Implementación de un Hardware Reconfigurable de los Bloques de un Sistema RSA

Freddy Bolaños Martínez*
Rubén Darío Nieto Londoño**
Álvaro Bernal Noreña***

RESUMEN

En este trabajo se presenta el diseño en hardware reconfigurable de los sub-bloques que constituyen un sistema RSA de criptografía. Se presentan las diferentes arquitecturas que reproducen los algoritmos seleccionados y los resultados de simulación comportamental obtenidos a partir de la especificación en lenguajes de descripción de hardware. De igual forma se presentan algunos análisis de desempeño de los bloques constituyentes mencionados.

Palabras Clave: Arreglo de puertas programables por campo, lenguajes de descripción, Aritmética de residuos, Criptografía de clave pública.

* Ingeniero Electrónico, Estudiante Maestría en Ingeniería Electrónica - Universidad del Valle - Grupo de Investigac. en Arquitecturas Digitales y Microelectrónica - Escuela de Ingeniería Eléctrica y Electrónica - Facultad de Ingeniería - Universidad del Valle, Santiago de Cali, Colombia.
e-mail:fremar@yubarta.univalle.edu.co

** M.Sc., Estudiante del Programa de Doctorado en Ingeniería - Universidad del Valle - Grupo de Investigación en Arquitecturas Digitales y Microelectrónica - Profesor Asistente - Escuela de Ingeniería Eléctrica y Electrónica - Facultad de Ingeniería - Universidad del Valle, Santiago de Cali, Colombia.
e-mail:rnieto@univalle.edu.co

*** Ph.D., Profesor Titular - Escuela de Ingeniería Eléctrica y Electrónica - Facultad de Ingeniería - Universidad del Valle, Santiago de Cali, Colombia.
e-mail: alvaro@univalle.edu.co

Fecha de recepción: Abril 29 de 2004
Fecha de aprobación: Agosto 17 de 2004

ABSTRACT

In this work the hardware design of the constituent elements of a RSA system is presented. This document shows the design and simulation results of the four blocks in which the RSA system can be split. Individual performances of such blocks are presented, and the future work on this subject is presented too.

Key Words: Field programmable gate arrays, Hardware design languages, Residue arithmetic, Public key cryptography.

1. INTRODUCCIÓN

En este trabajo se presenta la implementación de cada uno de los bloques constitutivos de un prototipo del sistema criptográfico de clave pública RSA [1]. Un sistema criptográfico de clave pública es aquel en donde cada uno de los usuarios posee dos claves: Una clave que es de conocimiento público y otra que se mantiene privada.

Para lograr sus objetivos (autenticación de personas, protección de la información, comunicación segura entre entidades) un sistema criptográfico de clave pública debe valerse de una serie de operaciones matemáticas de naturaleza especial, es decir operaciones matemáticas orientadas a proteger la información de usuarios no autorizados. Dada su complejidad, la eficiencia con la que se implementen tales operaciones define el desempeño del sistema. En este trabajo se muestran los resultados de la implementación de estas operaciones en hardware reconfigurable. Se han aprovechado las ventajas que ofrece dicha implementación y su combinación con los lenguajes de descripción de hardware para el diseño eficiente. Se ha optado por la implementación en hardware de las operaciones más complejas y que involucran tiempos de ejecución considerables. Por esta razón se diseñaron cuatro bloques hardware para realizar tales operaciones. La justificación de esta elección es simple: un usuario RSA deberá hacer uso de la operaciones de exponenciación y producto modular. Las operaciones relacionadas con la generación de números primos y el cálculo del inverso multiplicativo modular están reservadas para el administrador del sistema. La implementación entonces se ha dividido en cuatro partes: Producto modular, exponenciación modular, inverso multiplicativo modular y generación de números primos.

2. EL SISTEMA RSA

El sistema RSA, como todos los sistemas criptográficos de clave pública, está basado en una función cuya complejidad es de orden polinomial cuando se calcula en un sentido y de orden exponencial cuando se trata de calcular su inversa. Tales funciones son conocidas como *funciones de un solo sentido*. En el caso de RSA, se usa la *exponenciación modular* como función de un solo sentido. La Ecuación (1) muestra la notación usada para denotar la operación de exponenciación modular.

$$Z = x^{y} \bmod M \qquad (1)$$

Donde Z es el resultado de la operación de exponenciación modular, X es llamada la base de la operación, Y es el exponente y M es el módulo de la misma.

Cuando un usuario RSA desea encriptar o cifrar un número A, deberá calcular una operación de exponenciación con el número A como base y una de sus dos claves (la pública o la privada) como exponente. El módulo M se asume constante para cada usuario. Tanto el módulo como las dos claves de usuario son suministrados por el administrador del sistema.

