SENNOVA
Sistema de Investigación, Desarrollo Tecnológico e Innovación

HARDWARE PARA UNA PRUEBA PROBABILÍSTICA DE PRIMALIDAD

HARDWARE FOR A PRIMALITY PROBABILISTIC TEST

doi:http://doi.org/10.23850/23899573.1941
Recibido: 12.12.2018 Aceptado: 09.05.2019

Rodrigo Alberto Llorente Triana
Servicio Nacional de Aprendizaje SENA
rallorentet@sena.edu.co
Colombia

Iván Londoño
Servicio Nacional de Aprendizaje SENA
ivan.londono@misena.edu.co
Colombia

Para Citar:
Llorente, R., Londoño, I. (2019) Hardware para una prueba probabilística de primalidad, Revista SENNOVA: Revista del Sistema de Ciencia, Tecnología e Innovación, 4 (1), 19-33. doi:http://doi.org/10.23850/23899573.1941

doi:http://doi.org/10.23850/23899573.1061

RESUMEN

En este mundo moderno regido por computadoras parece que lo que más tiene valor es la información. El dinero por ejemplo no es más que números en una computadora. Pero sí la información es tan importante mucho más lo es el tener un acceso restringido a la misma. Es de esta necesidad de donde nace la criptografía, ciencia o técnica que se encarga de buscar formas de ocultar la información almacenada o que viaja por las redes públicas a personas no autorizadas. En un principio la criptografía se desarrolló totalmente en software, pero con la necesidad de hacer las comunicaciones cada vez más rápidas pronto surgió la idea de implementar los algoritmos criptográficos en hardware. El trabajo presentado a continuación proviene de la idea de implementar en hardware el más popular de los algoritmos criptográficos de clave pública, el algoritmo RSA. En realidad RSA es demasiado extenso y complejo para ser realizada una implementación hardware de él en un solo trabajo. Por esta razón se dividió el algoritmo en cuatro partes siendo una de ellas la generación de números primos, parte fundamental en RSA y tema principal de este trabajo. Haremos una breve introducción a lo que es la criptografía y nos daremos cuenta de por qué son importantes los números primos para RSA y la manera cómo se deben hallar estos números primos. A diferencia de lo que uno piensa, encontrar números primos es una tarea complicada cuando los números son grandes. La búsqueda de números primos grandes se basa en las pruebas probabilísticas de primalidad, pruebas que no son más que algoritmos que nos dan información acerca de la probabilidad que tiene un número de ser primo, siendo la mejor de ellas la prueba de Miller – Rabin y de la que se hará una implementación hardware en este trabajo.

Palabras clave: FPGA, prueba de primalidad, Miller-Rabin, números primos, RSA, criptografía.

ABSTRAC

In this modern world ruled by computers it seems that what counts most is information. Money, for example, is nothing more than numbers on a computer. But if information is so important, it is much more important to have restricted access to it. It is from this need that cryptography, science or technology that is responsible for finding ways to hide stored or traveling on public networks information to unauthorized persons.

At first, cryptography was developed entirely in software, but with the need to make communications faster and faster, the idea of implementing cryptographic algorithms in hardware soon arose. The work presented below comes from the idea of implementing in hardware the most popular cryptographic algorithms of public key, the RSA algorithm.

Actually RSA is too extensive and complex to be made a hardware implementation of it in a single job. For this reason, the algorithm was divided into four parts, one of them being the generation of prime numbers, a fundamental part in RSA and the main theme of this work.

We will make a brief introduction to what cryptography is and we will realize why the prime numbers for RSA are important and how these prime numbers should be found. Unlike what one thinks, finding prime numbers is a complicated task when the numbers are large. The search for large prime numbers is based on the probabilistic tests of primality, tests that are nothing more than algorithms that give us information about the probability of having a prime number, the best of which is the Miller - Rabin test. A hardware implementation of this test will be made in this work.

Keywords: FPGA, Primality Test, Miller-Rabin, Prime Numbers, RSA, Cryptography.

INTRODUCCIÓN

Según el diccionario de la Real Academia, la palabra criptografía se define como: "Arte de escribir con clave secreta o de un modo enigmático". Obviamente la criptografía hace años que dejó de ser un arte para convertirse en una técnica, o más bien un conglomerado de técnicas, que tratan sobre el ocultamiento de la información a observadores no autorizados .

La encriptación se refiere a la transformación de datos en texto claro a datos en texto cifrado, lo cual les da casi imposibilidad de leerlos sin el conocimiento de una "clave". El recuperar el texto claro del texto cifrado requiere de la clave, proceso que se conoce como desencriptación. Esta clave es información secreta, la privacidad del texto cifrado depende de la privacidad y la longitud dicha clave.

