}$$

Si $x \times y$ se expresa como $\left(\sum_{i=0}^{m-1} x_i \beta^i\right) \times y$, q y A las ecuaciones 17 y 18 son equivalentes a las ecuaciones 14 y 15.

$$q = \left(\left(\sum_{i=0}^{m-1} x_i \beta^i \times y\right) \bmod \beta^m\right) \times p' \bmod \beta^m \tag{16}$$

$$A = \left(\left(\sum_{i=0}^{m-1} x_i \beta^i \times y\right) + q \times p\right) / \beta^m. \tag{17}$$

Expandiendo la sumatoria de la ecuación 16, tenemos:

$$\begin{aligned} q &= (x_0\beta^0 \times y \bmod \beta^m) \times p' \bmod \beta^m + \cdots + (x_{m-1}\beta^{m-1} \times y \bmod \beta^m) \times p' \bmod \beta^m \\ &= \sum_{i=0}^{m-1} \left[(T_i \beta^i \bmod \beta^m) \times p' \bmod \beta^m\right], T_i = (x_i \times y) \end{aligned} \tag{18}$$

Sustituyendo la ecuación 18 en la ecuación 17 tenemos la siguiente expresión:

$$
\begin{aligned}
a &= \left(\left(\sum_{i=0}^{m-1} x_i \beta^i \times y \right) + q \times p \right) / \beta^m. \\
&= \left[\left(\sum_{i=0}^{m-1} T_i \beta^i \right) + \left(\sum_{i=0}^{m-1} [(T_i \beta^i \bmod \beta^m) \times p' \bmod \beta^m] \right) \times p \right] / \beta^m \\
&= \left[\sum_{i=0}^{m-1} \left(T_i \beta^i + [(T_i \beta^i \bmod \beta^m) \times p' \bmod \beta^m] \times p \right) \right] / \beta^m \\
&= \sum_{i=0}^{m-1} \frac{T_i \beta^i + [(T_i \beta^i \bmod \beta^m) \times p' \bmod \beta^m] \times p}{\beta^m} \\
&= \sum_{i=0}^{m-1} \left[\frac{T_i \beta^i}{\beta^m} + \frac{[(T_i \beta^i \bmod \beta^m) \times p' \bmod \beta^m] \times p}{\beta^m} \right]
\end{aligned} \tag{19}
$$

La expresión q_i es equivalente a $[(T_i \beta^i \bmod \beta^m) \times p' \bmod \beta^m]$, la cual, puede demostrarse que es equivalente a $[T_i \times p' \bmod \beta^{m-i}] \times \beta^i$. La ecuación 20 muestra el resultado de sustituir esta expresión en la ecuación 19.

$$
\begin{aligned}
a &= \sum_{i=0}^{m-1} \left[\frac{T_i \beta^i}{\beta^m} + \frac{[T_i \times p' \bmod \beta^{m-i}] \times \beta^i \times p}{\beta^m} \right] \\
&= \sum_{i=0}^{m-1} \left[\frac{T_i}{\beta^{m-i}} + \frac{[T_i \times p' \bmod \beta^{m-i}] \times p}{\beta^{m-i}} \right] \\
&= \sum_{i=0}^{m-1} \left[\frac{T_i + [T_i \times p' \bmod \beta^{m-i}] \times p}{\beta^{m-i}} \right]
\end{aligned} \tag{20}
$$

La sumatoria en la ecuación 20 puede realizarse progresivamente para calcular $A = x \times y \times \beta^{-m}$, donde la suma acumulativa de los términos de la ecuación 20 en la iteración i, estaría dada por la siguiente recurrencia.

$$A_0 = 0 \tag{21}$$

$$A_{i+1} = \frac{A_i + T_i + q_i \times p}{\beta} \tag{22}$$

2.3 Un ejemplo

Considere el mismo ejemplo de la sección 1. En la tabla 3 se muestra un ejemplo del algoritmo Montgomery iterativo (algoritmo 3) para multiplicación de dos números con reducción modulo. Sean

$x = 61, y = 5, p = 79$. Entonces, sabemos que $A = x \times y \bmod p = (61 \times 5) \bmod 79 = 305 \bmod 79 = 68$. Usemos $\beta = 2$, y $r = \beta^8$.

Primero convertimos los valores x, y al números en el dominio Montgomery. Esto se puede hacer, como ya hemos visto, mediante las operaciones $X = MM(x, (2^8)^2)$ y $Y = MM(y, (2^8)^2)$. Por ello, lo que comúnmente se hace es pre-calcular el valor $R^2 \bmod p$, que en este caso es $(2^8)^2 \bmod 79 = 45$. Así,

$$X = MM(x, r^2) = MM(61, 45) = 53 \tag{23}$$

$$Y = MM(y, r^2) = MM(5, 45) = 16 \tag{24}$$

Estos resultados pueden corroborarse de la siguiente forma.

$$X = 61 \times 2^8 \bmod 79 = 61 \times 256 \bmod 79 = 53 \tag{25}$$

$$Y = 5 \times 2^8 \bmod 79 = 5 \times 256 \bmod 79 = 16 \tag{26}$$

Una vez que hemos obtenido los valores a multiplicar en el dominio Montgomery, los que resta es hacer la operación $MM(X, Y)$. Esta operación debe ser igual a $x \times y \times r \bmod p$.

$$\begin{aligned} MM(X, Y) &= (X \times Y \times r^{-1}) \bmod p = (x \times y \times r) \bmod p \\ MM(53, 16) &= 53 \times 16 \times 2^{-8} \bmod 79 = (61 \times 5 \times 2^8) \bmod p \\ &= 305 \times 2^8 \\ &= 28 \bmod 79. \end{aligned} \tag{27}$$

Sabemos que una operación $MM(X, Y) = (53 \times 16) \times 2^{-8} = 848 \times 2^{-8}$ NO IMPLICA que se tenga que hacer un simple corrimiento de 8 bits a la derecha a 848. Esto se podría hacer siempre y cuando los 8 bits menos significativos de 848 fueran cero, cosa que no ocurre. Ahora, sabemos que el resultado de la función MM debe ser igual ($(x \times y \times \beta^m) \bmod p = 28$). Así el valor esperado del cálculo de $MM(53, 16) = 28 \bmod 79$.

Como $\beta^m = 2^8$, tenemos que X, Y, p son números representados con 8 bits.

Tabla 3: Funcionamiento del algoritmo Montgomery Iterativo para calcular $53 \times 16 \text{ mód } 79$.

Iteración	q, A
0	q = 0 = (0 + 1 x 0) mod 2 A = 8 = (0 + 1 x 16 + (0 x 79)) div 2
1	q = 0 = (0 + 0 x 0) mod 2 A = 4 = (8 + 0 x 16 + (0 x 79)) div 2
2	q = 0 = (0 + 1 x 0) mod 2 A = 10 = (4 + 1 x 16 + (0 x 79)) div 2
3	q = 0 = (0 + 0 x 0) mod 2 A = 5 = (10 + 0 x 16 + (0 x 79)) div 2
4	q = 1 = (1 + 1 x 0) mod 2 A = 50 = (5 + 1 x 16 + (1 x 79)) div 2
5	q = 0 = (0 + 1 x 0) mod 2 A = 33 = (50 + 1 x 16 + (0 x 79)) div 2
6	q = 1 = (1 + 0 x 0) mod 2 A = 56 = (33 + 0 x 16 + (1 x 79)) div 2
7	q = 0 = (0 + 0 x 0) mod 2 A = 28 = (56 + 0 x 16 + (0 x 79)) div 2

$$X = 00110101_2 \tag{28}$$

$$Y = 00010000_2 \tag{29}$$

$$p = 01001111_2 \tag{30}$$

$$r = 100000000_2 \tag{31}$$

Cada una de las iteraciones del algoritmo Montgomery iterativo se detallan en la tabla 3.

