IMPLEMENTACIÓN DE LA FUNCIÓN POTENCIA MODULAR EN HARDWARE REPROGRAMABLE.

Freddy. Bolaños, Rubén. Nieto, Álvaro. Bernal.

Escuela de Ingeniería Eléctrica y Electrónica.
Grupo de Arquitecturas Digitales y Microelectrónica.
Universidad del Valle.
Carrera 100 No. 13-00, Cali- Colombia.

fremar@yubarta.univalle.edu.co,
alvaro@mafalda.univalle.edu.co,
rnieto@eiee.univalle.edu.co.

ABSTRACT

Modern cryptosystems execute operations using big number representation. For that reason, researching and looking for efficient architectures becomes mandatory. Programmable devices allow to compare different kind of architectures more easily in order to evaluate their performance. This paper shows some results of the exponentiation function implemented using programmable logic devices. Besides, taking advantage of the flexibility of such devices, a performance comparison between several designs alternatives have been made.

RESUMEN

El cálculo eficiente de la función potencia modular para operandos de gran tamaño es una característica importante en muchos de los sistema criptográficos modernos. Tradicionalmente, se ha optado por la implementación de estas operaciones en software, pero la aparición de los lenguajes de descripción de hardware y de los dispositivos lógicos reprogramables abren la posibilidad de llevarlas a hardware de una manera más flexible. Este artículo muestra los resultados de la implementación de la función potencia modular usando un dispositivo lógico programable. Se presentan además comparaciones de desempeño entre diferentes alternativas de diseño, que pudieron ser implementadas gracias a la gran flexibilidad que ofrecen estos dispositivos.

IMPLEMENTACIÓN DE LA FUNCIÓN POTENCIA MODULAR EN HARDWARE REPROGRAMABLE.

Freddy Bolaños, Rubén Nieto, Álvaro. Bernal.

Escuela de Ingeniería Eléctrica y Electrónica.
Grupo de Arquitecturas Digitales y Microelectrónica.
Universidad del Valle.
Carrera 100 No. 13-00, Cali- Colombia.

fremar@yubarta.univalle.edu.co, alvaro@mafalda.univalle.edu.co,
rnieto@eiee.univalle.edu.co.

RESUMEN

Este artículo muestra los resultados de la implementación de la función potencia modular usando un dispositivo lógico programable. Se presentan además comparaciones de desempeño entre diferentes alternativas de diseño, que pudieron ser implementadas gracias a la gran flexibilidad que ofrecen estos dispositivos.

ABSTRACT

This paper shows some results of the exponentiation function implemented using programmable logic devices. Besides, taking advantage of the flexibility of such devices, a performance comparison between several designs alternatives have been made.

1. INTRODUCCIÓN

La función potencia modular es una operación indispensable en muchas de las técnicas criptográficas modernas [1,2]. En gran parte de estos protocolos, el nivel de seguridad de todo el sistema depende del tamaño de los operandos usados en las operaciones criptográficas.

Consecuentemente, el esfuerzo de los desarrolladores de tales sistemas está encaminado a lograr que este tipo de operaciones sean hechas en el menor tiempo posible. Si bien una implementación hardware es inherentemente más rápida que una en software, la primera alternativa solo ha comenzado a ser considerada en los últimos años, gracias a la aparición de dispositivos lógicos programables, que ofrecen mucha más flexibilidad en el diseño y la depuración de una aplicación.

Con la aparición de los dispositivos lógicos programables de alta densidad, y de herramientas de diseño como los lenguajes de descripción de hardware, el diseño de aplicaciones hardware se ha convertido en un proceso rápido, flexible y barato.

Se ha usado el VHDL como lenguaje de descripción de hardware y un dispositivo lógico programable complejo (CPLD) de Altera Corp para evaluar el desempeño del hardware para ejecutar la función exponencial de números representados en notación modular. Se ha diseñado un prototipo que opera números de 32 bits a fin de evaluar el desempeño y se discute la expansión de tal diseño a números enteros de mayor tamaño.