Existen dos tipos fundamentales de criptosistemas: simétricos o de clave secreta, y asimétricos o de clave pública. Los criptosistemas simétricos o de clave secreta son aquellos que emplean la misma clave tanto para encriptar como para desencriptar. Presentan el inconveniente de que para ser empleados en comunicaciones, la clave debe estar tanto en el emisor como en el receptor, lo cual nos lleva a preguntarnos cómo transmitir la clave de forma segura.

Los criptosistemas asimétricos o de clave pública emplean una doble clave, una se conoce como clave privada y la otra como clave pública. Una de ellas sirve para encriptar y la otra para desencriptar. Estos criptosistemas deben cumplir además que el conocimiento de la clave pública no permita calcular la clave privada. Ofrecen un abanico superior de posibilidades, pudiendo emplearse para establecer comunicaciones seguras por canales inseguros, -puesto que únicamente viaja por el canal la clave pública, que solo sirve para encriptar -, o para llevar a cabo autenticaciones.

En la práctica se emplea una combinación de estos dos tipos de criptosistemas. Puesto que los segundos presentan el inconveniente de ser computacionalmente mucho más costosos que los primeros, en el mundo real se encriptan los mensajes (largos)

doi:http://doi.org/10.23850/23899573.1061

mediante algoritmos simétricos, que suelen ser muy eficientes, y luego se hace uso de la criptografía asimétrica para encriptar las claves simétricas (cortas).

Entre todos los algoritmos utilizados para los criptosistemas de clave pública, quizá RSA sea el más sencillo de comprender e implementar. Sus claves pública y privada sirven indistintamente tanto para encriptar como para autenticar. Debe su nombre a sus tres inventores: Ronald Rivest, Adi Shamir y Leonard Adleman, y estuvo bajo patente de los laboratorios RSA hasta el 20 de septiembre de 2000, por lo que su uso comercial estuvo restringido hasta esa fecha.

RSA se basa en la dificultad para factorizar grandes números. Las claves pública y privada se calculan a partir de un número que se obtiene como producto de dos primos grandes. El atacante se enfrentará, si quiere recuperar el texto legible a partir de su correspondiente ilegible y la clave pública, a un problema de factorización.

Los fundamentos matemáticos del algoritmo RSA plantean la necesidad del uso de dos números primos como parte esencial de la generación de las claves. A primera vista conseguir números primos parece una tarea fácil, pero en realidad es mucho más complicada de lo que uno puede imaginar cuando los números primos son muy grandes.

Por otra parte, el método utilizado para encontrar números primos grandes debe ser razonablemente eficiente; pero sí factorizar números grandes es difícil -recordemos que en este principio se basa la seguridad de RSA- ¿cómo podríamos generar números primos de una manera sencilla? El truco está en que responder la pregunta: ¿es *n* primo?, es mucho más fácil que responder: ¿cuáles son los factores primos de *n*?

MATERIALES Y MÉTODOS

La manera incorrecta de encontrar números primos es generar un número al azar y luego hacer esfuerzos por factorizarlo, factorización que se puede conseguir comprobando sí *n* es divisible por cualquier número p.rimo $\leq \sqrt{n}$. En la práctica se necesitan métodos mucho más eficientes para encontrar números primos. La manera correcta es generar un número al azar y probar sí lo es, sí no lo es, volver a generar otro número y repetir la prueba hasta que se consiga uno que lo sea.

Consideremos la primera aproximación a la solución del problema [1]:

1. Generar al azar un número impar y del tamaño apropiado.
2. Hacerle una prueba de primalidad
3. Sí es compuesto, regresar al primer paso.

Una ligera modificación es considerar candidatos restringidos a alguna búsqueda secuencial comenzando desde *n*; puede ser usada una búsqueda secuencial trivial tal como n, n+2, n+4… etc.; la búsqueda secuencial tiene ventajas.

En el paso 2, la prueba de primalidad podría ser una que pruebe que el candidato es primo (caso en el que el resultado del generador es llamado un primo comprobable), o una prueba que establezca un resultado débil, tal como que *n* es "probablemente primo" (en tal caso se dice que el resultado del generador es un probable primo).

En este último caso deben ser dadas cuidadosas consideraciones al significado exacto de esta expresión. Muchas pruebas, llamadas pruebas probabilísticas de primalidad, son totalmente correctas cuando declaran que un candidato *n* es compuesto, pero cuando lo declaran "probablemente primo" no dan una prueba matemática de que *n* es primo. Sin embargo, cuando uno usa apropiadamente el último caso, frecuentemente puede sacar conclusiones más que apropiadas para sus propósitos. Por esta razón, dichas pruebas son llamadas más apropiadamente pruebas de composición que pruebas probabilísticas de primalidad. También existen pruebas de primalidad las cuales le permiten a uno concluir con total certidumbre matemática que un número *n* es primo, pero generalmente requieren considerablemente grandes recursos computacionales.