Una vez que obtenemos $MM(53, 16) = 28 \text{ mód } 79 = 61 \times 5 \times 2^8 \text{ mód } 79$, lo que resta es convertir este número de vuelta a la representación normal (quitarle el factor 2^8), por lo que esto lo podemos hacer con la llamada $MM(28, 1)$. Ejecutando nuevamente el algoritmo Montgomery iterativo, obtenemos el valor que esperabamos: $MM(28, 1) = 68 \text{ mód } 79$.

Un diagrama ilustrativo del la secuencia de valores obtenidos para el calculo de $61 \times 5 \text{ mód } 79 = 68$ se muestra en la figura 1.

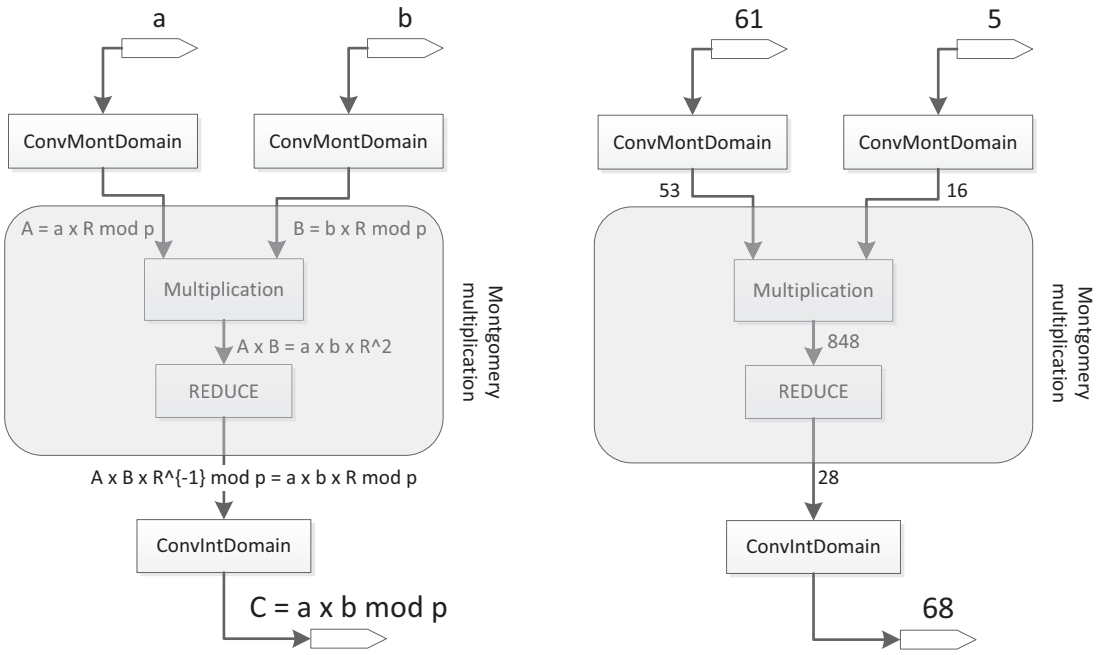

Figura 1: Flujo de datos para el cálculo de una multiplicación modulo p usando el algoritmo Montgomery. A la izquierda el flujo de datos téorico, a la derecha, el flujo de datos para el cálculo de $61 \times 5 \text{ mód } 79 = 68$.

2.4 Algoritmo Montgomery iterativo en Java y generación de vectores de prueba

La multiplicación Montgomery es típicamente usada en criptografía, para cálculo de exponenciaciones, que son operaciones comunes en esquemas de cifrado asimétrico con el algoritmo RSA. En RSA, el cifrado, descifrado, generación y verificación de firma digital se implementan como una operación de exponenciación. La exponenciación es una multiplicación acumulativa de un número perteneciente a un campo finito GF(p).

$$g^a \bmod n = (g \times g \times g \times \cdots \times g) \bmod n \tag{32}$$

$$= (((g \times g \bmod n) \times g \bmod n) \times \cdots \times g) \bmod n \tag{33}$$

La implementación del algoritmo Montgomery iterativo (ver algoritmo 3 en la sección 2) se encuentra disponible en la dirección `http://helios.tamps.cinvestav.mx/~mmorales/FFMul/`. También se encuentra disponible un archivo con vectores de prueba para multiplicación en los campos finitos GF(p) para números de 1024 y 512 bits.

3 Implementación hardware del algoritmo Montgomery

En las implementaciones del Algoritmo Montgomery en hardware para el campo GF(p) se han considerado dos enfoques:

1. Implementar el algoritmo iterativo, requiriendo $n+1$ iteraciones, donde $n = log_2(M)/k$, siendo M el módulo y $\beta = 2^k$ la base de representación de M. Por ejemplo, en la arquitectura reportada en [Mentens et al., 2007] se usan los siguientes módulos.

   a) 4 $n * k$-bit registros

   b) 2 multiplicadores de $(n+1) \times k$ bits

   c) 1 sumador de 5 entradas de $(n+1) \times k$ bits

   d) 1 sumador de 2 entradas de k-bits

Tabla 4: Arquitecturas hardware representativas en el estado del arte para multiplicación en $\mathbb{GF}(p)$ usando el método Montgomery

Trabajo	$\log_2 p$	FPGA	Tiempo μs	Slices	F (MHz)	Enfoque
[Mondal et al., 2012]	256	XCHVHX250T	-	1572	69	64 DSP48E
[Mondal et al., 2012]	256	XCHVHX250T	-	2106	102	
[Mondal et al., 2012]	256	XCHVHX250T	-	2346	147	
[Hamilton et al., 2011]	127	Virtex5	0.80	264	161	64 DSP48E
[Hamilton et al., 2011]	512	Virtex5	9.68	957	54	
[Gong and Li, 2010]	256	EP3C40F324C6	0.098	23405 LES 81 mults.	30.38	multiplicadores embebidos. Usa internamente Karatsuba.
[Chow et al., 2010]	512	Virtex6	0.01	62557	300	Montgomery + Karatsuba 324 multiplicadores embebidos.
[Oksuzoglu and Savas, 2008]	1024	XC3S500E	7.62	1553	119	10 mults.
[Huang et al., 2008]	1024	Virtex2-6000	10.24	4178 (65 PEs)	100	Enfoque sistólico
[Huang et al., 2008]	1024	Virtex-II	69.61	2596	63	Iterativo, palabras de 32 bits, 4 mults.

e) 1 multiplicador de $k \times k$-bits

f) 1 inversor mod $\beta = 2^k$ de k-bits

2. Implementar el algoritmo completamente paralelo, requiriendo 1 iteración. En este caso la complejidad espacial es mucho mayor, ya que se requieren 3 multiplicadores de $log_2(M)$ bits, 1 para realizar $X \times Y$, 1 para realizar $(X \times Y) \times M'$ y uno mas para realizar $q_i \times M$. Otros enfoques usan un solo multiplicador y los reutilizan para realizar las tres multiplicaciones, pagando el precio de calcular la multiplicación Montgomery en mas de una iteración y usando multiplexores y registros temporales, lo que incrementa el camino crítico y el uso de área. Este es el caso del trabajo reportado en [Mondal et al., 2012, Gong and Li, 2010]. En estos trabajos, el objetivo es implementar el multiplicador con los multiplicadores embebidos en los FPGAs recientes.

