DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA
ÁREA DE TECNOLOGÍA ELECTRÓNICA

PROYECTO FIN DE CARRERA

"SISTEMA DE ENCRIPTACIÓN RSA"

INGENIERÍA DE TELECOMUNICACIÓN
ESCUELA SUPERIOR DE INGENIEROS
UNIVERSIDAD DE SEVILLA

TUTOR: JORGE CHÁVEZ ORZÁEZ
AUTOR: FRANCISCO JAVIER PÉREZ GONZÁLEZ

ÍNDICE

1. INTRODUCCIÓN A LA CRIPTOGRAFÍA.

La palabra criptografía proviene del griego *kryptos*, que significa esconder y *gráphein*, escribir, es decir, escritura escondida. La criptografía ha sido usada a través de los años para mandar mensajes confidenciales cuyo propósito es que sólo las personas autorizadas puedan entender el mensaje.

La criptografía es un área dentro de la criptología, vocablo griego que podemos traducir como estudio o ciencia de lo escondido. Una primera división inmediata de la criptología podría ser:

- Criptografía: se encarga de la generación de técnicas basadas en una clave secreta, para encriptar los datos, de manera que sólo la persona que tenga acceso a dicha clave podrá interpretarlos.
- Análisis criptográfico: desarrolla técnicas para desencriptar los datos sin conocimiento de la clave utilizada.

Ambas disciplinas están fuertemente relacionadas, el diseño de un buen algoritmo de encriptado se basa en los métodos y herramientas que están al alcance de los analistas criptográficos. El responsable de la implementación de medidas de seguridad debe tener conocimiento y estar prevenido de los potenciales ataques de un intruso, además, un buen ataque a un algoritmo de encriptado debe basarse en un profundo conocimiento de los potenciales ataques de un intruso.

La criptografía clásica se caracterizaba porque tanto los procedimientos de cifrado como las claves son secretas, sin embargo en la criptografía moderna los procedimientos son públicos, y el único secreto es la clave. Además, para que un operador criptográfico sea fiable ha de estar expuesto al constante estudio de los analistas y resistir sus ataques. La criptografía actual se inicia en la segunda mitad de la década de los años 70. No es hasta la invención del sistema conocido como **DES** (**D**ata **E**ncryption **S**tandard) en 1976 que se da a conocer mas ampliamente, principalmente en el mundo industrial y comercial. Posteriormente con el sistema **RSA** (**R**ivest, **S**hamir, **A**dleman) en 1978, se abre el comienzo de la criptografía en un gran rango de

aplicaciones: en transmisiones militares, en transacciones financieras, en comunicación de satélite, en redes de computadoras, en líneas telefónicas, en transmisiones de televisión, etcétera.

La criptografía se divide en dos grandes ramas, la criptografía de clave privada o simétrica y la criptografía de clave pública o asimétrica, DES pertenece al primer grupo y RSA al segundo.

Para poder entender un poco de la criptografía, es tiempo de plantear que tipo de problemas resuelve ésta. Los principales problemas de seguridad que resuelve la criptografía son: la privacidad, la integridad, la autenticación y el no rechazo.

La privacidad se refiere a que la información sólo pueda ser leída por personas autorizadas. En la comunicación por Internet es muy difícil estar seguros que la comunicación es privada, ya que no se tiene control de la línea de comunicación. Por lo tanto al cifrar (esconder) la información cualquier intercepción no autorizada no podrá entender la información. Esto es posible si se usan técnicas criptográficas, en particular la privacidad se logra si se cifra el mensaje con un método simétrico.

La integridad se refiere a que la información no pueda ser alterada en el transcurso de ser enviada. En Internet las compras se pueden hacer desde dos ciudades muy distantes, la información tiene necesariamente que viajar por una línea de transmisión de la cual no se tiene control. La integridad también se puede solucionar con técnicas criptográficas, particularmente con procesos simétricos o asimétricos.

La autenticidad se refiere a que se pueda confirmar que el mensaje recibido haya sido mandado por quien dice lo mando o que el mensaje recibido es el que se esperaba. Las técnicas necesarias para poder verificar la autenticidad tanto de personas como de mensajes usan quizá la más conocida aplicación de la criptografía asimétrica que es la firma digital, de algún modo ésta reemplaza a la firma autógrafa que se usa comúnmente. Para autenticar mensajes se usa criptografía simétrica.

El no rechazo se refiere a que no se pueda negar la autoría de un mensaje enviado.

Cuando se diseña un sistema de seguridad, una gran cantidad de problemas puedes ser evitados si se puede comprobar autenticidad, garantizar privacidad, asegurar integridad y evitar el no-rechazo.

1.1. SEGURIDAD EN CRIPTOGRAFÍA.

La criptografía sólo se encarga de poner barreras más o menos difíciles de franquear para poder garantizar un cierto nivel de confianza, ya que la seguridad absoluta en comunicaciones digitales es realmente una utopía.

Algunos aspectos a tener en cuenta en el concepto de seguridad son el conocer de qué o quiénes debemos protegernos y analizar los medios de seguridad que tenemos a nuestro alcance.

Respecto al primer punto podemos analizar que si nos protegemos de la lectura o copiado de datos estamos primando la seguridad y confidencialidad, mientras que si nos protegemos del uso ilegal de una red o de la integridad de los datos nos referimos a la identificación y validación del emisor de la información, es decir, la autentificación del emisor.