Pruebas probabilísticas de primalidad

Las pruebas probabilísticas de primalidad son métodos mediante los cuales se prueban enteros positivos arbitrarios, para obtener información parcial respecto de su primalidad [1]. Específicamente, las pruebas probabilísticas de primalidad tiene el siguiente marco teórico:

Para cada entero positivo e impar n, se define un conjunto $W(n) \subset Z_n$ tal que sean válidas las siguientes propiedades:

1. Dado $a \in Z_n$, puede ser probado en tiempo polinomial determinístico sí $a \in W(n)$;
2. Si n es primo, entonces $W(n) = \phi$ (conjunto vacío); y
3. Si n es compuesto, entonces $\#W(n) \geq \frac{n}{2}$.

Si n es compuesto, los elementos de $W(n)$ son llamados *testigos* de la composición de n y los elementos del conjunto complementario $L(n) = Z(n) - W(n)$ son llamados *mentirosos* o *cómplices*.

Una prueba probabilística de primalidad utiliza estas propiedades de los conjuntos $W(n)$ de la siguiente manera. Suponga que n es un número cuya primalidad está en cuestión. Se elige al azar un entero $a \in Z_n$, y se prueba sí $a \in W(n)$. La prueba responde "compuesto" sí $a \in W(n)$ o responde "primo" sí $a \notin W(n)$. Sí realmente $a \in W(n)$, entonces se dice que n falla la prueba de primalidad para la base a; en tal caso, n es sin duda compuesto. Sí $a \notin W(n)$, entonces se dice que n pasa la prueba de primalidad para la base a; en este caso, ninguna conclusión puede ser obtenida con absoluta certeza acerca de la primalidad de n, y la declaración "primo" puede ser incorrecta.

doi:http://doi.org/10.23850/23899573.1061

En este punto se puede ver claramente porque una prueba probabilística de primalidad es llamada más apropiadamente, prueba de *composición*.

Cualquier ejecución de la prueba que declare "compuesto", establecerá esto con certeza; por otra parte, todas las ejecuciones independientes y sucesivas de la prueba, las cuales declaren "primo", dan un nivel de confianza de que la entrada es un primo, nivel que se puede elevar al valor que se desee. Por ser las ejecuciones de la prueba hechos independientes, la probabilidad acumulativa de error es multiplicativa. Si la prueba es corrida independientemente t veces, para el mismo número compuesto n, la probabilidad de que sea declarado "primo" todas las t veces, es decir la probabilidad de error, es a lo sumo $1/2^t$.

Se le llama *probable primo* a un entero n, del cual se piensa que es primo basándose en pruebas probabilísticas de primalidad. La prueba probabilística de primalidad más usada en la práctica es la prueba de Miller – Rabin, también conocida como la prueba del seudoprimo fuerte [1].

Búsqueda aleatoria de números primos

Por el teorema de los números primos, la proporción de enteros positivos $\leq x$ que son primos es aproximadamente 1 / 1n *x*. Dado que la mitad de todos los enteros $\leq x$ son pares, la proporción de enteros impares $\leq x$ que son primos es aproximadamente 2 / 1n *x*. Por ejemplo, la proporción de todos los enteros impares $\leq 2^{512}$ que son primos es aproximadamente 2 / (512 × 1h 2) ≈1 / 177 . Esto sugiere que una estrategia razonable para seleccionar un probable primo aleatorio de k bits es escoger repetidamente al azar enteros n impares de k bits hasta que sea encontrado uno que sea declarado "primo" por **MILLER – RABIN** (n,t) para un valor apropiado del parámetro de seguridad t. [1]

Sí un entero n, aleatorio, impar y de k bits es divisible por un pequeño primo, entonces es menos costoso computacionalmente hablando, descartar el candidato n por la prueba de la división que usando la prueba de Miller – Rabin.

Dado que es relativamente grande la probabilidad de que un entero aleatorio n tenga un pequeño divisor primo, antes de aplicar la prueba de Miller – Rabin, el candidato n debe probarse para pequeños divisores que se encuentren bajo un límite B predeterminado. Esto se puede conseguir dividiendo n por todos los primos abajo de B, o calculando los máximos comunes divisores de n y de los productos de varios de los primos $\leq B$. La proporción de candidatos enteros e impares n no descartados por la prueba de la división es $\prod_{3\leq p\leq B}\left(1-\frac{1}{p}\right)$ la cual, por el teorema de Martens, es aproximadamente 1.12 /1n B (aquí p fluctúa sobre valores primos). Por ejemplo, sí $B=256$, entonces solo el 20 % de los candidatos n enteros e impares, pasan la prueba de la división, es decir, el 80 % son descartados antes de que sea desarrollada la prueba

de Miller – Rabin que es más costosa. La Tabla 1 muestra el algoritmo para la búsqueda aleatoria de números primos usando la prueba de Miller-Rabin.

Tabla 1.
Búsqueda aleatoria de un primo usando la prueba de Miller – Rabin

BÚSQUEDA ALEATORIA (k, t)

ENTRADA: un entero k , y un parámetro de seguridad t .