   En la tabla 4 se muestran trabajos representativos en estado del arte de multiplicadores Montgomery en FPGA para el campo GF(p).

   De la tabla 4 se observa que los trabajos con menores requerimientos de área reportados son [Hamilton et al., 2011] para el FPGA Virtex5 y [Oksuzoglu and Savas, 2008] para el FPGA Spartan3E con una utilización de slices de 957 (operandos de 512 bits) y 1553 (operandos de 1020 bits), respectivamente. Sin embargo, el consumo de slices en [Hamilton et al., 2011] no considera los 64 DSP48E block que se utilizan en el FPGA para implementar multiplicadores de 48-bit. Lo

mismo ocurre en [Oksuzoglu and Savas, 2008], donde se usan 10 multiplicadores embebidos en el FPGA. Así, las metas para conseguir arquitecturas compactas del algoritmo Montgomery se fijan para obtener una utilización de slices por debajo de lo que han reportado [Hamilton et al., 2011] y [Oksuzoglu and Savas, 2008].

4 Una arquitectura hardware compacta del algoritmo Montgomery

Aunque una implementación paralela del método Montgomery resultaría ser la más rápida, es la que más recursos de área consume, lo que la hace inadecuada para aplicaciones en sistemas embebidos o de cómputo móvil. Es por ello que para contar con una arquitectura más compacta se tiene que implementar el método Montgomery desde un enfoque iterativo. Diversos algoritmos iterativos del método Montgomery se han propuesto en la literatura e implementado en hardware usando tanto tecnología ASIC como FPGA.

En el trabajo de Koc et al [Koç et al., 1996] se reportan diversos algoritmos para calcular una multiplicación usando el método Montgomery. Considere el algoritmo 3 listado en la sección 1 para calcular la multiplicación modulo M de dos números X, Y bajo el método de Montgomery. Se asume que los números X, Y, M están conformados por $n+1$ símbolos usando base $\beta = 2^k$, es decir:

$$M = (M_n \cdots M_0)_\beta \tag{34}$$

$$X = (X_n \cdots X_0)_\beta \tag{35}$$

$$Y = (Y_n \cdots Y_0)_\beta \tag{36}$$

El ciclo principal considera cada símbolo Y_i de Y, y calcula $X \times Y_i \bmod \beta^{-(n+1)} \bmod M$ en $n+1$ iteraciones. En cada iteración i es necesario calcular q_i y A_i, lo cual puede realizarse desde un enfoque secuencial mediante el siguiente conjunto de operaciones:

$$t_1 = X \times Y_i \tag{37}$$

$$t_2 = a_0 + X_0 \times Y_i \tag{38}$$

$$q_i = t_2 \times M' \bmod \beta \tag{39}$$

$$t_4 = M \times q_i = (r_n r_{n-1} \cdots r_1 r_0) \tag{40}$$

$$t_5 = A_i + t_1 = (s_n s_{n-1} \cdots s_1 s_0) \tag{41}$$

$$t_6 = t_5 + t_4 \tag{42}$$

$$A_{i+1} = t_6/\beta \tag{43}$$

Las multiplicaciones internas t_1, q_i y t_4 son multiplicaciones entre un número de k-bits y otro de $n * k$-bits. Note que t_2 es un número de $2k$-bits. Dado que q_i es el resultado de una operación módulo β, solo los k-bits menos significativos de t_2 se consideran para calcular q_i. Además M' es un valor de k-bits, constante durante toda la operación de multiplicación. Por tanto, q_i puede calcularse solo con un multiplicador de $k \times k$-bits. Por otra parte, se puede aplicar el método escolar de multiplicación (shift and add) para calcular iterativamente t_1 y t_4 en n iteraciones, procesando k-bits a la vez de los multiplicandos X y M.

Usando la representación polinomial de X en base β, la operación $X \times Y_i$ puede expresarse de la siguiente forma:

$$\begin{aligned} X \times Y_i &= \left(\sum_{k=0}^{n} \beta^k X_k \right) \times Y_i \\ X \times Y_i &= \sum_{k=0}^{n} \beta^k (X_k \times Y_i) \\ &= \beta^n (X_n \times Y_i) + \cdots + \beta (X_1 \times Y_i) + (X_0 \times Y_i) \end{aligned} \tag{44}$$

Cada término $(X_k \times Y_i)$ es de tamaño $2k$-bits. La suma acumulativa de la expresión $X \times Y_i$ se muestra gráficamente en la figura 2 (a), mientras que el circuito que implementa la sumatoria completa bajo el enfoque del multiplicador SHIFT and ADD se muestra en el la figura 2 (b).

Se requiere de un registro de corrimiento ZX de tamaño $(2k + k(n+1))$-bits, el cual puede verse como dos registros, la parte alta de $2k$-bits (Z) y otro de tamaño $k(n+1)$-bits (X). Aunque tradicionalmente en una multiplicación el número más pequeño se considera como el multiplicador, para reducir el numero de

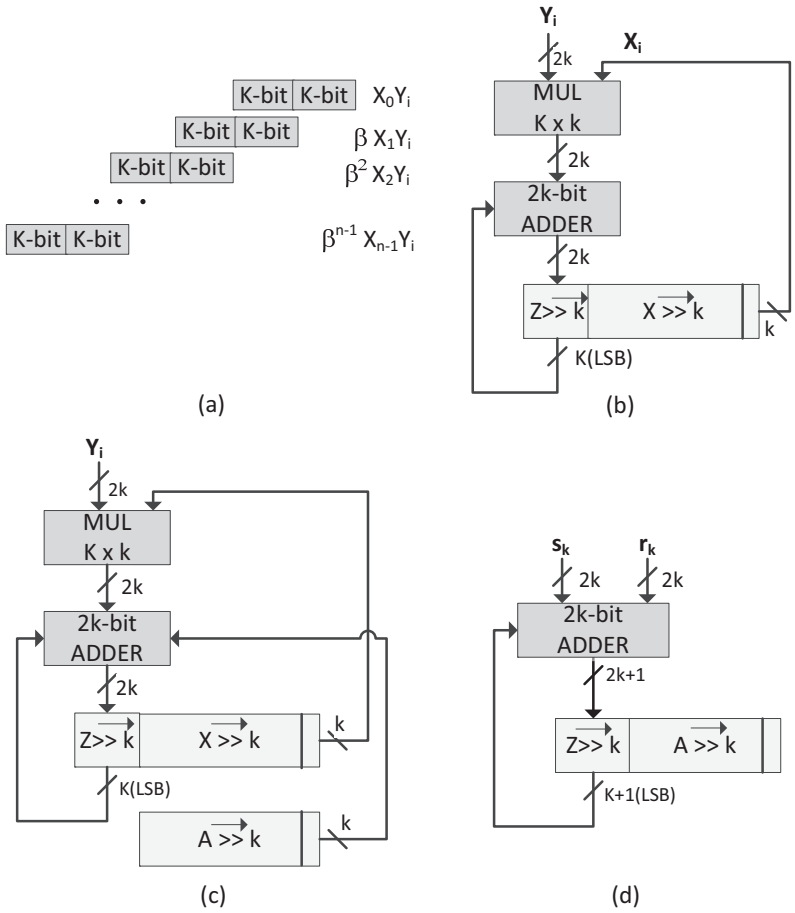

Figura 2: Implementación iterativa de la multiplicación $X \times Y_i$. (a) Ilustración de los productos parciales involucrados en la sumatoria. (b) Arquitectura hardware del multiplicador usando el método SHIFT and ADD. (c) Cálculo paralelo de t_1, t_2 y t_5. (d) Módulo hardware para calcular t_6.