2.1. La exponenciación modular:

La exponenciación modular se realiza sobre un conjunto finito de números caracterizado por el módulo de la operación. A este conjunto de

números se le conoce como espacio o campo finito. El tamaño (la cantidad de elementos que conforman el conjunto finito) de un campo finito definido por un módulo M es exactamente igual a M. El resultado de cualquier operación realizada sobre un campo finito, deberá ser un número que pertenezca al mismo.

La exponenciación modular referenciada en (1) es el equivalente matemático a calcular la exponenciación natural X^Y y luego calcular el módulo o residuo de este resultado con respecto al número *M*. Está claro que el resultado *Z* siempre estará en el rango $0 \leq Z \leq M-1$. Sin embargo, esta forma de cálculo no es la más eficiente, porque supone el almacenamiento del resultado intermedio correspondiente a la exponenciación natural X^Y. Lo anterior se deriva de que en un entorno RSA, el tamaño en bits de las claves es considerable. La cantidad de celdas de memoria y unidades de procesamiento necesarias para manejar el resultado temporal X^Y impide la utilización de esta solución.

En lugar de eso, se han sugerido varios algoritmos [2] tendientes a mejorar el desempeño del cálculo en la exponenciación modular. Para la implementación de la que trata este documento, se ha optado por el algoritmo de *Exponenciación de Montgomery*, que a su vez está basado en una operación conocida como *Producto de Montgomery* [3].

Como se había mencionado, para encriptar o desencriptar datos, un usuario RSA sólo necesita desarrollar una operación de exponenciación modular. A su vez, para poder calcular una exponenciación modular, es necesario desarrollar una serie de productos de Montgomery. Lo anterior explica el porqué de dos de los bloques funcionales de los que trata este artículo. Estos bloques son los encargados de calcular el producto y la exponenciación de Montgomery.

2.2. Generación de claves:

En cualquier sistema criptográfico las operaciones de encriptado y desencriptado deben ser inversas entre sí. Es decir, si la función **E**(x) corresponde a la operación de encriptado y **D**(x) a la de desencriptado, se debe cumplir que:

$$D(E(x))=E(D(x))=x \qquad (2)$$

Lo anterior aplica también a RSA, donde las operaciones de encriptado y desencriptado corresponden a exponenciaciones modulares, de modo que la diferencia entre estas operaciones radica en el exponente usado para tal operación (la clave pública o la privada). De acuerdo con (2), si **e** es la clave privada de cierto usuario RSA y **d** es su clave pública, se debe cumplir:

$$(x^e \bmod M)^d \bmod M=(x^d \bmod M)^e \bmod M=x \bmod M \qquad (3)$$

Está claro que las claves no pueden ser escogidas al azar. Además, dado el carácter privado de una de las claves, debe existir un distribuidor confiable para las mismas (el administrador del sistema). Para generar claves RSA, el administrador del sistema debe encontrar dos números primos **p** y **q** cuya longitud en bits sea considerable. Lo anterior se hace con el fin de desalentar ataques en contra del sistema. El módulo **M** queda definido como el producto natural de los dos números primos referenciados antes:

$$M=p \cdot q \qquad (4)$$

A partir de (4) se define la función $\Phi(\mathbf{M})$, que corresponde a la cantidad de números en el campo finito definido por **M** que poseen inverso multiplicativo modular en este módulo. El inverso multiplicativo modular de un número **A** en un módulo **M** es aquel número **B** que satisface la ecuación:

$$A \cdot B \bmod M=B \cdot A \bmod M=1 \qquad (5)$$

La condición expresada en (5) se denota como $B=A^{-1} \bmod M$.

Una vez escogido el módulo para un usuario particular, se escoge de manera aleatoria un número **e** que posea inverso multipicativo modular. El número **e** corresponderá a la clave pública del usuario. Para calcular su clave privada (**d**), el administrador deberá encontrar el inverso multiplicativo modular del número **e** en módulo $\Phi(\mathbf{M})$, es decir:

$$d=e^{-1} \bmod \Phi(M) \qquad (6)$$

De esta forma, la única alternativa que posee un atacante para deducir la clave privada de un usuario a partir de la pública es factorizar su módulo y obtener los valores de **p** y **q**. Esta es la razón por la cual se escogen números primos **p** y **q** con un tamaño en bits considerable.

En conclusión, la labor del administrador del sistema requiere de dos operaciones que pueden llegar a ser lentas y complejas: Primero, debe generar los números primos **p** y **q** y en segundo lugar calcular el inverso multiplicativo modular de la clave pública **e**. Ésta es la explicación para los dos bloques faltantes en nuestra discusión: Un generador de números primos y un bloque para el cálculo del inverso multiplicativo modular.