SALIDA: un probable primo de k bits aleatorio.

1. Generar aleatoriamente un entero n impar y de k bits.
2. Usar la prueba de la división para determinar sí n es divisible por algún impar primo $\leq B$. Sí es así regresar al paso 1.
3. Sí MILLER – RABIN (n, t) devuelve "primo" entonces devolver (n).

De lo contrario, regresar al paso 1.

Fuente: Referencia [1]

Una técnica alternativa a generar candidatos n al azar en el paso 1 del algoritmo *Búsqueda Aleatoria*, es seleccionar primero un número n_0 aleatorio, impar y de k bits, y entonces probar los s números $n = n_0, n_0 + 2, n_0 + 4, \ldots, n_0 + 2(s-1)$ para primalidad. Sí son encontrados compuestos todos los s candidatos, se dice que el algoritmo ha fallado.

Prueba de Miller-Rabin

El seudocódigo para la prueba es el siguiente: [2]

Tabla 2
Prueba de Miller – Rabin en su forma binaria

MILLER – RABIN (n, t)

ENTRADA: un entero impar $n \geq 3$ y un parámetro de seguridad $t \geq 1$

SALIDA: una respuesta "primo" o "compuesto" a la pregunta: "¿es n primo?"

Para $i \leftarrow 1$ hasta t haga:

$a \leftarrow$ RANDOM $(1, n-1)$

Sí TESTIGO (a, n) entonces:

devuelva "compuesto"

Devuelva "primo"

Fuente: Referencia [2]

El algoritmo requiere de una función RANDOM $(1, n-1)$, la cual devuelve un entero a elegido al azar, tal que $1 \leq a \leq n-1$. El parámetro de seguridad t es la cantidad de intentos que se deben hacer por buscar un testigo de la composición de n .

La función auxiliar TESTIGO (a, n), devuelve "cierto" sí es posible construir con base en a una prueba de que n es compuesto. La prueba TESTIGO (a, n) es similar a $a^{n-1} \not\equiv 1 \mod n$, pero más

doi:http://doi.org/10.23850/23899573.1061

efectiva. El algoritmo en la Tabla 3 la enseña. [2]

Las líneas que se encuentran dentro del lazo, básicamente calculan $d = a^{n-1} \mod n$. Se puede observar que para cada bit se realiza como mínimo una multiplicación modular y adicionalmente, si el bit es 1, se realiza otra multiplicación modular. El secreto detrás de TESTIGO es la búsqueda de raíces cuadradas no triviales de 1 módulo n que realiza en cada iteración. Obsérvese que cada valor de d (el acumulado de la exponenciación) es almacenado en x de tal forma que, en el paso siguiente a la elevación de d al cuadrado, se pueda comprobar sí x es una raíz cuadrada no trivial de 1 módulo n.

Las últimas tres líneas devuelven "cierto" si el valor calculado para $a^{n-1} \mod n$ no es igual a 1.

Demostremos ahora que una prueba de que n es compuesto puede ser construida usando a. Sí TESTIGO (a, n) devuelve "cierto" (en la instrucción que no está dentro del ciclo), entonces éste ha descubierto que $d = a^{n-1} \mod n \neq 1$. Si n es primo, sin embargo, tenemos por el teorema de Fermat que $a^{n-1} \equiv 1 \mod n$ para toda $a \in Z_n^+$. Por lo tanto, n no puede ser primo y la ecuación $a^{n-1} \mod n \neq 1$ es una prueba de este hecho.

Tabla 3
Algoritmo para Testigo

TESTIGO (a, n)

ENTRADA: un entero impar $n \geq 3$ y un número entero a generado al azar.

$n - 1 = b_k \; b_{k-1} \ldots b_0$

SALIDA: una respuesta "cierto" o "falso" a la pregunta: "¿a es un testigo de la composición de n"?

$d \leftarrow 1$

Para $i \leftarrow k$ hasta 0 haga:

$x \leftarrow d$

$d \leftarrow (d \cdot d) \mod n$

Sí $d = 1$ y $x \neq 1$ y $x \neq n - 1$ entonces:
devuelva "cierto"

Sí $b_i = 1$ entonces:

$d \leftarrow (d \cdot a) \mod n$

Sí $d \neq 1$ entonces:
devuelva "cierto"

Devuelva "falso"

Fuente: Referencia [2]

Sí TESTIGO devuelve "cierto" en la instrucción que está dentro del ciclo, entonces ha descubierto que x es una raíz cuadrada no trivial de 1, módulo n, dado que tenemos que x no es congruente con $\pm 1 \mod n$, y sin embargo $x^2 \equiv 1 \mod n$. Existe un corolario que dice que sólo sí n es compuesto puede existir una raíz cuadrada no trivial de 1 módulo n, así que una demostración de que x es una raíz cuadrada no trivial de 1 módulo n es una prueba de que n es compuesto.