En cuanto a los medios de seguridad, primeramente cabe destacar que éstos no surgirán efecto sin una apropiada política de organización, es decir, si algunas medidas resultan confusas para el usuario puede existir el riesgo del descuido por parte de éste, por lo que siempre hay que pensar en el factor humano cuando hablamos de seguridad a cualquier nivel. Partiendo de esto podemos señalar que cualquier sistema criptográfico puede ser vulnerable si no está bien diseñado y sus posibles debilidades matemáticas no han sido analizadas, e incluso un sistema puede ser vulnerable aun a pesar de utilizar un operador criptográfico seguro.

Podemos diferenciar los siguientes niveles de seguridad ofrecida por la criptografía:

- Seguridad incondicional: Es la ofrecida por los métodos de cifrado para los que se puede demostrar que no existe la posibilidad de predecir la clave secreta de cifrado, es decir, que el conocimiento del cifrado no aporta ninguna información adicional que permita obtener el texto en claro de forma más sencilla que en el caso en que no se conozca. Estos tipos de cifrados se denominan perfectos, como es el caso del cifrado de Vernan o cifrado con clave de un solo uso.

- Seguridad computacional: Es la ofrecida por aquellos métodos de cifrado tales que no existe capacidad de cálculo suficiente en el universo para obtener su clave secreta asociada. La seguridad computacional está definida de acuerdo con la teoría de la complejidad, los conocimientos matemáticos actuales y las prestaciones de los ordenadores. El sistema RSA junto al resto de clave pública están incluidos en este apartado.

- Seguridad probable: Es la de aquellos métodos de cifrado que aun sin estar basados en principios matemáticos de seguridad demostrable no han podido ser violados pese a los continuados esfuerzos para conseguirlo. Estos procedimientos suelen utilizar técnicas especiales para el barajado de la información mediante operaciones lineales y no lineales de bits, de forma que el esfuerzo computacional necesario para romper el sistema se presupone razonablemente superior a la capacidad de cálculo del atacante.

- Seguridad condicional: Es la ofrecida por todos los demás métodos, es decir, por aquellos procedimientos de cifrado diseñados con fines específicos y para los que la dificultad de violación es siempre muy superior a la supuesta capacidad de análisis de un eventual atacante.

1.2. CRIPTOANÁLISIS ELEMENTAL.

Un criptosistema se puede atacar de muchas formas, la más directa sería la que hace uso únicamente del análisis del mensaje cifrado o criptograma, que es lo que se conoce como un análisis pasivo. También se pueden producir ataques activos, es decir, ataques apoyados en cierto conocimiento adicional o en cierto grado de intervención.

Los posible ataques, citados de mayor a menor dificultad, serían:

1. Sólo se conoce el criptograma.
2. Sólo se conoce el criptograma, pero éste va salpicado con partes en claro sin cifrar.
3. Se conocen varios criptogramas diferentes correspondientes al mismo texto claro, cifrados con claves diferentes.
4. Se conocen el criptograma y el texto claro correspondiente. Incluye el caso de que no se conozca enteramente el texto claro, pero sí parte de él, o bien que se conozcan palabras probables.
5. Se conoce el texto descifrado correspondiente a un criptograma elegido por el criptoanalista.
6. Se conoce el criptograma correspondiente a un texto claro escogido por el criptoanalista.
7. Se conoce el texto descifrado correspondiente a un criptograma elegido de forma adaptativa por el criptoanalista.
8. Se conoce el criptograma correspondiente a un texto claro escogido de forma adaptativa por el criptoanalista en función de los análisis previos.
9. Se conoce la clave o al menos se puede limitar el espacio de claves posibles.

Todos estos casos pueden estar modulados por el hecho de que se conozca o no el criptosistema en uso, cuando el algoritmo es conocido hay un ataque posible, denominado de fuerza bruta, y que consiste en probar todas las claves. Estas formas de ataque obligan a varias precauciones por parte del usuario para no dar facilidades al oponente:

- Cuando sea necesario repetir la transmisión de un mensaje cifrado se hará con la clave original, para evitar el ataque número 3.

- No se cifrará la información que ya es pública, para evitar el ataque número 4.
- No se enviará idéntica información en claro y en cifrado, aunque se haga por canales diferentes, para evitar el ataque número 4.
- No se enviarán en una misma comunicación partes en claro y en cifrado, para evitar los ataques 2 y 4.
- Se evitará enviar mensajes cifrados, referentes a mensajes en claro recibidos del oponente, para evitar el ataque número 6.
- Se elegirán las claves de forma aleatoria y carecerán de sentido, para no facilitar un ataque por fuerza bruta basándose en un diccionario reducido (ataque nº 9).
- Se procurará incorporar de alguna forma la fecha y hora de producción de un mensaje a la clave, lo que asegura de cierta forma el cambio de clave con cada mensaje.
- Las claves y algoritmos de cifrado han de ser secretos y conocidos por un número reducido de personas para evitar un ataque por fuerza bruta.
- Se cambiarán las claves con la mayor frecuencia posible y se tratará de evitar el uso de la misma clave con mensajes diferentes, para obligar al oponente que es capaz de romper el algoritmo recuperando la clave, a repetir el proceso de ataque con cada nuevo mensaje.