3. PRODUCTO MODULAR

El producto modular en módulo M de dos números X e Y se realiza calculando el producto de los dos números y luego calculando el residuo de dividir este producto por M de acuerdo con la ecuación (7).

$$C=X \cdot Y \bmod M \qquad (7)$$

Sin embargo, esta forma de calcularla resulta poco práctica si el tamaño de los operandos es considerable, tal como lo requiere un sistema RSA para obtener grados de seguridad confiables. Si se van a multiplicar dos números de n bits, se necesitarán 2n bits para el almacenamiento del resultado intermedio, además, el cálculo posterior del residuo resulta demasiado lento. Dado que la exponenciación modular es la secuencia de varios productos modulares, una forma de mejorar el desempeño de un bloque de exponenciación modular, consiste precisamente en la optimización de su operación básica: la multiplicación modular. En [8] sugerimos varias arquitecturas para calcular la multiplicación modular, con base en una de las técnicas más importantes desarrollada para evitar los inconvenientes mencionados anteriormente: el algoritmo de Montgomery [3]. Los resultados se obtuvieron para operandos de 64 y 128 bits, considerando que estamos evaluando el desempeño de un circuito prototipo. En la figura 1, se presenta una arquitectura detallada propuesta para realizar la multiplicación modular de Montgomery para operandos de n bits.

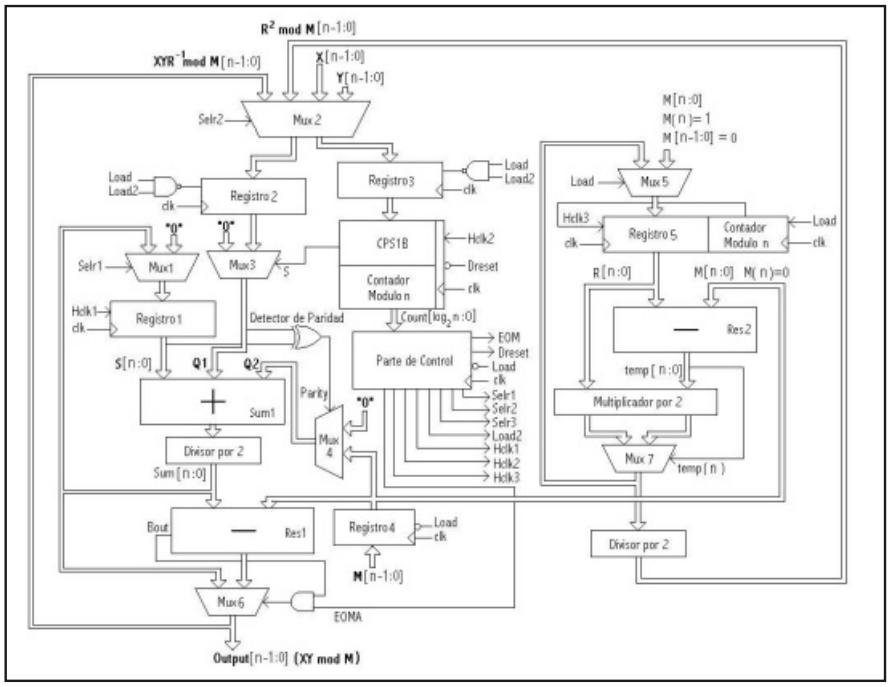


Figura 1. Diagrama Esquemático del Multiplicador de Montgomery para calcular $x \cdot y \bmod M$

La tabla 1 ilustra los resultados que se obtuvieron para los prototipos de 64 y 128 bits en el CPLD EP1K100FC484-1 de Altera que posee 4992 celdas lógicas.

Finalmente, en las figura 2 y 3 se muestran los resultados de la simulación de los diseños de 64 y 128 bits.

Tabla 1. Resultados de las simulaciones de los multiplicadores de 64 y 128 Bits

Diseño: 64 BITS			
Familia	**Dispositivo**	**Tiempo de Cálculo Promedio [μS]**	**Número de Celdas Lógicas Requeridas**
ACEX1K	EP1K100FC484-1	5,07	2488/4992
Diseño: 128 BITS			
ACEX1K	EP1K100FC484-1	13,9	4917/4992