Esto completa nuestra prueba de la validez de TESTIGO. Sí la invocación de TESTIGO (a, n) devuelve "cierto", entonces n es realmente compuesto y puede ser fácilmente determinada una prueba de que n es compuesto a partir de a y n. El procedimiento MILLER - RABIN es una búsqueda probabilística de que n es compuesto. Sí alguno de los a escogidos es un testigo de la composición de n entonces MILLER -RABIN devuelve "compuesto". Tal respuesta es siempre correcta, debido a la validez de TESTIGO. Sí no son encontrados testigos en los t intentos, MILLER - RABIN asume que esto es porque no hay testigos, y n es por lo tanto primo. Esta respuesta es probablemente correcta si t es lo suficientemente grande, sin embargo hay una pequeña probabilidad de que el procedimiento haya sido desafortunado en las selecciones de a y de que existan testigos aunque no haya sido escogido ninguno.

Implementación hardware

En el algoritmo "*Búsqueda aleatoria de un primo usando la prueba de Miller – Rabin*" no introducimos la búsqueda incremental, la cual consiste en aumentar el número *n* sí falla, en dos unidades en lugar de generar otro al azar. La búsqueda incremental se basa en el teorema de los números primos, el cual dice que la proporción de enteros positivos $\leq n$ que son primos es aproximadamente 1 /1n *n*. Dado que la mitad de todos los enteros $\leq n$ son pares, la proporción de enteros impares $\leq n$ que son primos es aproximadamente 2 /1n *n*. Esto tiene ciertas ventajas que se traducen en la aceleración del proceso de búsqueda de *n*.

Escribamos de nuevo el algoritmo introduciendo esta modificación para observar cómo lo llevaremos al hardware:

1. Generar un número aleatorio n, de tamaño igual o menor a k bits.
2. Poner a 1 el bit más significativo - garantizamos que el número es de - y el menos significativo – debe ser impar para poder ser primo-.
3. Para $i \leftarrow 1$ hasta t haga:

   Generar un número aleatorio a entre 1 y $n-1$

   Si WITNESS (a, n) $\Rightarrow$ COMPUESTO

doi:http://doi.org/10.23850/23899573.1061

WITNESS

Sea $b_k, b_{k-1}, \ldots, b_0$ la representación binaria de $n-1$

$d \leftarrow 1$

Para $i \leftarrow k$ hasta 0 haga:

$x \leftarrow d$

$d \leftarrow (d \cdot d) \mod n$

Sí $d = 1$ y $x \neq 1$ y $x \neq n-1$

$\Rightarrow$ CIERTO

Sí $b_i = 1 \Rightarrow d \leftarrow (d \cdot a) \mod n$

Sí $d \neq 1 \Rightarrow$ CIERTO

FALSO

PRIMO

4. Sí la prueba falla, incrementar n en dos unidades y volver al paso 3.

Todo el anterior algoritmo lo podemos escribir en lenguaje VHDL y presentarlo en un diagrama de bloques como muestran la Figura 1 y la Figura 2.