Una posibilidad de aumentar la protección de la información frente al criptoanálisis consiste en hacer una compresión previa de la información antes de proceder a su cifrado. Hay técnicas de compresión de archivos, basados en la Teoría de la Información, que permiten codificar de forma eficaz la información de un mensaje, reduciendo su redundancia y alcanzando una apariencia de un ruido aleatorio.

Debemos tener cuidado con el tipo de compresor que se emplea, actualmente hay muchos compresores comerciales de archivos en los distintos sistemas informáticos, casi todos basados en el algoritmo Ziv-Lempel, como el ZIP, ARC, LHA, etc. Estos compresores están pensados únicamente para ahorrar espacio de almacenamiento y no con fines criptográficos, por tanto están altamente formateados y tienen cabeceras standard que pueden emplearse para un ataque por texto claro conocido.

1.3. PROCEDIMIENTOS CLÁSICOS DE CIFRADO.

Dentro de la criptografía clásica aparecen dos procedimientos de cifrado básicos que se han ido repitiendo en épocas posteriores hasta llegar a nuestros días. Éstos son los procedimientos de sustitución y transposición.

El procedimiento de sustitución consiste en establecer una correspondencia entre las letras del alfabeto en el que está escrito el mensaje original y los elementos de otro conjunto, que puede ser el mismo o distinto alfabeto. De esta forma, cada letra del texto claro se sustituye por su símbolo correspondiente en la elaboración del criptograma. En recepción, el legítimo receptor, que conoce asimismo la correspondencia establecida, sustituye cada símbolo del criptograma por el símbolo correspondiente del alfabeto original, recuperando así la información inicial.

El procedimiento de transposición consiste en barajar los símbolos del mensaje original colocándolos en un orden distinto, de manera que el criptograma contenga los mismos elementos del testo claro, pero colocados de tal forma que resulten incomprensibles. En recepción, el legítimo receptor, con conocimiento de la transposición, recoloca los símbolos desordenados.

Todos los procedimientos criptográficos clásicos se fundamentan en uno u otro principio, o bien en una superposición de ambos.

Algunos ejemplos históricos de cifrados por sustitución son el cifrado de Vigenére (1586), en el que se utilizaba una clave secreta que se sumaba símbolo a símbolo en módulo al mensaje para obtener el criptograma, y el cifrado de Beaufort (1710), que es una variante del anterior con la particularidad de que se suma la clave con la inversa de cada símbolo del texto en claro. Podemos expresarlo en términos matemáticos de la forma:

$$Y_i = Z_i \oplus (-X_i) \pmod{26}$$

donde Y_i corresponde a la i-ésima letra del criptograma, X_i a la i-ésima letra del texto en claro, y Z_i a la i-ésima letra de la clave, además se realiza módulo 26 porque ese era el número de letras del alfabeto. Para la recepción el procedimiento se repite:

$$X_i = Z_i \oplus (-Y_i) = Z_i \oplus (-Z_i) \oplus X_i \pmod{26}$$

Se aprecia que tanto cifrado como descifrado se reducen a la misma operación, luego el dispositivo que lleva a efecto este proceso es el mismo tanto en emisión como en recepción. Este tipo de cifrado se conoce como cifrado recíproco o involutivo, donde si aplicamos dos veces la misma transformación obtenemos el texto original.

El cifrado de Vernam (1917) representa el caso límite del cifrado de Vigenére, emplea un alfabeto binario y tiene como novedad que la clave se utiliza solamente una vez (one_time pad). Este método fue utilizado durante la segunda guerra mundial por espías de diversas nacionalidades, a los que se les daba una secuencia binaria aleatoria con la recomendación de utilizarla para un único proceso de cifrado.

Posteriormente Shannon demostró teóricamente la seguridad total de este método mediante las condiciones de secreto perfecto partiendo de dos hipótesis básicas:

1. La clave secreta se utilizará solamente una vez, a diferencia de lo que sucedía en los métodos clásicos, en los que la clave era fija.
2. El enemigo criptoanalista tiene acceso sólo al criptograma, luego está limitado a un ataque sobré texto cifrado únicamente.

Un sistema criptográfico verifica las condiciones de secreto perfecto si el texto claro es estadísticamente independiente del criptograma, es decir, que la información sobre el texto claro aportada por el criptograma es nula. Asimismo, y basado en el concepto de entropía, Shannon determinó la menor cantidad de clave necesaria para que pudieran verificarse las condiciones de secreto perfecto. La longitud de la clave tiene que ser, al menos, tan larga como la longitud del texto claro, en el caso de igualdad tendríamos el caso del cifrado de Vernam.

Dentro del panorama criptográfico actual, el cifrado Vernam es el único procedimiento incondicionalmente seguro o con seguridad probada matemáticamente, pero presenta un inconveniente evidente, debido a que requiere un dígito de clave secreta por cada dígito de texto claro, y teniendo en cuenta las exigencias actuales de información a cifrar, el método resulta inviable para su aplicación habitual en criptografía. Queda reservado para aquellas circunstancias en las que se requiere unas condiciones máximas de seguridad pero un mínimo de información a proteger.

2. EL ALGORITMO RSA.

El sistema de cifrado asimétrico de clave pública RSA fue propuesto por R. L. Rivest, A. Shamir y l. Adleman en el año 1977 y posteriormente patentado por el MIT (Massachusetts Institute of Technology).

En los cifrados asimétricos o de clave pública, la clave de descifrado no se puede calcular a partir de la de cifrado.