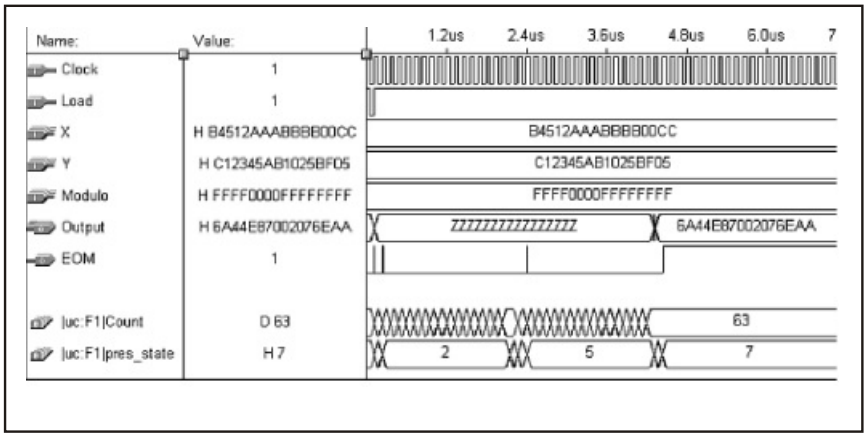


Figura 2. Simulación obtenida para el multiplicador de 65 Bits

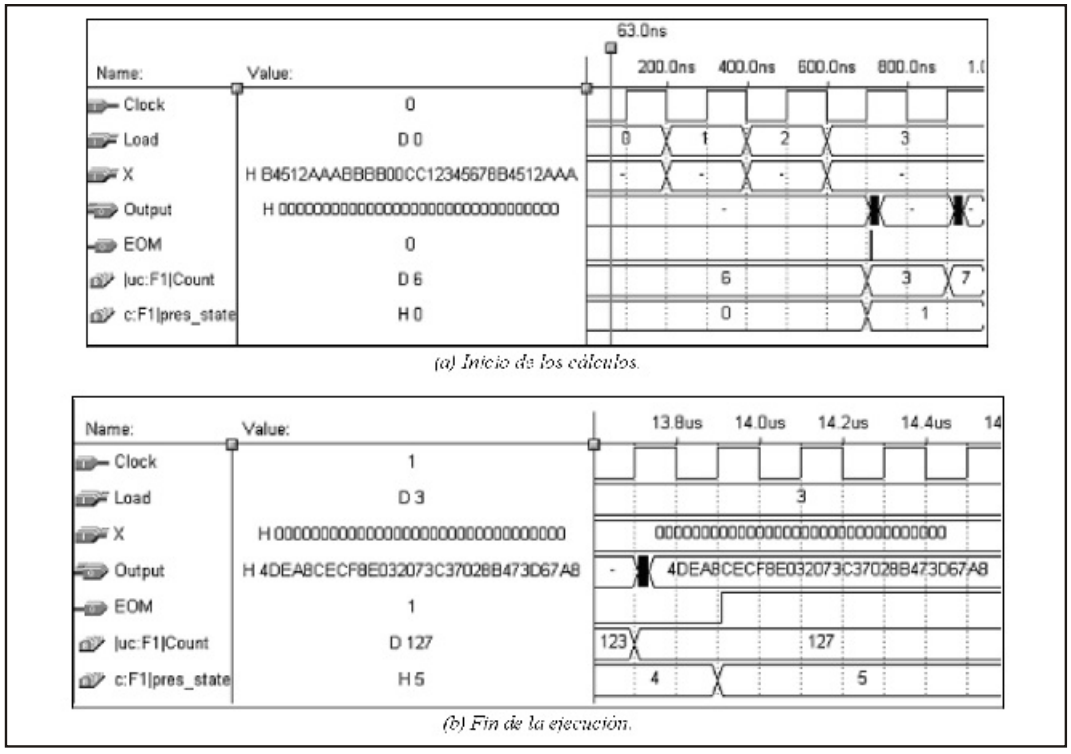


Figura 3. Simulación obtenida para el multiplicador de 128 Bits

4. EXPONENCIACIÓN MODULAR

La exponenciación modular es el equivalente a calcular una exponenciación natural para luego calcular el módulo del resultado, aunque esta aproximación es extremadamente ineficiente. También se dijo que la operación de exponenciación se basó en un operador más simple conocido como Multiplicador de Montgomery.

En el proceso de diseño del bloque de exponenciación se probaron varias implementaciones [4]. Luego de comparar el desempeño de cada una, se optó por el algoritmo de Exponenciación de Montgomery [5]. Para esta aproximación, el número de productos de Montgomery promedio es de 1.5·N, donde N es el tamaño en bits de los operandos. Además se cuenta con una versión concurrente del algoritmo, donde se deben calcular la misma cantidad de productos, pero la concurrencia hace que el tiempo de cálculo promedio sea el equivalente a N productos de Montgomery. El precio que se debe pagar por esta concurrencia es una complejidad superior en el hardware y un área de ocupación mayor dentro del dispositivo lógico programable.