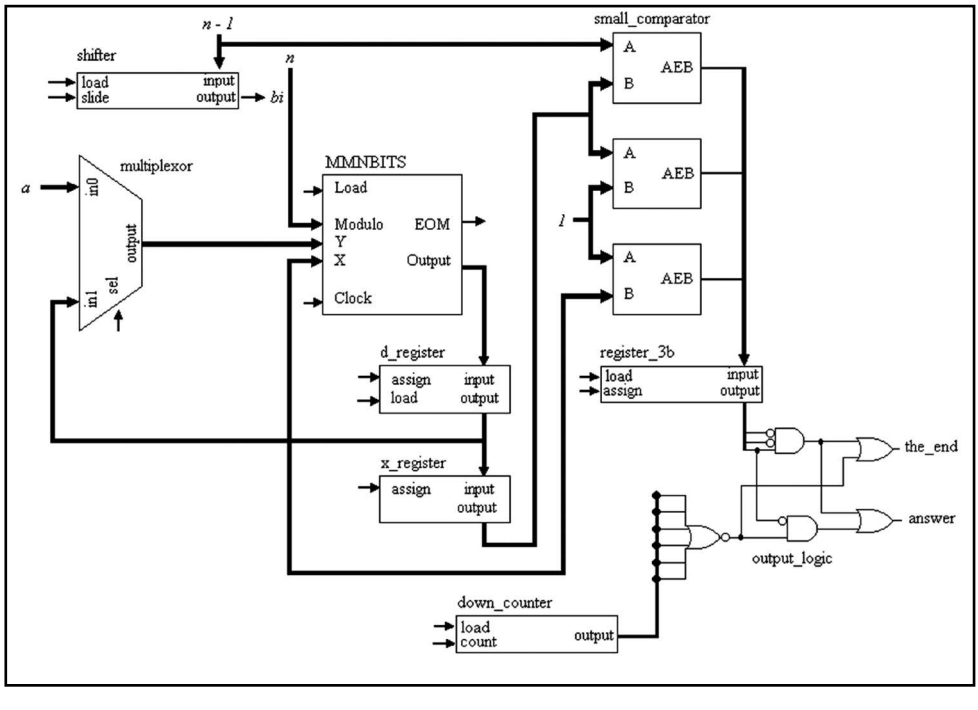

Figura 1. Sistema WITNESS completo

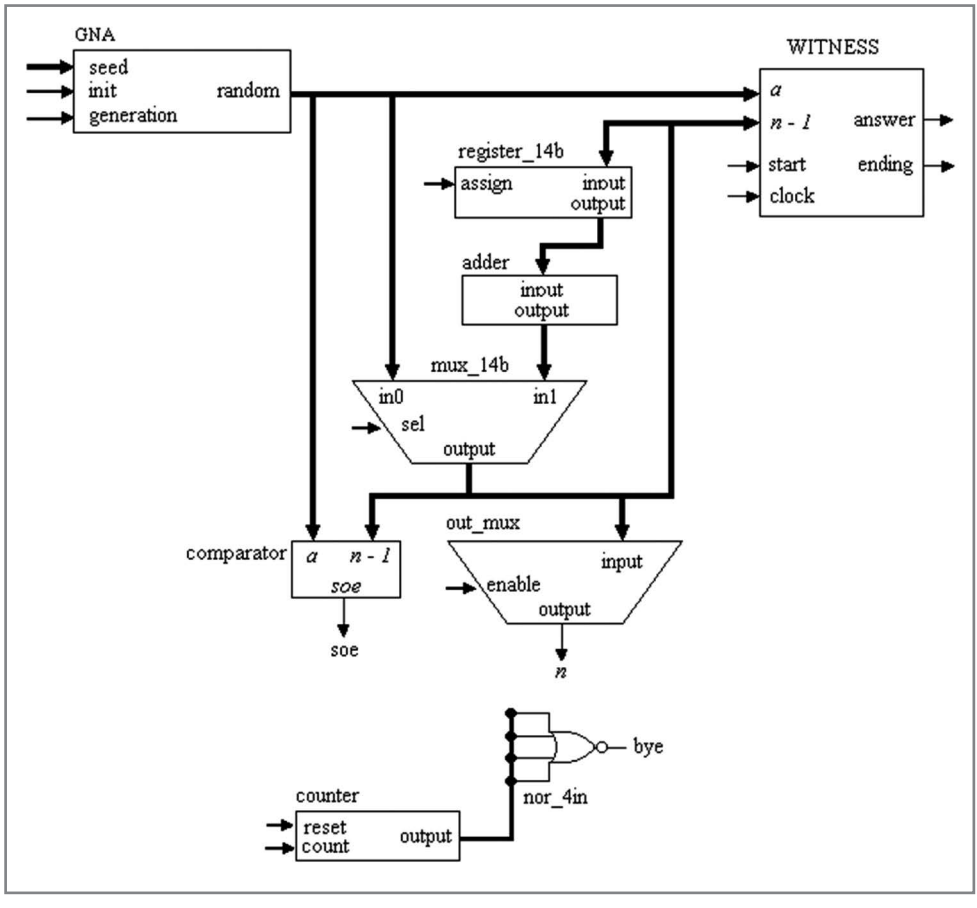

Figura 2. El sistema total (WITNESS se muestra como un bloque por razones de espacio)

RESULTADOS Y DISCUSIÓN

Teniendo el sistema completo sólo nos resta simular para observar los diferentes tiempos de búsqueda de un número primo aleatorio. Antes de mostrar los resultados obtenidos hagamos un cálculo aproximado de cuánto tiempo promedio empleará el sistema en encontrar un número primo aleatorio para diferente número de bits.

El sistema básicamente lo que hace es tomar un número aleatorio n de k bits, luego busca un número a menor que n y trata de calcular $a^{n-1} \bmod n$, la exponenciación puede que se realice toda pero también puede que no, es decir sí el sistema encuentra una raíz no trivial, de inmediato suspende la exponenciación y descarta el número por ser compuesto. En general la mayor parte del tiempo el sistema realiza toda la exponenciación para poder descartar el número. Ahora, el sistema está diseñado para que pruebe el número 5 veces, o sea que realiza 5 exponenciaciones sí ha encontrado un número probablemente primo.

doi:http://doi.org/10.23850/23899573.1061

La Tabla 4 lista la cantidad de ciclos empleados por el multiplicador modular para realizar una sola multiplicación para diferente número de bits [3].