términos en la sumatoria (y por tanto el número de sumas reduciendo el camino crítico de los sumadores), al tratarse en este caso de sumas acumulativas, consideramos el caso contrario, considerar al número más grande como el multiplicador, permitiendo así el uso de un circuito multiplicador más pequeño aunque con una penalización en la latencia. Inicialmente el multiplicador se carga en el registro X y el registro Z se inicializa en 0. Cada término $(X_k \times Y_i)$ de la sumatoria se calcula en el multiplicador de $k \times k$-bits y el valor es almacenado en los $2k$-bits más significativos del registro de corrimiento (regsitro Z). El registro ZX se recorre k-bits a la derecha en cada iteración, lo que produce dos cosas: la primera es que un nuevo término X_i esta disponible para realizar una nueva multiplicación (los k-bits menos significativos del registro X); la segunda es que con el corrimiento a la derecha, en el registro Z se deja solo la parte alta del resultado de la multiplicación previa, que debe sumarse con el resultado de la siguiente multiplicación, afín de que las sumas acumulativas parciales se realicen de manera correcta (ver figura 2 (a)). Es por ello que después de realizar el corrimiento sobre ZX, los k-bits menos significativos de Z se retroalimentan al sumador.

En lugar de esperar que todos los términos t_1 se calculen para calcular t_5, se pueden realizar ambos cálculos concurrentemente, ya que $A + X \times Y_i$ puede expresarse de la siguiente forma:

$$A + X \times Y_i = \sum_{k=0}^{n} \beta^k A_k + \left(\sum_{k=0}^{n} \beta^k X_k \right) \times Y_i \tag{45}$$

$$= \sum_{k=0}^{n} \beta^k A_k + \sum_{k=0}^{n-1} \beta^k (X_k \times Y_i) \tag{46}$$

$$= \sum_{k=0}^{n} \beta^k (A_k + (X_k \times Y_i)) \tag{47}$$

Se tendría que utilizar un registro de corrimiento de k-bits a la derecha para A, afín de procesar un dígito A_k a la vez. En cada iteración, el dígito A_k se debe sumar al resultado del multiplicador de $X_k \times Y_i$, junto con la parte baja del registro Z. Esto se consigue remplazando el sumador de dos operandos de la figura 2 (b) por uno de tres operandos. Así, la arquitectura hardware que se muestra en la figura 2 (c) calcula t_5 al tiempo que se calcula t_1, iterativamente, obteniendo un dígito s_k en cada iteración. Cada dígito s_k se puede sobre-escribir en el registro X.

En la arquitectura de la figura figura 2 (c), en la primera iteración, el valor t_2 esta disponible a la salida del sumador. Este valor puede almacenarse en un registro de k bits para usarse en el cálculo de q_i. Una

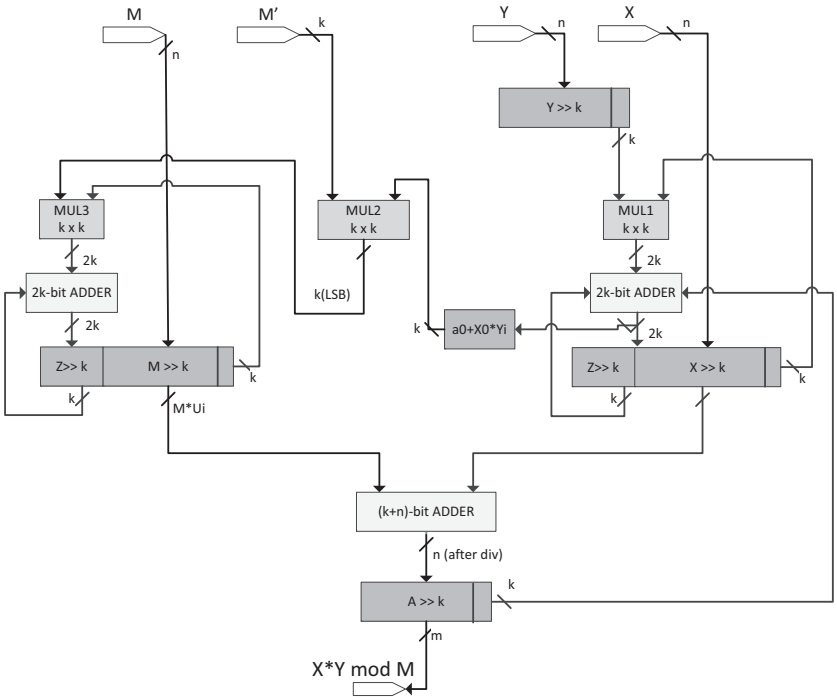

Figura 3: Arquitectura hardware para ejecución del algoritmo Montgomery iterativo: Primera propuesta

vez obtenido q_i, se puede usar nuevamente un multiplicador como el de la figura 2 (b) para calcular t_4, considerando como multiplicador a M_i y como multiplicando a q_i, calculando en cada iteración un dígito r_k. Dado que el calculo de t_1 y t_4 son independientes, pueden calcularse concurrentemente, estando retrasado el cálculo de r_k un ciclo de reloj respecto al cálculo de s_k. Se deberá realizar una sincronización en el cálculo de estos valores a fin de obtener la suma acumulativa $(s_k + r_k)$.

Después de $n+1$ iteraciones, los valores de t_4 y t_5 están calculados, y almacenados en los registros X y M por lo que sigue es calcular $t_6 = t_4 + t_5$. EL resultado final se divide por 2^{n+1}, lo que equivale a descartar los $n+1$ símbolos menos significativos de t_6 para obtener el resultado de la multiplicación bajo el método Montgomery.

En la figura 3 se muestra la primera propuesta de una arquitectura compacta para la ejecución del algoritmo Montgomery iterativo (algoritmo 3 de la sección 2).

Se pueden realizar diversas optimizaciones a la arquitectura presentada en la figura 3. No es necesario esperar $n+1$ ciclos para calcular t_6. Cada dígito r_k y s_k calculado en cada iteración permite realizar la suma $(s_k + r_k)$ y calcular t_6 iterativamente al mismo tiempo que se calculan t_1, t_4 y t_5. Esto se ilustra en el inciso

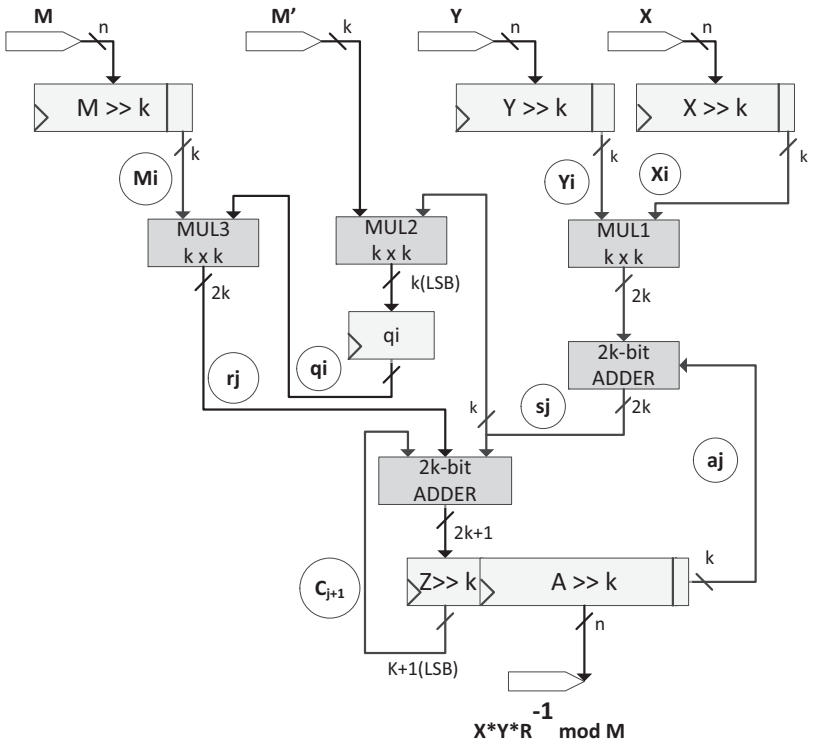

Figura 4: Una arquitectura hardware compacta para ejecutar el algoritmo Montgomery iterativo, procesando palabras de k-bits.