Está claro que el diseño del multiplicador de Montgomery puede afectar de manera ostensible el desempeño en el cálculo de la potencia, de modo que se probaron dos alternativas en el diseño final. Cada una satisface una de las metas en el compromiso área-velocidad. La tabla II muestra los resultados obtenidos cuando se probaron los dos multiplicadores en el dispositivo reconfigurable EPF10K10AQC208-1 de Altera. Se trata de multiplicadores con operandos de 32 bits de longitud.

Tabla 2. Comparación del Desempeño y Área para dos multiplicadores de Montgomery

Nombre del Diseño	Área del Dispositivo Ocupada	Tiempo de Ejecución
MONT1	18%	2.688 µS
MONT4	51%	0.944 µS

Para la potencia se probaron igualmente dos alternativas: Los diseños POT2 y POT3. El primer diseño implementa una versión no-concurrente del algoritmo de exponenciación de Montgomery, y dada su menor complejidad y la necesidad de un solo multiplicador, fue posible usar el multiplicador MONT4 en su implementación. En el diseño POT3 sin embargo, que corresponde a la versión concurrente del algoritmo, era necesario tener dos multiplicadores independientes. Por esta razón no fue posible usar el multiplicador MONT4 referido en la Tabla I y en su lugar se optó por el diseño MONT1. los resultados de simulación para los diseños POT2 y POT3 sobre el dispositivo EPF10K10AQC208-1 se muestran en la Tabla III.

Tabla 3. Comparación entre las dos Alternativas para la Exponenciación de Montgomery

Nombre del Diseño	Área del Dispositivo Ocupada	Tiempo de Ejecución
POT2	96%	90.72 µS
POT3	64%	108.02 µS

Los resultados mostrados en la Tabla III, se explican del hecho que, si bien el diseño concurrente de la exponenciación de Montgomery es inherentemente más rápido, fue necesario usar multiplicadores más lentos en su implementación debido a problemas de área. La figura 4 muestra un esquema de la ruta de datos simplificada para este multiplicador. Como puede verse, son necesarios dos multiplicadores de Montgomery en su implementación, y la complejidad de la ruta de control se duplica con respecto a la del diseño POT2.

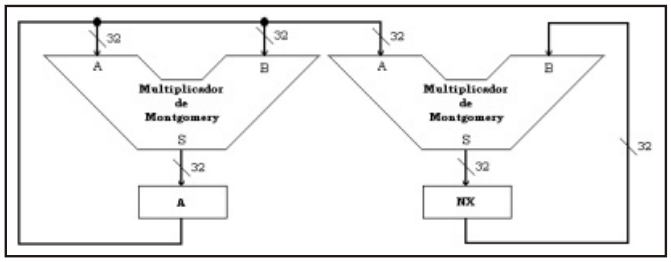
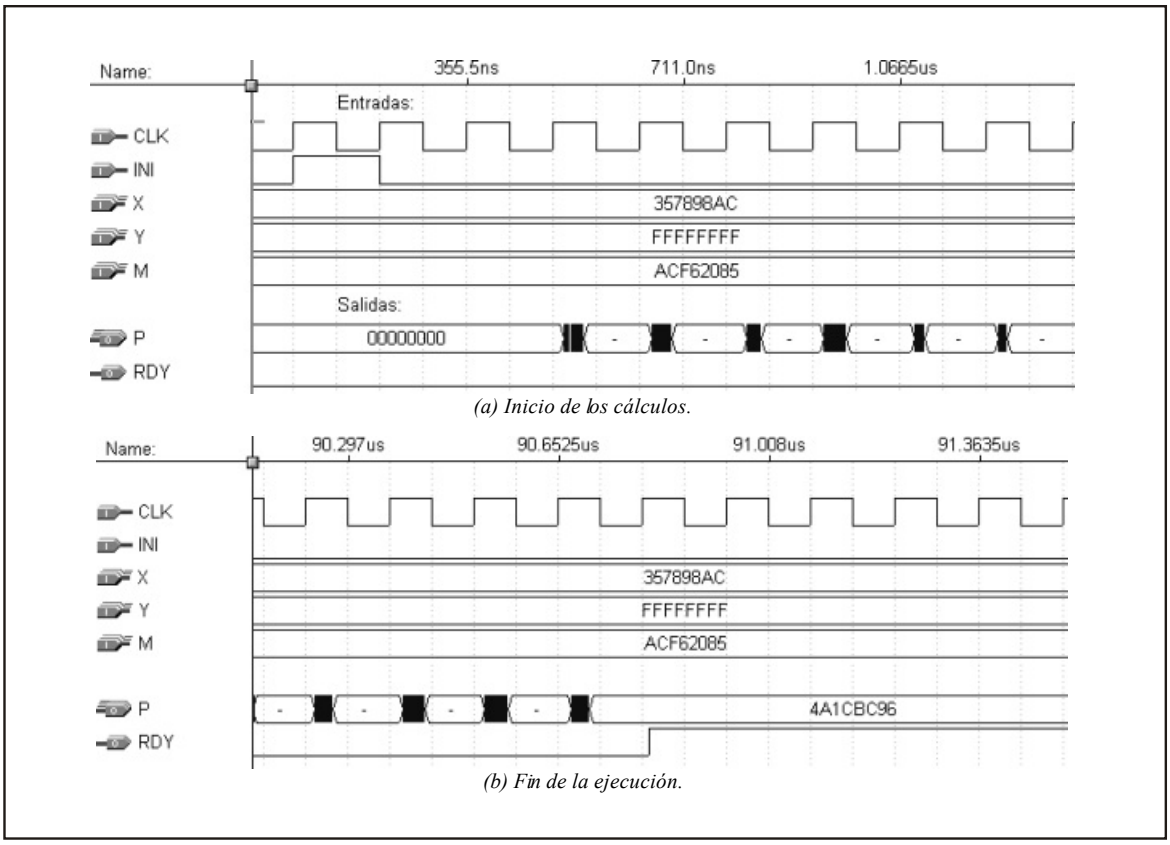