2.LA FUNCIÓN EXPONECIACIÓN MODULAR

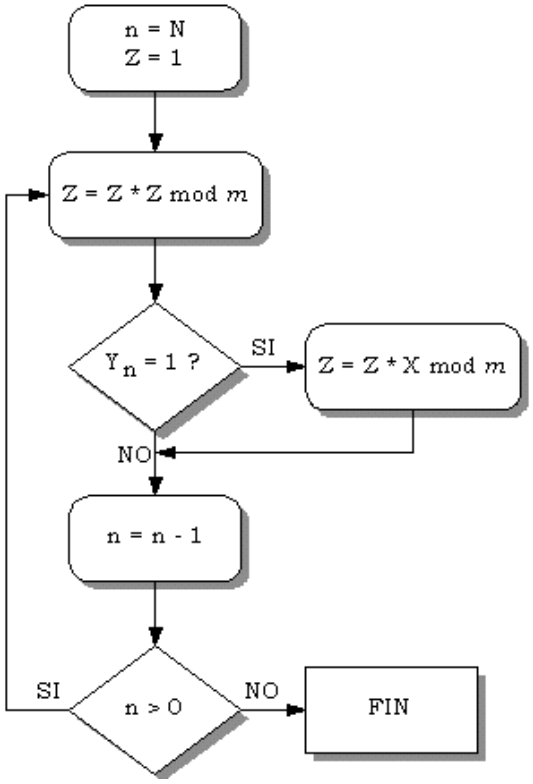
Se define la función potencia modular de acuerdo a la siguiente ecuación:

$$E = X^Y \bmod M. \qquad (1)$$

Donde E es el resultado de la exponenciación modular, X es la base de la potencia, Y es el exponente y M es el módulo. La operación potencia modular descrita en la ecuación (1) es el equivalente matemático a calcular X^Y, para luego dividir el resultado obtenido por M y tomar el residuo.

Sin embargo, calcular una potencia modular de esta forma implica varios problemas en una implementación práctica. El más grave de ellos es el problema del almacenamiento. Si los números X e Y son lo suficientemente grandes (cosa que es común en sistemas criptográficos), es posible que los requerimientos de almacenamiento del resultado intermedio X^Y sean

imposibles de cumplir en un sistema práctico. Se han propuesto varios algoritmos para un cálculo más eficiente de la función potencia modular. Entre ellos se encuentra el algoritmo binario [3], el cual se presenta en su versión de derecha a izquierda a continuación.



Pseudocódigo

```
n = N
Z = 1
Mientras (n>0)
      Z = Z · Z mod m
      Si (Y_n = 1) entonces
            Z = Z · X mod m
      Fin Si
      n = n - 1
Fin Mientras
```

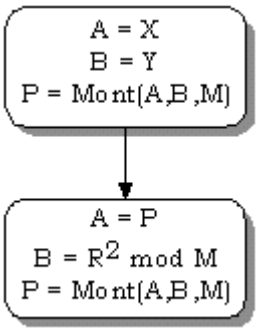
Como se puede ver, la ejecución de la exponenciación modular no es más que la ejecución iterativa y condicionada de productos modulares. El producto o multiplicación modular está definido por la ecuación (2).

$$P = XY \bmod M \qquad (2)$$

Donde P es el producto modular, X e Y son los multiplicandos y M es el módulo de la operación. El producto modular es el equivalente matemático de multiplicar X e Y, luego dividir el resultado por M y tomar el residuo de la división. A pesar de su sencillez, esta forma de calcular el producto modular es poco usada en aplicaciones prácticas.

Para la implementación tratada en este artículo se usó el algoritmo de multiplicación modular de Montgomery [4], que es especialmente eficiente cuando deben calcularse una cantidad considerable de productos, como es el caso del algoritmo presentado.