(d) de la figura 2. El resultado de la suma $(s_k + r_k)$ puede almacenarse en el mismo registro de corrimiento ZA. Al ir considerando cada uno de los términos r_k y s_k para calcular t_6 en cada iteración, no es necesario almacenar estos valores, los cuales se obtienen de los sumadores de tres entradas. Lo que si es necesario almacenar es la parte alta del resultado $(s_k + r_k)$, ya que esta parte se requiere en la suma acumulativa con el término en la siguiente iteración. Entonces, tomando en cuenta esta consideración, después de $n + 1$ iteraciones la variable t_6 es calculada y lo que resta es calcular el valor $A = t_6/\beta$. Esta operación e consigue con un corrimiento más a la derecha del registro ZA, quedando en el registro A el valor final A_{i+1}. Después de $n + 1$ iteraciones en el ciclo principal del algoritmo 3, la operación $X \times Y \times R^{-}1 \bmod M$ es finalmente calculada.

Con estas optimizaciones, una nueva arquitectura compacta del algoritmo Montgomery iterativo se muestra en la figura 4, la cual tiene el siguiente costo:

1. 3 registros de corrimiento de $log_2(M)$-bits
2. 1 registro de corrimiento de $log_2(M) + 2k$-bits
3. 3 multiplicadores de $k \times k$-bits
4. 2 sumadores de $2k$-bits
5. 1 sumador de k-bits

El algoritmo resultante para calcular una multiplicación Montgomery desde un enfoque iterativo se lista en el algoritmo 4.

5 Implementación y resultados

La arquitectura mostrada en la figura 4 se describío en VHDL y se sintetizó para dos dispositvos FPGA. El primero fue un FPGA Spartan3E, el cual es de baja capacidad y costo. El segundo FPGA fue un Virtex5, el cual incluye mucho más lógica reconfigurable y permite lograr diseños con frecuencias de reloj más altas. Este último FPGA es de los más recientes en el mercado. Se realizó una descripción hardware parametrizable, de tal forma que se pudieran evaluar diversas métricas para el multiplicador propuesto, en particular el área ocupada expresada en slices del FPGA, la frecuencia de operación, medida en millones de

Algorithm 4 Iterative Montgomery algorithm

Entrada: integers $X = \sum_{i=0}^{n} X_i\beta^i$, $Y = \sum_{i=0}^{n} Y_i\beta^i$ and $M = \sum_{i=0}^{n} M_i\beta^i$, with $0 < X, Y < 2 * M$, $R = \beta^{n+1}$ with gcd$(M, \beta) = 1$, and $M' = -M^{-1} \bmod \beta$
Salida: $A = \sum_{i=0}^{n} a_i\beta^i = X \times Y \times R^{-1} \bmod M$

```
1: A ← 0
2: for i ← 0 to n do
3:    c_0 ← 0
4:    for j ← 0 to n do
5:       s_j ← [a_0 + X_j × Y_i] /*a_0 takes each a_j value */
6:       if j = 0 then
7:          q_i ← (s_j × M') mód β
8:       end if
9:       r_j ← q_i × M_j
10:      {c_{j+1}, t6_j} ← s_j + r_j + c_j
11:      A ← SHR(A)
12:      a_n ← t6_j
13:   end for
14:   A ← SHR(A) /*A ← A/β */
      a_n ← c_{n+1}
15: end for
16: return A
```

ciclos por segundo (MHz), el tiempo para calcular una multiplicación Montgomery en milisegundos (μs), el throughput, expresado como la cantidad de bits por segundo (Mbps) (o dicho de otra forma, el número de multiplicaciones por segundo) y la eficiencia, la cual se expresa como la cantidad de bits por segundo por slice.

La implementación del multiplicador compacto Montgomery se realizó en base a tres enfoques:

1. *Arquitectura 1 (CSR)* - La memoria requerida para almacenar los valores de X, Y, y M se implementaron como registros de corrimiento circular usando los recursos lógicos del FPGA (no se utilizaron los bloques de memoria BRAM incluidos en el dispositivo). Al hacer esto, en cada iteración i no es necesario recargar nuevamente los valores en los registros mencionados. El registro A se implementó como un registro de corrimiento a la derecha usando como valor de relleno la parte baja de la suma $(s_k + r_k)$, como se muestra en la figura 4. Los multiplicadores de $k \times k$ bits se implementaron usando los multiplicadores MULT18x18 incluidos en el FPGA Spartan3E y los bloques DSP48E incluidos en el FPGA Virtex5.

2. *Arquitectura 2 (BRAM)* - En un segundo enfoque, la memoria requerida para almacenar los valores de X, Y, y M se implementaron usando los bloques de memoria BRAM incluidos en el dispositivo FPGA. Esto redujo considerablemente el uso de recursos hardware, ya que la mayor parte de lógica era usada para implementar los registros de corrimiento circular. El registro A y los multiplicadores se implementaron como en la Arquitectura 1.

3. *Arquitectura 3 (2MULs)* - En un tercer enfoque, se decidío describir la arquitecura del multiplicador usando solo dos multiplicadores de $k \times k$ bits. Debido a que el multiplicador `MUL2` solo se usa una vez en cada iteración i del algoritmo 4 para clacular q_i, este mismo multiplicador se puede utilzar para calcular r_j (y eliminar `MUL3`). El costo adicional es agregar multiplexores y lógica de control a fin de multiplexar en el instante correcto las entradas al multiplicador. Las memorias para los registros X, Y, M y A se implementaron como en la Arquitectura 2, al igual que los multiplicadores requeridos.

A fin de analizar los resultados de implementación considerando diferentes configuraciones en cuanto al tamaño de los operandos y la base usada, se implementó un *generador de código VHDL* en Java para cada una de las 3 arquitecturas desarrolladas. Para la experimentación se consideraron los tamaños de operando {256, 512, 1024} y para la base $\beta = 2^k$ se consideraron los valores $k \in \{1, 2, 4, 8, 16, 32\}$, los cuales son valores típicamente utilizados en el estado del arte y en implementaciones prácticas del algoritmo Montgomery en aplicaciones criptográficas.

La síntesis se realizó usando el software ISE 14.2, el cual es el más reciente liberado por la empresa Xilinx, fabricante de los FPGA usados para implementación en esta investigación. A pesar de que la herramienta provee una interfaz gráfica de usuario, dado que la combinación de todas las arquitecturas a evaluar es muy elavada, a saber 3*3*6 = 54, se decidió realizar la implementación usando archivos script desde la línea de comandos. Para la síntesis, se consideraron los dos criterios de optimización, por velocidad y por área. Se escribió en Java un *generador de archivos script* que permitieran sintetizar los 54 archivos VHDL considerando estas restricciones de síntesis. De igual forma, se escribio en Java un *extractor de resultados*, que permitiera automáticamente extraer de los archivos de reporte de implementación generados por ISE los valores de interés, como son el número de slices utilizados, el número de LUTs, el número de flip-flops y la frecuencia de operación.