Figura 4. Ruta de datos simplificada del diseño POT3

Finalmente, en la figura 5 se muestran los resultados de simulación del diseño POT2, escogido para la implementación final.



(a) Inicio de los cálculos.

(b) Fin de la ejecución.

Figura 5. Simulación obtenida para el diseño final de la exponenciación modular

5. INVERSO MULTIPLICATIVO MODULAR

Para la implementación del bloque que calcula el inverso multiplicativo modular, se tuvieron en cuenta dos algoritmos [2]. Se trata del Algoritmo de Euclides y del Algoritmo Binario. En ambos casos, el cálculo del inverso multiplicativo modular está basado en operaciones aritméticas tradicionales como suma, resta producto y división, de modo que buena parte del trabajo de diseño se centró en la optimización de estas operaciones usando el lenguaje VHDL [6] y [7].

Las figuras 6 y 7 presentan las arquitecturas usadas para los algoritmos de Euclides y Binario, respectivamente [10]. Como puede verse, se trata de diseños bastante complejos, que pueden llegar a ocupar un área considerable. Es por esta razón que su verificación se realizó para operandos de ocho bits solamente, lo cual no implica que la arquitectura pueda ser extendida a un mayor número de bits.

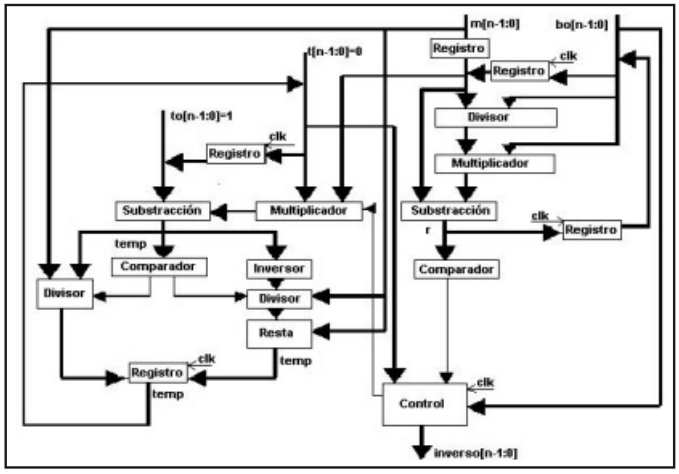


Figura 6. Arquitectura para el Algoritmo de Euclides

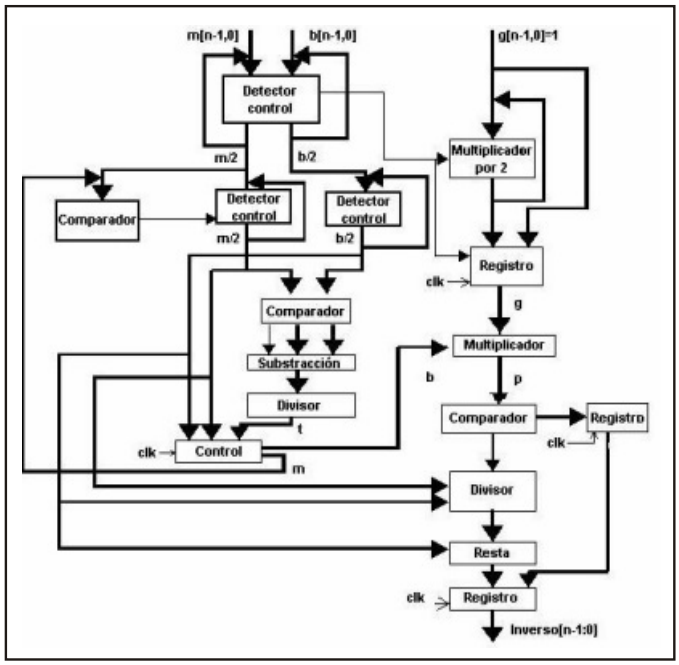


Figura 7. Arquitectura para el algoritmo binario

La Tabla IV muestra una comparación entre el desempeño de las dos alternativas mostradas. La simulación de estos diseños se hizo sobre el dispositivo EP1K100FC484-2 de Altera. De nuevo, es evidente el compromiso existente entre desempeño y área.

Tabla 4. Comparación entre las dos alternativas para el cálculo del inverso multiplicativo modular

Arquitectura	Área del Dispositivo Ocupada	Tiempo de Ejecución
Euclides	25%	0.6 µS
Binario	16%	2.5 µS

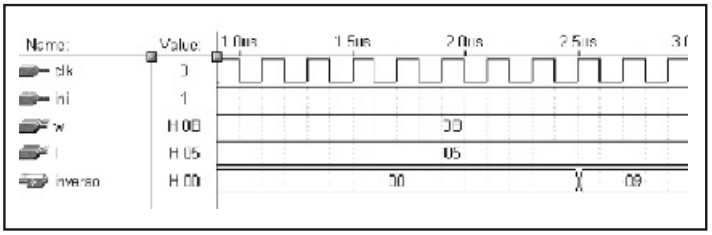


Figura 8. Resultados de la simulación para la arquitectura basada en el algoritmo de Euclides.

De nuevo entre las dos alternativas de diseño se optó por la más rápida, de modo que se adoptó la Arquitectura basada en el algoritmo de Euclides para la implementación final. La figura 8 muestra una simulación típica de este diseño con las mismas condiciones que las referidas para la Tabla IV.