El algoritmo de multiplicación de Montgomery está basado en un nuevo operador llamado *Multiplicador de Montgomery*. A continuación se muestra la forma de calcular el producto modular: $\mathbf{P}=\mathbf{X}\cdot\mathbf{Y} \bmod \mathbf{M}$, usando el multiplicador de Montgomery.



Pseudocódigo

```
P = Mont(X,Y,M)
P = Mont(P,R^2 mod M,M)
```

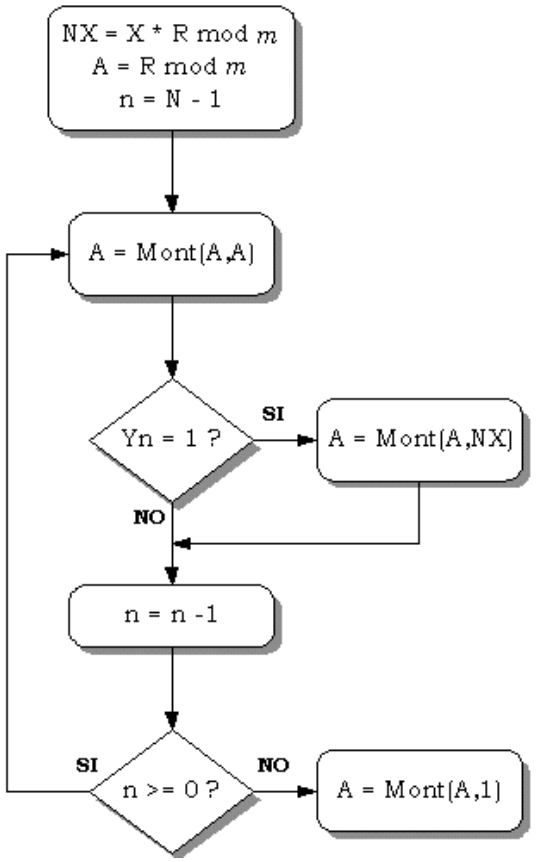
En la figura, el operador de Montgomery está denotado por *Mont(A,B,M)*, donde A y B son los multiplicandos y M es el módulo de la operación. El valor de $R^2 \bmod M$ es una constante que depende del tamaño en bits de los operandos X e Y. Si el tamaño de los operandos es N bits, se dice que $R = 2^N$ y que $R^2 \bmod M = RR \bmod M$. Para poder usar este algoritmo de existir el inverso multiplicativo modular de R, es decir, un número $I = R^{-1} \bmod M$ tal que:

$$R \cdot I \bmod M = 1 \qquad (3)$$

Se deduce entonces que el cálculo de una potencia modular requerirá como máximo 2N productos modulares, donde N es el tamaño en bits del exponente. Cada uno de estos productos modulares requiere a su vez dos ejecuciones sucesivas del operador *Mont* y en la segunda de estas ejecuciones se multiplica siempre por la misma constante $R^2 \bmod M$.

El algoritmo de exponenciación de Montgomery [5] elimina la segunda ejecución del operador *Mont* incluyendo la constante en la primera ejecución del

mismo. Tal algoritmo se presenta en su versión de derecha a izquierda a continuación.



Pseudocódigo

NX = X · R mod m
A = R mod m
n = N-1
Mientras (n >= 0)
    A = Mont(A,A)
    Si (Y_n = 1) **entonces**
        A = Mont(A,NX)
    Fin Si
    n = n – 1
Fin Mientras
A = Mont(A,1)