En la figura 5 se muestran los resultados de implementación del multiplicador Montgomery en el FPGA Spartan3E xc3s500e. A pesar de que esta investigación tiene como principal objetivo la obtención

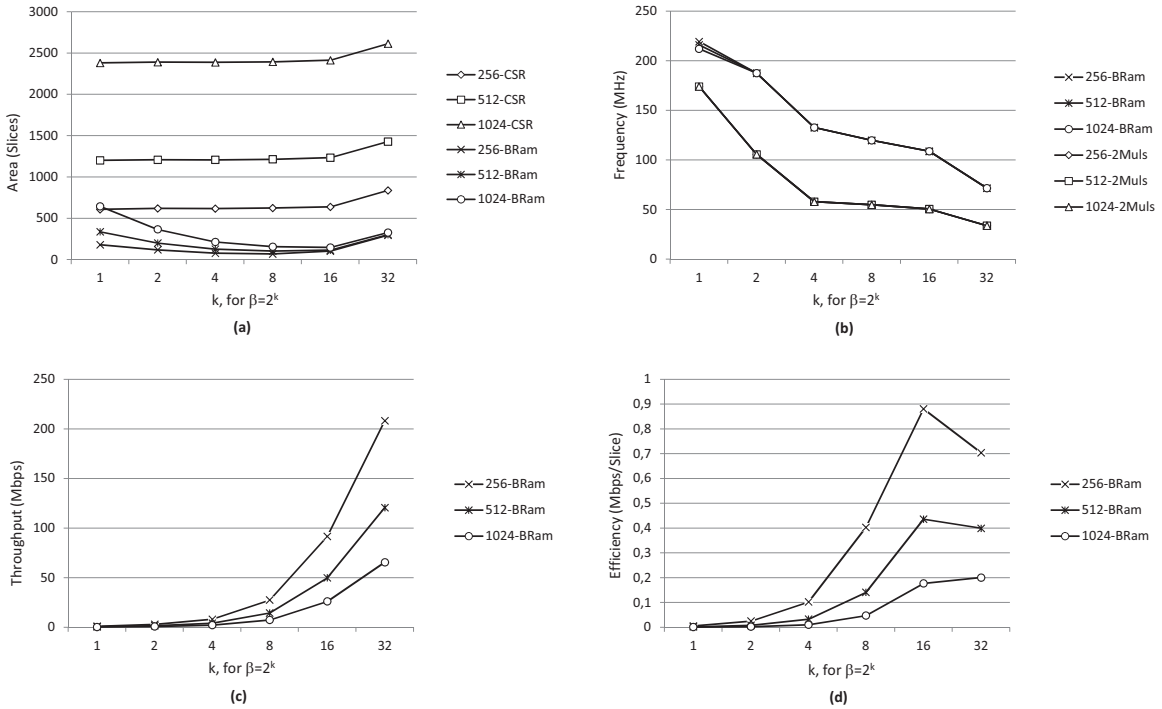

Figura 5: Resultados de implementación del multiplicador Motgomery compacto en un FPGA Spartan3E

de arquitecturas hardware compactas para el algorimto Montgomery, en los resultados obtenidos se observó que una optimización por área obtenía muy poca reducción de área comparada con la optimización por velocidad, la cual fue de alrededor del 0.2 %. Por el contrario, una optimización de implementación por velocidad presentaba una mejora más evidente en la frecuencia de operación obtenida, y por ende en el performance del multiplicador. Bajo el criterio de optimización por velocidad las frecuencias de operación mejoraron en un 5 % respecto a una optimización por área.

En la figura 5 (a) se observa la elevada cantidad de área que consume la Arquitectura 1 al implementar los registros de corrimiento circular, la cual se incrementa linealmente de acuerdo al tamaño del operando. En contraste, la Arquitectura 2 presenta una reducción considerable de slices al usar los bloques BRAM del dispositvo en lugar de implementar los registros de corrimeinto circular. Además, cabe destacar que para la Arquitectura 2, un incremento en área es significativa solo cuando se cambia la base de representación, más no cuando el tamaño del campo se cambia. Esto se verifica al ver que las curvas que representan la utilización de slices para la Arquitectura 2 se encuentran muy cerca unas de otras. En los resultados obtenidos, los recursos de área para la Arquitectura 3 son muy parecidos a los de la Arquitectura 2, por lo

que no se muestran en la figura 5 (a). Para la Arquitectura 3, el ahorro en área solo se percibe para el caso de operandos de tamaño 1024, donde el área se reduce de 0.3 % a 13 % respecto a la Arquitectura 2.

La figura 5 (b) muestra los resultados obtenidos respecto a la frecuencia de operación, una métrica muy importante ya que impacta el performance del multiplicador. Como se puede observar, la Arquitectura 3 presenta una significativa reducción en la frecuencia de operación comparada con la Arquitectura 2, por lo que el ahorro que se puede tener en área al utilizar solo dos multiplicadores (la cual es mínima como se comentó anteriormente) no se justifica. Las frecuencias de operación de la Arquitectura 1 son muy similares a los de la Arquitectura 2, por lo que éstas no se muestran en la figura 5 (b).

De acuerdo a los resultados obtenidos en cuanto a área y frecuencia de operación, podemos concluir que la mejor estrategia de implementación del algoritmo Montgomery es la Arquitectura 2. Para esta arqutiectura se midieron tanto el rendimiento expresado en Mbps y la eficiencia expresada como Mbps/Slice. En la figura 5 (c) se puede observar que a medida que la base de representación se incrementa (y por ende, un digito más grande se procesa a la vez) más alto es el rendimiento obtenido. En particular, para $\beta = 2^{16}$ y $\beta = 2^{32}$ se obtienen los desempeños más altos, alrededor o más de 50Mbps. Sin embargo, usar tamaños de base más grande implica mayor costo en slices. La eficiencia permite evaluar que tan justificado es el precio de área de acuerdo al desempeño que se obtiene por la arquitectura. En la figura 5 (d) se puede observar que con una base $\beta = 2^{16}$ se obtiene la mejor eficiencia.

Debido a que en la literatura el principal objetivo ha sido implementar el algoritmo Montgomery lo más rápido posible, escasos trabajos se enfocan en conseguir implementaciones compactas. El trabajo donde se ha reportado la implementación más compacta del algoritmo Montgomery es en [Oksuzoglu and Savas, 2008]. En este trabajo se implementa un algoritmo Montgomery iterativo usando un enfoque de implementación basado en arreglos sistólicos de elementos de procesamiento. En la tabla 5 se muestra una comparación de los resultados obtenidos en esta investigación contra los reportados en [Oksuzoglu and Savas, 2008]. Se considera una comparación justa, ya que se utiliza el mismo dispositivo FPGA y un tamaño de operando muy parecido. Comparado con la implementación en [Oksuzoglu and Savas, 2008], el multiplicador Montgomery desarrollado en esta investigación es 3.8 mas pequeño, y aunque es dos veces más lento, presenta una mejor eficiencia, procesando el doble de bits por segundo por slice.