En 1975, dos ingenieros electrónicos de la Universidad de Stanford, Whitfield Diffie y Martin Hellman, sugieren usar problemas computacionalmente irresolubles para el diseño de criptosistemas seguros. La idea consiste básicamente en encontrar un sistema de cifrado computacionalmente fácil (o al menos no difícil), de tal manera que el descifrado sea, por el contrario, computacionalmente irresoluble a menos que se conozca la clave. Para ello, hay que usar una transformación criptográfica Tk de fácil aplicación, pero de tal forma que sea muy difícil hallar la transformación inversa Tk' sin la clave de descifrado. Dicha función Tk es, desde el punto de vista computacional, no invertible sin cierta información adicional (clave de descifrado) y se llama función de una vía o función trampa.

En estos esquemas se utiliza una clave de cifrado (clave pública) k que determina la función trampa Tk, y una clave de descifrado (clave secreta o privada) que permite el cálculo de la inversa Tk'. Cualquier usuario puede cifrar usando la clave pública, pero sólo aquellos que conozcan la clave secreta pueden descifrar correctamente.

En consonancia con el espíritu de la criptografía moderna, y tal cómo sucedía en los sistemas simétricos, los algoritmos de cifrado y de descifrado son públicos, por lo que la seguridad del sistema se basa únicamente en la clave de descifrado. Según Diffie y Hellman, todo algoritmo de clave pública debe cumplir las siguientes propiedades de complejidad computacional:

1. Cualquier usuario puede calcular sus propias claves pública y privada en tiempo polinomial.
2. El emisor puede cifrar su mensaje con la clave pública del receptor en tiempo polinomial.

3. El receptor puede descifrar el criptograma con la clave privada en un tiempo polinomial.
4. El criptoanalista que intente averiguar la clave privada mediante la pública se encontrará con un problema intratable.
5. El criptoanalista que intente descifrar un criptograma teniendo la clave pública se encontrará con un problema intratable.

La criptografía de clave pública se basa principalmente en las siguientes características:

- Cada usuario obtiene un par de claves, una pública y otra privada.
- La clave pública de cada persona será difundida de tal manera que puede ser conocida por cualquiera. Un usuario A que desee enviar un mensaje a un usuario B utilizará la clave pública de B para cifrar el mensaje que desea enviarle además de su propia clave secreta (la de A).
- La clave secreta se mantiene en secreto por el propietario de la misma. Con esta clave se podrán descifrar los mensajes que lleguen. Así con el ejemplo anterior, B utilizaría la clave pública de A y su propia clave secreta para descifrar el mensaje enviado por A.
- Con el esquema anterior, se puede observar que únicamente las claves públicas serán difundidas y podrán ser conocidas por cualquiera. Las claves privadas no son transmitidas o compartidas.
- La base de este criptosistema está en que no se puede obtener la clave privada del usuario receptor a partir de su clave pública.

Es más difícil diseñar un sistema de clave pública seguro contra un ataque con texto original escogido que un sistema de clave secreta seguro frente al mismo tipo de ataque. En la construcción de criptosistemas se pueden observar diferencias entre los algoritmos para sistemas simétricos y los usados en clave publica. En primer lugar, existen mayores restricciones de diseño para un algoritmo asimétrico que para uno simétrico, debido a que la clave pública representa información adicional que potencialmente un enemigo puede usar para llevar a cabo el criptoanálisis. Normalmente, el algoritmo de clave pública basa su seguridad en la dificultad de resolver algún problema matemático

conocido, mientras que algunos algoritmos simétricos, cómo el DES, se diseñan de tal manera que las ecuaciones matemáticas que los describen son tan complejas que no son resolubles de forma analítica.

En segundo lugar, existen grandes diferencias en la generación de claves. En los algoritmos simétricos, en los que el conocimiento de la clave de cifrado es equivalente al de la de descifrado, y viceversa, la clave se puede seleccionar de forma aleatoria. Sin embargo, en los algoritmos asimétricos, cómo la relación entre clave de cifrado y de descifrado no es pública, se necesita un procedimiento para calcular la pública a partir de la clave privada que sea computacionalmente eficiente y tal que el cálculo inverso sea imposible de realizar.

Hace algunos años, este tipo de sistemas no parecía tener ninguna ventaja en el mundo criptográfico, porque tradicionalmente la criptografía se usaba sólo con propósitos militares y diplomáticos, y en estos casos el grupo de usuarios es lo suficientemente pequeño cómo para compartir un sistema de claves. Sin embargo, en la actualidad, las aplicaciones de la criptografía han aumentado progresivamente, hasta alcanzar muchas otras áreas donde los sistemas de comunicación tienen un papel vital. Cada vez con mayor frecuencia se pueden encontrar grandes redes de usuarios en las que es necesario que dos cualesquiera sean capaces de mantener secretas sus comunicaciones entre sí. En estos casos, el intercambio continuo de claves no es una solución muy eficiente.

Por otro lado, hay que resaltar la ventaja que representa en los sistemas asimétricos la posibilidad de iniciar comunicaciones secretas sin haber tenido ningún contacto previo.

A continuación, se nombrarán algunos de los sistemas de clave pública que han tenido más trascendencia.

- **Sistema RSA.** Se basa en el hecho de que no existe una forma eficiente de factorizar números que sean productos de dos grandes primos.
- **Sistema de Rabin.** Se basa también en la factorizacion.
- **Sistema de ElGamal.** Se basa en el problema del logaritmo discreto.
- **Sistema de Merkle-Hellman.** Esta basado en el problema de la mochila.