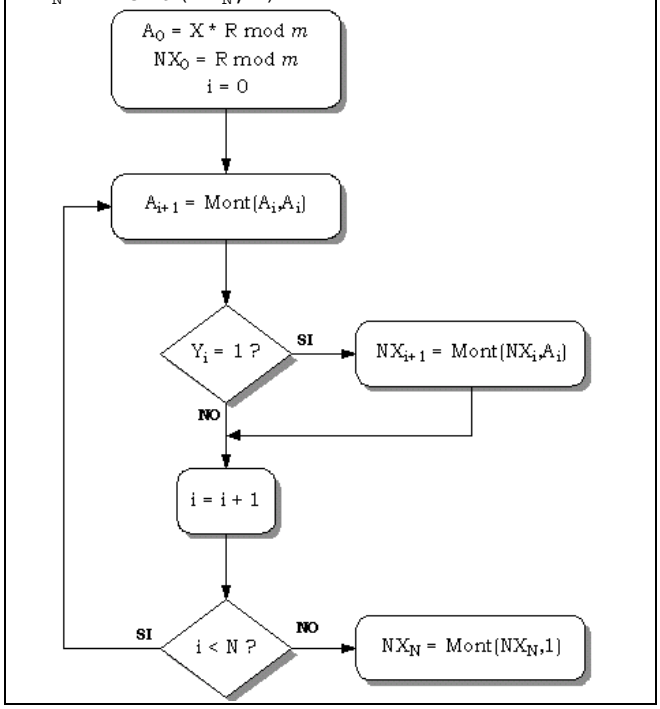
En promedio, este algoritmo requerirá alrededor de la mitad del tiempo para el cálculo de una potencia modular dada, si en ambos algoritmos se usa el mismo operador *Mont*.

En la versión derecha a izquierda del algoritmo puede verse que la ejecución de cada operación *Mont* debe hacerse en forma secuencial, es decir, para cada iteración el resultado del primer producto *Mont* es un operando para el segundo producto.

Esta situación ha sido corregida en el algoritmo de exponenciación de Montgomery en su versión izquierda–derecha, en donde las dos operaciones *Mont* pueden hacerse en forma concurrente. Tal algoritmo se presenta enseguida.

Pseudocódigo

A_0 = X · R mod m
NX_0 = R mod m
i = 0
Mientras (i < N)
    A_{i+1} = Mont(A_i, A_i)
    Si (Y_i = 1) **entonces**
        NX_{i+1} = Mont(NX_i,A_i)
    Fin Si
    i = i + 1
Fin Mientras
NX_N = Mont(NX_N,1)



Es posible entonces deducir que el tiempo máximo de ejecución para el algoritmo de exponenciación de Montgomery en su versión derecha-izquierda es igual a 2N veces el tiempo de ejecución máximo de una operación *Mont*. Por otra parte, el tiempo máximo de ejecución para la versión izquierda-derecha puede llegar a ser N veces el tiempo de ejecución de la operación *Mont*, siempre y cuando tales operaciones se realicen en forma concurrente.

Considerando todo esto se decidió implementar los algoritmos descritos en el dispositivo lógico EPF10K30AAQC208-1 de Altera Corp, para luego

comparar su desempeño en términos de velocidad de ejecución y área del dispositivo ocupada.

3. DETALLES DE LA IMPLEMENTACIÓN

Los lineamientos para el diseño del operador *Mont* estuvieron orientados al cálculo rápido de una cantidad considerable de productos modulares. Los diseños de este y otros elementos complejos de la implementación se realizaron a nivel de transferencia entre registros [6], estilo que es completamente soportado por el lenguaje VHDL y que esta orientado a la implementación en hardware de algoritmos como los presentados.

En una descripción de hardware al estilo RTL, el diseño se divide en dos partes. Por un lado está la parte combinatoria del diseño, que generalmente ocupa mucha área y cuya complejidad algorítmica es reducida. La parte combinatoria de un diseño describe la forma en que se mueven los datos dentro del hardware y por eso es referida comúnmente como *datapath*. Por otra parte está la parte de control, que se encarga de manejar convenientemente los elementos del *datapath* con el fin de lograr el resultado esperado. Por lo general la etapa de control o *controlpath* ocupa poca área pero posee una complejidad algorítmica elevada.