Tabla 4
Ciclos empleados por el multiplicador modular para una multiplicación

bits	ciclos
16	12
32	20
64	36

Fuente: Elaboración propia

En general sí k es el número de bits y $k \geq 8$ entonces el número de ciclos empleados por el multiplicador modular para realizar una multiplicación es:

$$6 + \sum_{i=3}^{\log_2 k} 2^{i-2}$$

Por otra parte la cantidad de multiplicaciones que se realicen en una exponenciación depende de la cantidad de unos que posea el exponente, es decir $n-1$. Se realizan k multiplicaciones más 1 por cada uno que posea el exponente. Sí consideramos que el número con menos unos que se puede presentar es el número que solo tenga un 1 en el MSB y que el número que más unos puede tener es el que posea todos sus bits a 1 excepto –obviamente- el LSB que siempre es cero ya que estamos hablando de $n-1$, podemos deducir entonces que la cantidad de multiplicaciones promedio que se realizan en una exponenciación es:

$$\textit{cantidad de multiplicaciones} = \frac{(k+1)+(k+k-1)}{2} \quad \text{ó}$$

$$\textit{cantidad de multiplicaciones} = \tfrac{3}{2}k$$

Solo resta saber cuántas exponenciaciones debemos hacer antes de encontrar un probable primo. La respuesta la podemos encontrar fácilmente sí recordamos dos cosas: (1) la probabilidad de que Miller-Rabin descarte un número por ser compuesto en el primer intento es del 75%, por eso podemos asumir que los números que no sean primos serán descartados con solo una exponenciación; (2) según el teorema de los números primos, dichos números están espaciados con un intervalo de 1/1n es decir encontraremos un número primo cada 1n n números, donde n es el número alrededor del cual estamos buscando. También sabemos que para que n sea primo, n debe ser impar, o sea que la cantidad a descartar se reduce a la mitad, es decir $[(\ln n)/2]-1$ o expresado en la cantidad de bits k, [(1n 2^k)/2]−1. La cifra anterior más las 5 veces que el sistema prueba el número ($t=5$) dan como resultado el número promedio de exponenciaciones que se realizan.

Con todas estas deducciones podemos decir que el sistema empleará un tiempo igual a la cantidad de exponenciaciones promedio multiplicada por la cantidad de multiplicaciones promedio para cada exponenciación multiplicado por la cantidad de ciclos necesarios para cada multiplicación multiplicado por el periodo mínimo (ciclo) que se necesita en el sistema, periodo que está en función del número de bits k y que se lista en la tabla 5

Tabla 5

Mínimo periodo del sistema total necesitado en función del número de bits.

bits	T(ns)
16	83.6
32	175.9
64	364.9

Fuente: Elaboración propia

Así, para k bits tenemos que el tiempo de búsqueda es:

$$tiempo = \left\{\left[\left(\ln 2^k\right)/2\right]-1+5\right\}\times\frac{3}{2}k\times\left(6+\sum_{i=3}^{\log_2 k}2^{i-2}\right)\times periodo \quad para\ k\geq 8$$

Sí k es igual a **16**, **32** y **64** bits los tiempos serán respectivamente: ***229.82 µs***, ***2.548 ms*** y ***33.016 ms***.

Se realizaron simulaciones para 16 y 32 bits, para 64 bits se intentó realizar simulaciones pero el tiempo de la simulación fue alto.

Tabla 6

Números encontrados y tiempos de búsqueda simulados para números de 16 bits

Semilla	Número encontrado	tiempo (µs)
1	32779	211.14
3449	39671	208.80
10348	53479	321.66
13797	60373	313.80
17246	34499	180.21
20695	41399	255.61
24144	48299	267.82
27594	55201	302.10
31043	62099	324.00
34492	36187	194.59
37941	43093	330.69

Continuación tabla 6

Semilla	Número encontrado	tiempo (µs)
41391	50047	340.38
44840	56963	194.59
48289	63857	168.84
55187	44819	260.96
58637	51769	157.13
62086	58661	157.13
65535	65497	261.80

Fuente: Elaboración propia

Tabla 7

Números encontrados y tiempos de búsqueda simulados para números de 32 bits.

Semilla	Número encontrado	tiempo (ms)
1	2147 483 659	1.218
0D79 435E	2599 585 477	1.394
1AF2 86BC	3051 687 307	2.441
286B CA1A	3503 789 129	2.574
35B5 0D78	3955 890 947	2.479
435E 50D6	2260 509 113	1.796
50D7 9434	2712 610 939	2.417
5E50 D792	3164 712 769	3.500
6BCA 1AF0	3616 814 573	1.893
7943 5E4E	4068 916 403	2.882
86BC A1AC	2373 534 637	2.206
9435 E50A	2825 636 573	1.858
A1AF 2868	3277 738 103	10.221
AF28 6BC6	3729 839 999	6.343
BCA1 AF24	4181 941 937	4.382
CA1A F282	2486 560 217	3.047
D794 35E0	2938 661 651	1.993
E50D 793E	3390 763 709	1.037
F286 BC9C	3842 865 673	1.761
FFFF FFFA	4294 967 111	3.446

Fuente: Elaboración propia

doi:http://doi.org/10.23850/23899573.1061

Tabla 8

Números encontrados y tiempos de búsqueda simulados para números de 64 bits.