- **Sistema de McEliece.** Se basa en la teoría de la codificación algebraica, utilizando el hecho de que la decodificación de un código lineal general es un problema NP-completo.
- **Sistemas basados en curvas elípticas.** En 1985, la teoría de las curvas elípticas encontró de la mano de Miller aplicación en la criptografía. La razón fundamental que lo motivó fue que las curvas elípticas definidas sobre cuerpos finitos proporcionan grupos finitos abelianos, donde los cálculos se efectúan con la eficiencia que requiere un criptosistema, y donde el cálculo de logaritmos es aún más difícil que en los cuerpos finitos. Además, existe mayor facilidad para escoger una curva elíptica que para encontrar un cuerpo finito, lo que da una ventaja más frente a su predecesor, el sistema de ElGamal.

De todos los sistemas de clave pública mencionados, el sistema RSA es uno de los métodos de cifrado más utilizados hoy en día. Es uno de los sistemas más seguros de la actualidad, pero su principal problema radica en su incapacidad, debido a su baja velocidad para encriptar y/o decriptar grandes volúmenes de datos.

En este capítulo describiremos las características principales de los sistemas RSA, tanto su descripción interna de funcionamiento, como los posibles ataques a dichos sistemas.

2.1. DESCRIPCIÓN DEL ALGORITMO RSA.

Sean p y q dos números primos grandes y N = pq su producto. Sea $\phi(N)$ la función de Euler de N, indicadora del número de valores enteros menores que N que son relativamente primos con N. En este caso particular, la función $\phi(N)$ viene dada por el producto:

$$\phi(N) = (p - 1)(q - 1)$$

En estas condiciones, sea e un valor entero aleatorio relativamente primo con $\phi(N)$ tal que $1 < e < N$ y sea d otro entero tal que verifica la congruencia:

$$ed \equiv 1 \pmod{\phi(N)}$$

es decir, que:

$$ed = k\phi(N) + 1$$

con k valor entero. En estas condiciones, para cualquier valor entero M se verifica que si $C \equiv M^{e} \pmod{N}$ entonces $M \equiv C^{d} \pmod{N}$, es decir que:

$$M^{ed} \equiv M \pmod{N}$$

Para demostrarlo se utilizará el teorema chino de los restos congruentes. Dicho teorema asegura que la congruencia anterior se verifica si y sólo si se verifican a su vez las dadas por:

$$M^{ed} \equiv M \pmod{p}$$

$$M^{ed} \equiv M \pmod{q}$$

lo cual es trivial por el teorema de Fermat:

$$M^{ed} \equiv M^{k\phi(N)+1} \equiv M^{k(p-1)(q-1)+1} \equiv M^{k(p-1)(q-1)}\ M \equiv M \pmod{p}$$

$$M^{ed} \equiv M^{k\phi(N)+1} \equiv M^{k(p-1)(q-1)+1} \equiv M^{k(p-1)(q-1)}\ M \equiv M \pmod{q}$$

En el caso de que M no sea relativamente primo con N debe ser múltiplo de p o múltiplo de q, con lo que las congruencias anteriores se verifican igualmente de forma trivial. Esta demostración es la base del sistema de cifrado RSA. Dicho sistema esta

constituido por las claves N, e y d, respectivamente módulo de trabajo y exponentes de cifrado y descifrado. El valor de N es público así como el de uno de los dos exponentes, mientras que el otro permanece secreto.

Para realizar el cifrado C de un mensaje en claro M tal que $1 < M < N$, el mensaje se eleva a la potencia e y el resultado se reduce al módulo N. Es decir:

$$C = M^{e} \pmod{N}$$

La recuperación del mensaje en claro M se lleva a cabo elevando el mensaje cifrado C a la potencia d (clave pareja de e) y reduciendo la exponenciación módulo N de acuerdo con la congruencia:

$$M \equiv C^{d} \pmod{N}$$

Los números primos p y q son secretos, constituyendo la trampa del sistema. El cálculo de la clave secreta d es sencillo conocida la trampa, mientras que sin el conocimiento de los factores primos p y q, el cálculo de la clave secreta es equivalente al cálculo de la factorización de la clave pública N. Aquí es donde radica la seguridad del método RSA, puesto que el problema de la factorización de un número compuesto N, producto de dos factores primos, tiene una complejidad exponencial dada por la razón:

$$e^{\sqrt{\ln(N)\ln\ln(N)}}$$

Por esta razón se requiere la utilización de números primos muy grandes. A su vez, aunque el sistema de cifrado RSA funciona para cualquier valor entero M, es conveniente que M sea relativamente primo con la clave pública N, puesto que de lo contrario la factorización de N es trivial. A continuación se muestra una tabla en la que se indica el tiempo necesario para calcular la factorización con un ordenador de $2*10^{8}$ instrucciones por segundo:

Nº BITS	DIAS
60	$1.7*10^{-8}$
120	$1.5*10^{-5}$
200	$1.6*10^{-2}$
256	$1*10^{0}$
283	$5.3*10^{0}$
363	$9*10^{2}$
502	$2.4*10^{6}$

Tabla 2.1. Tiempo de factorización de N en función del nº de bits.