Considerando todo esto se puede explicar el porqué las implementaciones de los algoritmos binario y de Montgomery en su versión de derecha a izquierda poseen *datapaths* idénticos. Esto se debe a que en ambos casos los algoritmos son simplemente acciones de control sobre el operador *Mont*.

Para el algoritmo de Montgomery versión izquierda a derecha sin embargo, deben existir dos operadores *Mont*, para aprovechar la ejecución de los mismos en forma concurrente.

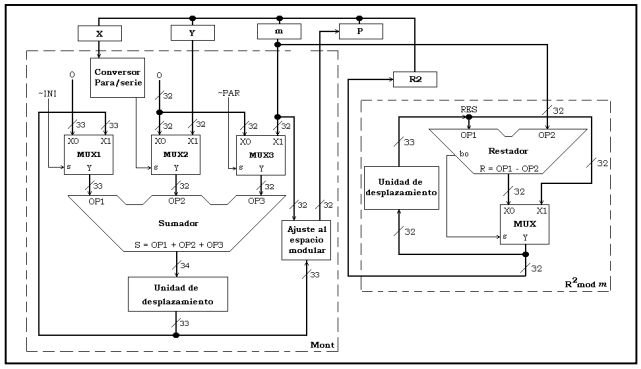


Figura 1. Flujo de datos de la arquitectura.

La Figura 1 muestra el *datapath* usado para el algoritmo binario presentado al principio, que de hecho es igual al *datapath* usado para la implementación del algoritmo de Montgomery versión derecha a izquierda.

Las líneas punteadas muestran la separación entre el operador *Mont* y el hardware para el cálculo de $R^2 mod\ m$.

4. RESULTADOS OBTENIDOS

Cada uno de los algoritmos discutidos en las secciones anteriores fue llevado a una implementación para N = 32 bits, usando el mismo dispositivo lógico en cada caso, para efectos de comparación. La implementación del algoritmo binario fue llamada POT1 en la etapa de diseño. Por otra parte, las implementaciones alternativas de los algoritmos de exponenciación de Montgomery fueron llamadas POT2 y POT3 respectivamente. El diseño POT2 corresponde a la versión derecha a izquierda mientras que POT3 corresponde a l versión izquierda a derecha.

Se usaron dos implementaciones para el operador *Mont*, presentado en la sección anterior. Estas dos implementaciones fueron llamadas MONT1 y MONT2 en la etapa de diseño. La tabla 1 muestra una comparación de desempeño entre tales diseños.

Tabla 1. Comparación entre las alternativas de diseño para el operador Mont.

Diseño	Área del dispositivo.	Tiempo de ejecución.
MONT1	18 %	2688 nS
MONT2	51 %	944 nS

Para los diseños POT1 y POT2 se usó el multiplicador MONT2, debido a su desempeño superior.

Sin embargo, el multiplicador MONT2 no puede usarse para la implementación del algoritmo de Montgomery versión izquierda a derecha, ya que como se recordará, se requiere usar dos multiplicadores independientes, con el fin de lograr la ejecución de productos concurrente. Es por eso que para el diseño POT3 tuvo que usarse el diseño MONT1, que a pesar de ser menos rápido ocupa un área de dispositivo considerablemente menor.

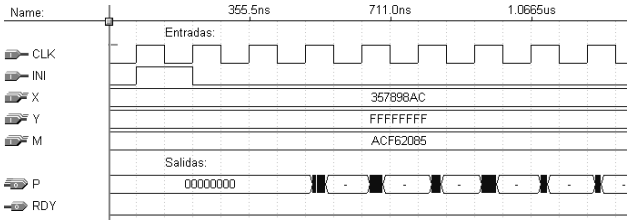
Los resultados de la implementación de los tres diseños son referidos en la tabla 2.

Tabla 2. Comparación entre las alternativas de diseño para la ejecución de la potencia modular.