En la figura 6 se muestran resultados de implementación de la Arquitectura 2 del multiplicador Montgomery propuesto en un FPGA Virtex5 xc5vlx50. Los resultados, aunque mejores, se deben a que

Tabla 5: Comparación del multiplicador Montgomery propuesto contra la implementación más compacta reportada en la literatura, usando el mismo FPGA, un Xilinx xc3s500e.

Ref.	Tam. operando	Mults.	BRams	Slices	Tiempo μs	Throughput (Mbps)	Eficiencia Mbps/Slices
[Oksuzoglu and Savas, 2008]	1020	10	4	1553	7.62	133.8	0.086
This work $k = 16$	1024	3	3	147	39.41	25.98	0.176
This work $k = 32$	1024	3	3	327	15.63	65.48	0.20

el FPGA Virtex5 incluye más lógica por bloque lógico configurable (CLB). Un slice en un FPGA Virtex5 tiene el doble de LUTs y Flip-flops que un slice en el FPGA Spartan3E. Además, las LUTS en un FPGA Virtex5 son de 6 entradas en comparación con las LUTs de un Spartan3E, las cuales son de 4 entradas. El multiplicador obtiene mejores frecuencias de operación, lo que se refleja en un mejor desempeño. En cuanto a recursos de área, en la figura 6 (a) se observa que el multiplicador Montgomery mantiene un consumo de LUTs muy similar independientemente del tamaño de operando utilizado, y que prácticamente para todas las bases que se probaron, excepto para $\beta = 2^{32}$, el consumo de LUTs se mantiene entre 50 y 100 LUTs. Las gráficas en la figura 6 (b) revelan que para la base 2^8 es para la que se obtiene la menor cantidad de de flip-flops. Al igual que en los resultados de implementación para el FPGA Spartan3E, la mejor eficiencia parace alcanzarse para una base igual a 2^{16}.

En la tabla 6 se presenta una comparación de los resultados obtenidos en esta investigación contra trabajos representativos en estado del arte, tanto implementaciones software [Tenca and Çetin Kaya Koç, 2003, Brown et al., 2001, Itoh et al., 1999] como implementaciones en FPGA [Huang et al., 2011, Mondal et al., 2012, Hamilton et al., 2011, Gong and Li, 2010]. Aunque una comparación de este tipo no es justa, dado las diferentes plataformas de implementación y capacidades de cada una de ellas, esta comparación se provee solo como una referencia. La comparación contra las implementaciones en software es con el objetivo de justificar la motivación de diseñar arquitecturas dedicadas para la ejecución del algoritmo Montgomery. Como se observa en los resultados de la tabla 6, el multiplicador propuesto supera los tiempos de ejecución del algoritmo Montgomery sobre procesadores de propósito general como el ARM, Pentium-II o el DSPTMS320C6201. Estos dispositivos se encuentran comúnmente en sistemas embebidos o de cómputo móvil. La comparación contra las arquitecturas hardware en FPGA son con el objetivo de contrastar las diferencias en los recursos utilizados. De los resultados en la tabla 6, es de resaltar diseños como [Huang et al., 2008, Huang et al., 2011], que aunque pueden realizar una multiplicación en tiempo mucho menor, los recursos de área son bastante elevados, haciendo inviable la utilización de esas

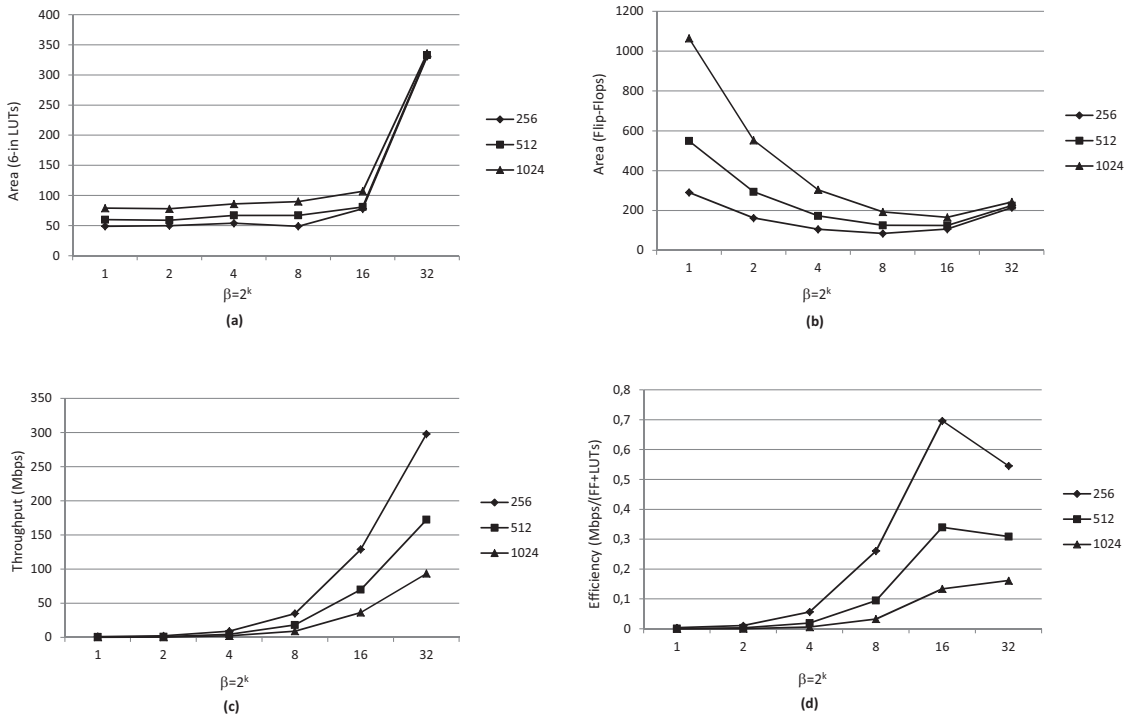

Figura 6: Resultados de implementación del multiplicador Montgomery (Arquitectura 2) propuesto en el FPGA Virtex5.

Tabla 6: Comparación del multiplicador Montgomery propuesto (Arquitectura 2) contra implementaciones software y en FPGAs.

Ref.	Tam. operando	Plataforma	Slices	Frec. (MHz)	Tiempo μs
[Tenca and Çetin Kaya Koç, 2003]	1024	ARM	-	80	570
[Tenca and Çetin Kaya Koç, 2003]	256	ARM	-	80	42.3
[Brown et al., 2001]	256	Pentium-II	-	400	1.57
[Itoh et al., 1999]	256	DSPTMS320C6201	-	200	2.68
Fig. 3 Arch.2 $k = 32$	256	XC3S500e	296	71.61	1.22
Fig. 3 Arch.2 $k = 32$	512	XC3S500e	302	71.61	4.24
Fig. 3 Arch.2 $k = 32$	1024	XC3S500e	327	71.61	15.63
Fig. 3 Arch.2 $k = 16$	256	xc5vlx50	107FF/78LUTs	153.109	1.98
Fig. 3 Arch.2 $k = 16$	512	xc5vlx50	125FF/81LUTs	153.109	7.31
Fig. 3 Arch.2 $k = 32$	1024	xc5vlx50	243FF/336LUTs	102.42	10.93
[Hamilton et al., 2011]	521	xc5vlx50	957	54	9.64
[Huang et al., 2011] $k = 2$	1024	Virtex-II6000	5,356 LUTs	106.4	0.56
[Mondal et al., 2012]	256	XCHVHX250T	1572 64DSP48E	69.44	0.014
[Gong and Li, 2010]	256	Cyclone3	23405 LEs 81Mults	30.38	0.1
[Huang et al., 2008]	1024	Virtex-II	4178 65PEs	100	10.88

soluciones en aplicaciones de sistemas embebidos o de cómputo móvil.