Como se aprecia en la tabla anterior se tardaría cientos de años en factorizar un número de 500 bits aun utilizando ordenadores de hasta $2*10^{12}$ instrucciones por segundo. Actualmente, se recomienda utilizar claves públicas N del orden de los mil bits.

2.2. DISEÑO DEL SISTEMA RSA.

En el diseño del sistema de clave pública RSA hay que tener en cuenta ciertas condiciones a la hora de elegir los parámetros del mismo.

En cuanto a la elección de los números primos p y q que forman la clave pública N, la primera condición que deben satisfacer es la de no estar demasiado próximo el uno del otro. En caso contrario, es decir, si $p \cong q$ y suponemos por ejemplo que p > q se cumpliría que (p – q)/2 es un entero pequeño y (p + q)/2 es ligeramente superior a $\sqrt{N}$. Además, se verifica la relación dada por:

$$\frac{(p+q)^2}{4} - N = \frac{(p-q)^2}{4}$$

En estas condiciones, una forma de factorizar <N es elegir aleatoriamente valores enteros $x>\sqrt{N}$ hasta que se encuentre uno tal que (x^2 – N) es un cuadrado perfecto. Entonces, si $x^2 - N = y^2$, es sencillo verificar que p = x + y, y que q = x – y. Por esta razón, para que este procedimiento de factorización resulte impracticable es aconsejable que las longitudes de los números primos p y q difieran en unos pocos bits.

Otra condición necesaria que deben satisfacer los números primos p y q es que el mínimo común múltiplo de los valores (p – 1) y (q – 1) sea lo más elevado posible. El objeto de la misma es asegurar que el número de claves secretas parejas de una determinada clave pública sea el mínimo posible. Para verlo, sea γ el mínimo común múltiplo de (p – 1) y (q – 1) y sea e una clave pública con 1 < e < N y relativamente prima con ϕ(N). Asimismo, sea $d_\gamma = e^{-1} \pmod{\gamma}$.

La clave pública e tiene λ claves parejas d_i con $1 < d_i < N$ tales que:

$$d_i = d_g + ig$$

para i = 0, 1, ..., λ, con $\lambda = \lfloor (N - d_g)/g \rfloor$.

Se puede observar que (e,d_i) son claves parejas. Para ello es necesario que todo valor entero M verifique la congruencia dada por:

$$M^{ed_i} \equiv M \pmod{N}$$

Se puede apreciar que siempre se cumplirá $l \geq 1$, ya que $N > 2g$, por lo que siempre existirán al meno dos claves parejas de una dada.

En lo referente a la elección de la clave pública e hay que mencionar que dicha clave esta íntimamente relacionada con el número de mensajes que no se pueden cifrar. Así la elección de dicha clave pública debe ser tal que el número de mensajes no cifrables sea mínimo. Un mensaje M se dice que es no cifrable si cumple la congruencia:

$$M^{e} \equiv M \pmod{N}$$

Según el teorema chino de los restos congruentes esto es a su vez equivalente a satisfacer simultáneamente las ecuaciones:

$$M^{e} \equiv M \pmod{p}$$
$$M^{e} \equiv M \pmod{q}$$

En general se puede demostrar que el número de soluciones x de la ecuación congruencial:

$$x^{y} \equiv x \pmod{z}$$

tales que $0 \leq x < z$ con z un número primo es $\sigma_z = 1 + \text{mcd}\ (y - 1, z - 1)$. Por tanto, el número de mensajes M no cifrables σ_N viene dado por:

$$\sigma_N = \sigma_p \sigma_q = [1 + \text{mcd}\ (e - 1, p - 1)][1 + \text{mcd}(e - 1, q - 1)]$$

Como los valores e, p y q son impares, podemos comprobar que la operación de máximo común divisor tendrá al menos valor 2, por lo que existen al menos 9 mensajes que no son cifrables en cualquier sistema RSA. El caso más desfavorable es aquél en el cual se verifican las relaciones:

$$\text{mcd}\ (e - 1, p - 1) = p - 1$$

$$\text{mcd}\ (e - 1, q - 1) = q - 1$$

de forma que $\sigma_N = N$, es decir, ningún mensaje es cifrable. Una forma óptima de conseguir que el número de mensajes no cifrables sea el mínimo posible es elegir e de tal forma que el resultado de ambas ecuaciones sea 2. En particular, una posible clave pública de cifrado que siempre lo verifica es e = 3, sin embargo este valor es demasiado pequeño y su utilización no es aconsejable por razones de seguridad.

Si tomamos p y q números primos fuertes de la forma:

$$p = 2\,p' + 1$$

$$q = 2\,q' + 1$$

con p’ y q’ números primos grandes, se garantiza que el máximo común divisor será 2 ó p’ en un caso y 2 ó q’ en el otro, siendo muy grande la probabilidad de que sea 2 en ambos casos, sin embargo es conveniente comprobar esta condición cuando se elige la clave pública e.

2.3. COMUNICACIONES CON EL SISTEMA RSA.

El sistema RSA además de un proceso de cifrado de clave pública permite ser utilizado como una herramienta básica para el desarrollo de protocolos en las comunicaciones digitales. La ventaja de la aplicación del cifrado de clave pública RSA con respecto a los cifrados en bloque de clave secreta deriva de su sencillez y elegancia. La clave pública puede estar en un directorio telefónico o bien puede comunicarse en un canal abierto, mientras que la clave secreta constituye un secreto no compartido.