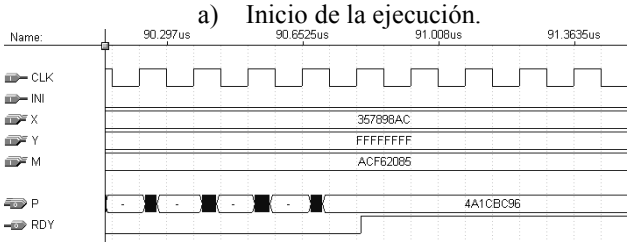
Diseño	Área del dispositivo.	Tiempo de ejecución.
POT1	89 %	162.07 µS
POT2	96 %	90.72 µS
POT3	64 %	108.02 µS

Estos resultados muestran la superioridad inherente que existe entre el algoritmo de exponenciación de Montgomery y el algoritmo binario.

Se puede ver además que el requerimiento de dos operadores *Mont* ejecutándose en paralelo puede ser una condición restrictiva para el desempeño del algoritmo de exponenciación de Montgomery en su versión izquierda a derecha.



a) Inicio de la ejecución.



b) Fin de la ejecución.

Figura 2. Resultados de la ejecución de una potencia modular usando el diseño POT2.

En la figura 2 se muestran resultados de simulación de la potencia modular usando el diseño POT2. El caso presentado en la figura es el de tiempo de ejecución máxima, es decir, cuando todos los bits del exponente son uno. Los valores presentados en la figura están todos en formato hexadecimal.

Una vez finalizada la ejecución de la potencia modular (lo que ocurre cuando la señal de control RDY ha cambiado a un estado activo), en P queda almacenado el valor $P = X^Y \bmod M$.

5. CONCLUSIONES

El desarrollo de hardware usando herramientas como los dispositivos lógicos programables y el lenguaje VHDL ofrece una flexibilidad muy cercana a la que se tiene en el desarrollo de una aplicación software. Paradójicamente, algunos desarrolladores critican esta ventaja dado que puede llegar a alentar una metodología de diseño de "prueba y error".

Del proceso de diseño se puede concluir que la mejor forma de lograr implementaciones eficientes en un dispositivo lógico programable es conocer en algún detalle su arquitectura interna. Si bien el uso de lenguajes de descripción de hardware portables como el VHDL desestimula al diseñador para que estudie los detalles de implementación en el dispositivo, muchas veces el conocimiento de tales detalles puede dar claves sobre como lograr diseños óptimos.

Debido a la capacidad del dispositivo lógico programable, los diseños presentados en este artículo están limitados a operandos de 32 bits. Para la expansión de este diseño a una cantidad de bits superior se plantean dos alternativas.

En primer lugar, se puede hacer uso de la multiplexión de las señales de interfaz, de modo que el dispositivo no esté limitado por la cantidad de pines de entrada salida. En segundo lugar, y considerando que un buen porcentaje del dispositivo ha sido usado en su etapa combinatoria, se puede pensar en dividir tal etapa en unidades funcionales. Que pueden ser a su vez implementadas en distintos dispositivos.

6. BIBLIOGRAFÍA

[1] BERNAL A. Conception et Etude d'une Architecture Numérique de haute performance pour le Calcul de la fonction Expnentielle Modulaire. 1999.

[2] RIVEST R, SHAMIR A, ADLEMAN L. A method for obtaining digital signatures and Public-key cryptosystems, Communications of the ACM, 21. 1978.

[3] HAN Y, MITCHELL C.J, GOLLMAN D. A fast modular exponentiation for RSA on systolic arrays. Inter. Journal computer Math., Vol. 63, 1997.

[4] MONTGOMERY P. L, Modular Multiplication without trial division. Mathematics of Computation. 1985.

[5] MENEZES A, OORSCHOT P. Van, VANSTONE S. Handbook of applied Cryptography. CRC press. 1998.

[6] ROZO A, GUERRERO M. Técnicas y tecnologías de diseño de alto nivel "VHDL". Centro de Microelectrónica de la Universidad de los Andes. 1997.