Semilla	*Número encontrado*	*tiempo (ms)*
1	8000 0000 0000 001D	17.078
435E 50D7 9435 E509	86BC 8003 2800 026B	66.810

Fuente: Elaboración propia

Los tiempos promedio de las anteriores tablas son presentados en la Tabla 9.

Tabla 9

Tiempo de búsqueda promedio para diferente número de bits.

bits	*Tiempo promedio*
16	247.29 µs
32	2.232 ms
64	No calculado

Fuente: Elaboración propia

Observemos que los tiempos obtenidos en las simulaciones son bastante cercanos a los calculados anteriormente por deducción.

CONCLUSIONES

El trabajo realizado hace parte de un intento inicial por desarrollar el algoritmo RSA en su totalidad para una posterior implementación en arquitecturas reconfigurables. Es decir, se tomó RSA y se dividió en cuatro partes: la multiplicación modular, la exponenciación modular, el cálculo del inverso multiplicativo y la generación de números primos, restando solamente un trabajo final en el cual se integrarían los mencionados anteriormente y donde se dé un verdadero cuerpo al sistema total.

RSA es el algoritmo más popular en la criptografía de clave pública y uno de los más usados, reconocimiento que no ha sido gratis sino que lo ha ganado a través de veinte años sosteniéndose ante ataques de todo tipo como los que puede realizar una comunidad tan grande y tan experta como Internet. Sin embargo RSA no es perfecto y tiene su principal dolencia en la cantidad de bits necesitados por la clave para hacer de éste un sistema seguro. En la actualidad RSA recomienda una longitud de clave no menor de 1024 bits y para casos militares de 2048 bits. Es precisamente esta característica la que hace a RSA un sistema "pesado" y poco recomendable para ser implementado en FPGAs. En el trabajo realizado con la generación de números primos se duplicó el número de celdas en la FPGA cuando se duplicó el número de bits del sistema. Lastimosamente no se pudo hacer pruebas para una cantidad de bits superior a 32, pero la impresión que dejaron las simulaciones acerca del espacio y del retardo de tiempo no es muy prometedora como para intentar extenderlo a un número de bits que brinde una seguridad aceptable. Recordemos que la generación de números primos es sólo una parte del algoritmo RSA y que aún faltarían otros cuatro trabajos para implementar totalmente RSA.

Volviendo con el tema del retardo, tengamos en cuenta que la criptografía tiene que ver principalmente con el intercambio de

información, y que la información demanda la mayor velocidad posible de transmisión. Es por esta razón que a la hora de intentar implementar un algoritmo criptográfico debemos buscar que éste sea lo más rápido posible sin olvidarnos de que debe ser poco costoso, hablando en términos de hardware. Para poder cumplir con estas dos exigencias debemos pensar en un cambio de algoritmo, uno que demande un menor número de bits en su clave para obtener una buena seguridad. Por otra parte el conocimiento profundo, detallado, milimétrico, de la arquitectura en la cual se desea implementar (en este caso FPGAs) es fundamental para cumplir los objetivos de ahorro de espacio e incremento de la velocidad de ejecución. Sí bien es cierto el amarrarse con una familia o marca específica de FPGA hace perder portabilidad, la portabilidad no es un objetivo principal ni es algo tan importante como la velocidad del sistema o el tamaño de la tarjeta donde se encuentre.

Por último, la principal tarea que debemos realizar antes de diseñar, es descartar las opciones que no nos sirven y que podemos ver a primera vista o después de un pequeño análisis. Como ejemplo tomemos las pruebas probabilísticas de primalidad, en principio podemos pensar en realizar el diseño hardware de todas y luego escoger cual es la mejor, pero sí nos damos cuenta que son algoritmos y que un algoritmo que podemos descartar frente a otro siempre será peor que el primero tanto en software como en hardware, entonces podremos ahorrar tiempo y esfuerzo.

REFERENCIAS

[1] MENEZES, Alfred J.; VAN OORSCHOT, Paul C.; y VANSTONE, Scott A. (1996) Handbook of Applied Cryptography. CRC Press.

[2] CORMEN, T.; LEISERSON, C.; y RIVEST, R. (1990) Introduction to Algorithms. Cambridge, MA; MIT Press.

[3] PALACIOS, Rubén Darío. (2001) Diseño e implementación en FPGAs de un multiplicador modular de 32 bits. Trabajo de grado en Ingeniería Electrónica. Universidad del Valle.

[4] NELSON, Victor P.; NAGLE, H. Troy; CARROL, Hill D.; IRWIN, J. David. (1996) Análisis y Diseño de Circuitos Lógicos Digitales. Prentice-Hall.

[5] WAKERLY, John F. (1992) Diseño Digital Principios y Prácticas. Prentice-Hall.

[6] Página web de RSA. www.rsa.com.