Mediante un sistema RSA podemos:

- Establecer comunicación entre emisor y receptor.
- Garantizar confidencialidad en las comunicaciones.
- Garantizar la autenticidad de origen y contenido de los mensajes transmitidos.

Para explicar cada una de las posibilidades anteriores se van a considerar dos comunicantes A y B, con claves públicas (N_a, e_a) y (N_b, e_b), y claves privadas d_a y d_b respectivamente.

2.3.1. Establecimiento de la comunicación.

Cualquier tipo de comunicación requiere el reconocimiento previo de los interlocutores. En el caso de las comunicaciones digitales existen multitud de procedimientos para el reconocimiento de los interlocutores, a continuación se describirá uno de ellos basado en el sistema de clave pública RSA.

Supongamos que los comunicantes A y B acuerdan identificarse mutuamente mediante sus claves públicas RSA. Lo primero que deben hacer los comunicantes es el intercambio de sus identidades, es decir, de sus claves públicas. En estas condiciones, supongamos que antes de enviar sus mensajes A desea estar seguro de que es realmente B quien se encuentra al otro lado de la línea. Para ello A elige un entero aleatorio R_a tal que $1 < R_a < \min(N_a, N_b)$ y envía a B el cifrado C_1:

$$C_1 \equiv R_a^{e_b} \pmod{N_b}$$

Por su parte, B elige un entero aleatorio R_b que cumple los mismos requisitos que R_a y envía a A el mensaje cifrado C_2:

$$C_2 \equiv R_b^{e_a} \pmod{N_a}$$

Si A y B están realmente conectados entonces A descifra C_2 con su clave secreta y recupera R_b y B descifra C_1 y recupera R_a:

$$R_b \equiv C_2^{d_a} \pmod{N_a}$$
$$R_a \equiv C_1^{d_b} \pmod{N_b}$$

En estas condiciones A envía a B el cifrado C_3:

$$C_3 \equiv R_b^{e_b} \pmod{N_b}$$

Igualmente, B envía a A el cifrado C_4:

$$C_4 \equiv R_a^{e_a} \pmod{N_a}$$

Entonces A descifra C_4 con su clave secreta y recupera R_a:

$$R_a \equiv C_4^{d_a} \pmod{N_a}$$

De esta forma A puede estar seguro de que realmente es B quien está al otro lado de la línea, ya que únicamente B pudo descifrar R_a con su clave secreta.

Por su parte, B descifra C_3 y recupera R_b mediante la congruencia:

$$R_b \equiv C_3^{d_b} \pmod{N_b}$$

Así B puede estar seguro de que es A quien realmente está al otro lado de la línea, ya que únicamente A pudo descifrar R_b con su clave secreta.

2.3.2. Confidencialidad.

Supongamos que el comunicante A desea enviar confidencialmente a B un mensaje M_{ab} tal que $1 < M_{ab} < N_b$. Para ello, A cifra el mensaje M con la clave pública de B:

$$C_{ab} \equiv M_{ab}^{e_b} \pmod{N_b}$$

Únicamente B, poseedor de la clave secreta d_b pareja de su clave pública e_b, puede recuperar el mensaje:

$$M_{ab} \equiv C_{ab}^{d_b} (\mathrm{mod}\, N_b)$$

De la misma forma B puede enviar confidencialmente a A un mensaje M_{ba} cifrado con la clave pública de A, siendo A el único que podrá descifrarlo ya que es el único que conoce su clave privada d_a.

Para establecer un sistema de comunicaciones como el anterior tan sólo es necesario publicar las claves públicas de los comunicantes, sin embargo esto podría suponer un problema, el receptor estaría expuesto a la recepción continua y malintencionada de mensajes anónimos. Este problema puede evitarse en las comunicaciones digitales basadas en el sistema RSA. La simetría operacional del RSA con respecto a los dos exponentes que actúan como claves permite la obtención de firmas digitales que garanticen la autenticidad de origen y contenido de los mensajes transmitidos.

2.3.3. Autenticidad de origen y contenido. Firma digital.

El sistema de cifrado de clave pública RSA ofrece la posibilidad de obtener firmas digitales de una forma sencilla y elegante. Supongamos que el comunicante A desea enviar a B un mensaje M de tal forma que quede garantizada la autenticidad de origen del mismo. Para ello A envía a B el mensaje M y una firma digital del mismo M_f:

$$M_f \equiv M^{d_a} (\mathrm{mod}\, N_a)$$

El comunicante B recibe y verifica la firma M_f , descifrándola mediante la clave pública e_a de A. De esta forma B recupera M:

$$M \equiv M_f^{e_a} (\mathrm{mod}\, N_a)$$

En estas condiciones, el valor entero M_f , cifrado de M con la clave secreta d_a de A, es el documento digital que garantiza a B que el mensaje M proviene de A, permitiéndole a su vez demostrarlo ante un tercero en caso necesario. Sin embargo, el esquema de firma no garantiza a A la autenticidad de contenido del mensaje firmado.

Para conseguir esto es conveniente incluir algún dato sobre la identidad del firmante en el mensaje firmado. De esta manera la generación de una firma válida de cualquier mensaje requiere a su vez la generación de una firma válida de dichos datos relativos a la identidad del firmante, para lo que es necesario el conocimiento de la clave secreta del mismo. Con ello se elimina la posibilidad de que cualquiera pueda generar una firma válida en nombre de A sin ejercer control sobre el mensaje firmado.