6. GENERADOR DE NÚMEROS PRIMOS

Una prueba de primalidad es básicamente un algoritmo que nos permite determinar si un número es primo o no. Los números primos juegan un papel importante en la criptografía, más exactamente en la generación de parámetros para los sistemas de clave pública. Antes de la aplicación del criptosistema de clave pública, cada participante debe generar un par de claves. Esto involucra las siguientes tareas:

- ☑ Determinar dos números primos, *p* y *q*.
- ☑ Seleccionar uno de los números *e* o *d* y calcular el número faltante por medio del inverso multiplicativo modular.

Nuestro objetivo es la determinación del par de números primos. A causa de que el valor de $n = pq$ será conocido por cualquier oponente potencial, para prevenir el descubrimiento de *p* y *q* por métodos exhaustivos, los primos deben ser escogidos de un conjunto suficientemente grande (por ejemplo *p* y *q* deben ser números grandes), por otra parte, el método usado para encontrar primos grandes debe ser razonablemente eficiente.

En el presente no hay técnicas útiles que generen primos arbitrariamente grandes, por tanto se requiere algún otro medio para abordar el problema. El procedimiento que es usado generalmente es escoger al azar un número impar del orden de magnitud deseada y ***probar si tal número es primo***. Si no, escoger sucesivamente números al azar hasta que sea encontrado uno que pruebe ser primo.

Han sido desarrolladas una variedad de ***pruebas de primalidad***. Casi invariablemente, las pruebas son *probabilísticas*, esto es, la prueba

simplemente determinará que un número entero es probablemente primo. A pesar de esta falta de certeza las pruebas pueden ser ejecutadas muchas veces, de tal manera que se haga la probabilidad tan cercana a uno como se desee. En [9], se sugiere el diseño de una arquitectura para realizar la prueba de primalidad para números enteros de 16 bits con base en el algoritmo de Miller-Rabin [2]. En la figura 9 se presenta un diagrama esquemático del diseño.

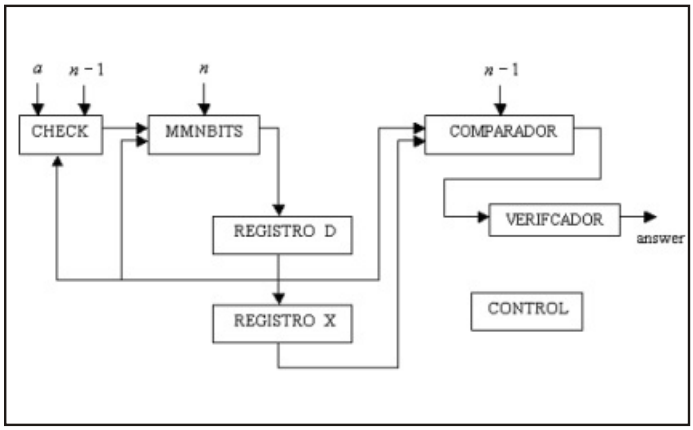


Figura 9. Diagrama de bloques para probador de primalidad

En la figura 10 se pueden apreciar los resultados gráficos de la simulación para 16 bits tomando como ejemplo la semilla 41391.