6 Conclusiones

La multiplicación modular mediante el método Montgomery es de gran interés cuando el número de multiplicaciones con reducción modular es muy grande, lo que ocurre en esquemas de cifrado de llave pública o en criptografía basada en emparejamientos. Aunque en la literatura existen implementaciones del algoritmo en hardware, acelerando el cálculo de una multiplicación Montgomery por varios órdenes de magnitud, la mayoría de estas implementaciones se enfocan en realizar la operación de multiplicación modular lo más rápido posible, lo que ha llevado a diseños que usan una gran cantidad de recursos de área, que en el caso de implementaciones sobre FPGAs, esto se traduce en una gran cantidad de slices.

En este trabajo se presentó el diseño de un multiplicador Montgomery que usa pocos recursos del FPGA, a costa de un incremento en la latencia. En muchas aplicaciones en sistemas embebidos o de cómputo móvil, es aceptable sacrificar el tiempo de procesamiento a fin de obtener diseños compactos que requieran poca energía y disipen poco calor. A partir de los resultados experimentales, se concluye que la mayor

parte de la lógica del FPGA cuando se implementa el algoritmo Montgomery es la memoria para almacenar los operandos involucrados, a decir, el multiplicador, multiplicando y módulo. Al hacer uso adecuado de los recursos de FPGA, usando los bloques de memoria que éstos incluyen, así como los multiplicadores embebidos con los que cuenta, se consiguen diseños compactos, que usan mucho menores slices que los diseños más compactos reportados a la fecha. Además, aunque el algoritmo Montgomery iterativo puede procesar dígitos de cualquier tamaño, los experimentos realizados demuestran que para específicos tamaños de base se obtiene la implementación más compacta, el throughput más alto o la mejor eficiencia. La cantidad mínima de área se consigue usando una base $\beta = 2^8$, el mejor throughput se consigue con una base $\beta = 2^{32}$ y la mejor eficiencia se consigue con una base $\beta = 2^{16}$.

El multiplicador Montgomery desarrollado en esta investigación puede servir para crear un módulo para exponenciación con reducción modular, el cual es típicamente usado en esquemas de cifrado RSA, intercambio de llaves usando el protocolo Diffie-Hellman y firmas digitales con el estándar DSS. Este módulo de exponenciación puede ser incluido en un sistema embebido o móvil.

.1 Vectores de prueba

Tamaño de operandos de 1024 bits (en notación hexadecimal)							
M =	6e79434fd91645c886aa3169c908791865815db7fb2						
	104ac291c321351c14fbaf746f25c920bfca40c808b						
	0c4cbfbf8f6c2fd11a2abfda32b0d8758135775a552						
	e70620f92648e4707d5c4bcbe77c01a8f9bbe410dc1						
	68338cb4cb1d457aeee35e144098393aa78a8776e06						
	bb7152560eab8ce7e4a985b41c8695a64f3805605						
Y =	18e75f311ce8a8459d7c39b2298c0f7a3bccfa6142a						
	c740cbe9b6d95d4732066dfe9f62997e0954034503d						
	7ea784fa332d5dc8fd555e3de4fe4bfe44f7daebc03						
	eded23c5b911a29e89bcabc093a3ae0a1aa3ecddc08						
	e55d558c845bff1f01503fc1f56a3a18ea38352d38f						
	857a51cd8b321322ffd533f639b02a6014aa6f5ef						
X =	29df4a5222a76d0c2a046902131ea045d32593bf6ab						
	39c0090a8872bd4f623f5f40bd541bfe1d5b2a66fa0						
	a40d2632f299937e8e1942ba2673b3e0c459bc250b9						
	316ccbf2f571c7407bf7dbfe5922740399974950d83						
	9f6ab072ce177f8ce2946bd9387c37a9f402ede9d00						
	a81e5efe4b860ddc20034fa611660c205c209bb0f						
	k = 1	k = 2	k = 4	k = 8	k = 16	k = 32	k = 64
M' =	1	3	3	33	f933	89f4f933	71591bbd89f4f933

Bibliografía

[Brown et al., 2001] Brown, M., Hankerson, D., López, J., and Menezes, A. (2001). Software implementation of the nist elliptic curves over prime fields. In *Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA*, CT-RSA 2001, pages 250–265, London, UK, UK. Springer-Verlag.

[Chow et al., 2010] Chow, G. C. T., Eguro, K., Luk, W., and Leong, P. (2010). A Karatsuba-based Montgomery multiplier. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 434–437, Washington, DC, USA. IEEE Computer Society.

[Gong and Li, 2010] Gong, Y. and Li, S. (2010). High-throughput FPGA implementation of 256-bit Montgomery modular multiplier. In *IEEE on Second International Workshop on Education Technology and Computer Science*, pages 173 –177.

[Hamilton et al., 2011] Hamilton, M., Marnane, W., and Tisserand, A. (2011). A comparison on FPGA of modular multipliers suitable for elliptic curve cryptography over GF(p) for specific p values. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 273 –276.

[Huang et al., 2011] Huang, M., Gaj, K., and El-Ghazawi, T. A. (2011). New hardware architectures for Montgomery modular multiplication algorithm. *IEEE Trans. Computers*, 60(7):923–936.

[Huang et al., 2008] Huang, M., Gaj, K., Kwon, S., and El-Ghazawi, T. (2008). An optimized hardware architecture for the Montgomery multiplication algorithm. In *Proceedings of the Practice and theory in public key cryptography, 11th international conference on Public key cryptography*, PKC'08, pages 214–228, Berlin, Heidelberg. Springer-Verlag.

[Itoh et al., 1999] Itoh, K., Takenaka, M., Torii, N., Temma, S., and Kurihara, Y. (1999). Fast implementation of public-key cryptography ona DSP tms320c6201. In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '99, pages 61–72, London, UK, UK. Springer-Verlag.

[Knezevic et al., 2010] Knezevic, M., Vercauteren, F., and Verbauwhede, I. (2010). Faster interleaved modular multiplication based on Barrett and Montgomery reduction methods. *IEEE Trans. Comput.*, 59(12):1715–1721.

[Koç et al., 1996] Koç, C. K., Acar, T., and Kaliski, Jr., B. S. (1996). Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33.

[Mentens et al., 2007] Mentens, N., Sakiyama, K., Preneel, B., and Verbauwhede, I. (2007). Efficient pipelining for modular multiplication architectures in prime fields. In *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, GLSVLSI '07, pages 534–539, New York, NY, USA. ACM.

[Mondal et al., 2012] Mondal, A., Ghosh, S., Das, A., and Chowdhury, D. R. (2012). Efficient FPGA implementation of Montgomery multiplier using DSP blocks. In *Proceedings of the 16th international conference on Progress in VLSI Design and Test*, VDAT'12, pages 370–372, Berlin, Heidelberg. Springer-Verlag.

[Montgomery, 1985] Montgomery, P. L. (1985). Modular multiplication without trial division. *Math. Computation*, 44:519–521.

[Oksuzoglu and Savas, 2008] Oksuzoglu, E. and Savas, E. (2008). Parametric, secure and compact implementation of RSA on FPGA. In *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*, pages 391 –396.

[Tenca and Çetin Kaya Koç, 2003] Tenca, A. F. and Çetin Kaya Koç (2003). A scalable architecture for modular multiplication based on Montgomery's algorithm. *IEEE Trans. Computers*, 52(9):1215–1221.