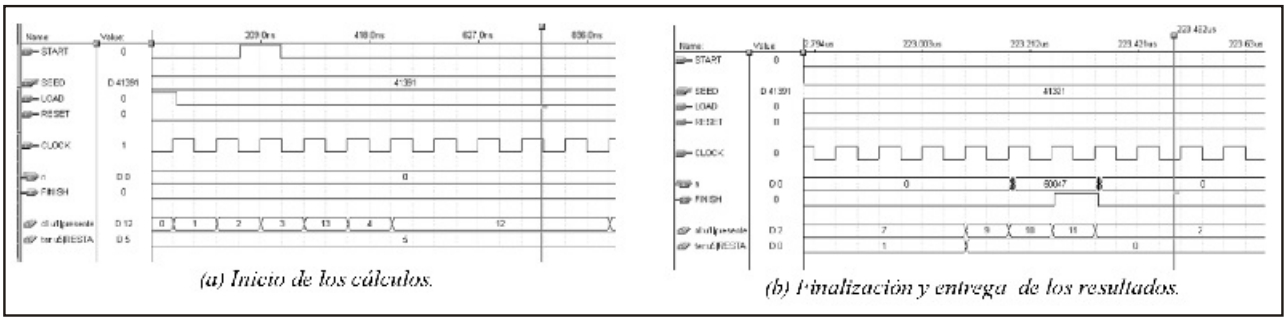


Figura 10. Resultados de la simulación del generador de números primos

7. CONCLUSIONES

De todas las operaciones referidas en las secciones anteriores, quizás la exponenciación modular es la más crítica. Esto se debe a dos razones. En primer lugar, la exponenciación es la operación que involucra los tiempos de cálculo más grandes. Recuérdese además que ésta es precisamente la operación usada para encriptar y desencriptar los datos en un sistema RSA. En comparación, las operaciones como el cálculo de números primos y el inverso multiplicativo modular, sólo deben ejecutarse una vez para cada usuario del sistema.

En consecuencia, nuestro interés actual se orienta a una implementación hardware más eficiente de la exponenciación modular. Existen actualmente dos trabajos paralelos: En primer Lugar se está trabajando en el uso de bases altas para la representación de los operandos de la exponenciación y en segundo lugar se desea combinar los métodos m-ario y de notación redundante para mejorar el desempeño.

Finalmente, se tiene planeado integrar todos los elementos referidos en este trabajo para la implementación de un sistema RSA completo. En tales condiciones se pueden definir dos modos de funcionamiento, que a su vez condicionan las operaciones que se pueden realizar en el sistema. Para el modo *usuario*, el sistema debe permitir realizar la operación de exponenciación, que a su vez sirve para encriptar y desencriptar los datos. De otro lado, en el modo *administrador* el sistema deberá permitir la generación de números primos y del inverso multiplicativo modular, lo que permite a su vez la generación de claves.

8. REFERENCIAS

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

[2] A MENEZES, P. V. OORSCHOT, VANSTONE S. "Handbook of Applied Cryptography". CRC Press. 1996. pp. 613 - 630.

[3] P. L. MONTGOMERY. "Modular Multiplication without trial divisions". Mathematics of Computation. 1985.

[4] F. BOLAÑOS, R. D. NIETO, A. BERNAL. Implementación de la función Exponenciación Modular en hardware Reconfigurable. IX Workshop IBERCHIP. 2003.

[5] C. D. WALTER. Montgomery Exponentiation with no final subtractions. Electronic Letters, 35(21):1831 1832. 1999.

[6] WIENER M. J. , Performance comparison of Public Key Cryptosystems, Cryptobytes, Vol4. Number I, Summer 1998.

[7] C.L. RIVERA, R. NIETO, A. BERNAL, *Implementación de la función inverso multiplicativo modular en hardware reprogramable*" ,., IX Workshop IBERCHIP IWS 2003, La Habana, Cuba, Marzo 26-28, 2003.

[8] R. PALACIOS, R. D. NIETO, A. BERNAL. Implementación de la Multiplicación Modular de Montgomery en Hardware Reprogramable**.** IX Workshop IBERCHIP. 2003.

[9] R. LLORENTE, Hardware para la prueba probabilística de primalidad en números enteros. Trabajo de grado en Ingeniería Electrónica. Universidad del Valle. Septiembre de 2001.