En el caso de la autenticidad de contenido, cuando se trata con mensajes muy largos, es una práctica común el firmar un resumen del mismo en lugar de firmar el mensaje completo. El resumen se obtiene aplicando al mensaje una función pública de una sola dirección. Las funciones resumen se utilizan fundamentalmente por razones prácticas, para agilizar el proceso de generación y verificación de firmas digitales, así como también para reducir las necesidades de almacenamiento. El concepto de función resumen plantea a su vez ciertos problemas de seguridad, puesto que al transformar el espacio de mensajes en un espacio de resúmenes más reducido, la posibilidad de que existan varios mensajes con el mismo resumen es inevitable. Por lo tanto, el estudio de funciones resumen seguras es uno de los campos más activos de investigación en la actualidad.

A continuación se mostrará un esquema general donde se tendrá en cuenta la autenticidad de origen y la autenticidad de contenido de los mensajes transmitidos digitalmente.

Supongamos que A desea enviar confidencialmente a B un mensaje M, para ello previamente A calcula un resumen del mensaje H(M), siendo H una función pública resumen de una sola dirección. Sabiendo que se cumple $1 < M, H(M) < \min (N_a , N_b)$, A cifra el mensaje M con la clave pública de B y el resumen con su clave secreta, y los envía a B:

$$C \equiv M^{e_b} \pmod{N_b}$$
$$H(M)_f \equiv H(M)^{d_a} \pmod{N_a}$$

El comunicante B descifra C con su clave secreta y la firma $H(M)_f$ con la clave pública de A, recuperando respectivamente los valores M y H(M):

$$M \equiv C^{d_b} \pmod{N_b}$$
$$H(M) \equiv H(M)_f^{e_a} \pmod{N_a}$$

En estas condiciones, haciendo uso de la función resumen pública H, el comunicante B calcula el resumen del mensaje recuperado M y verifica si éste coincide con el resumen recuperado H(M). La firma digital $H(M)_f$ del resumen del mensaje con la clave secreta de A es el documento digital que B guarda y que puede usar para demostrar a un tercero que el mensaje proviene de A, puesto que solamente A pudo haberlo generado con su clave secreta. Este esquema asegura que el documento digital está firmado por quien dice ser A, pero no garantiza que en realidad lo sea, ya que la clave secreta d_a de A puede haber sido sustraída.

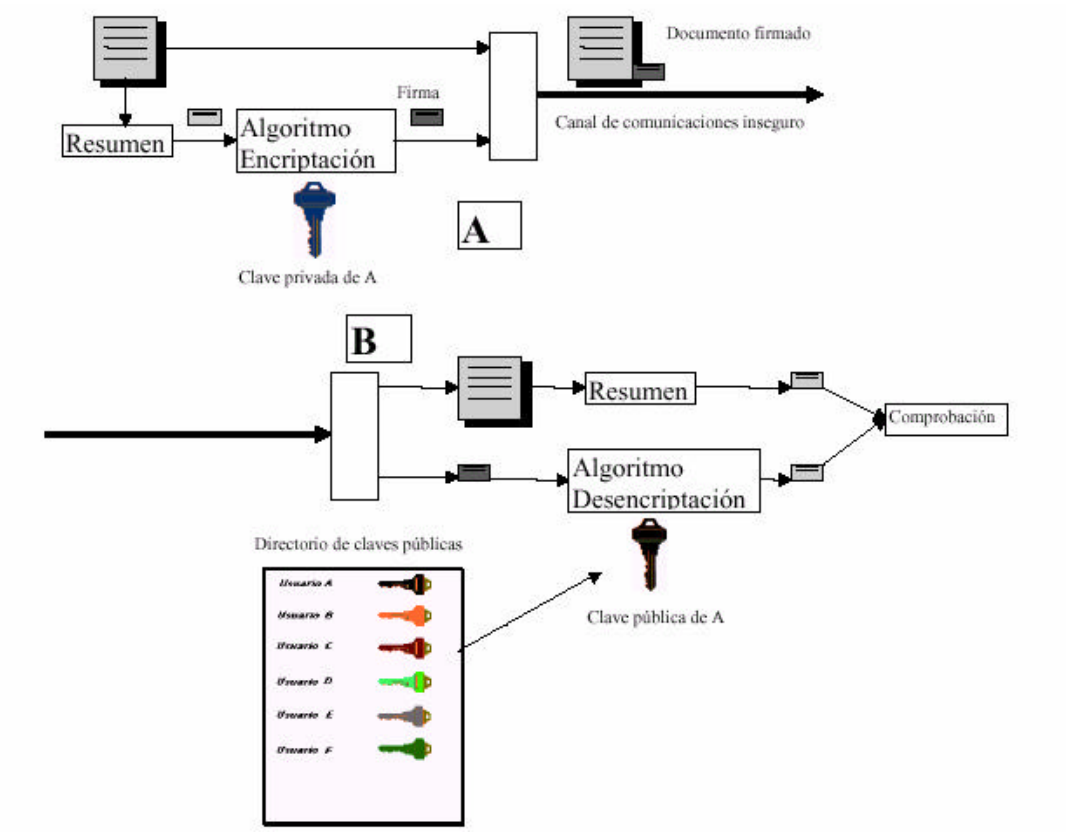

Figura 1: Esquema sitema RSA con autenticidad en origen y destino.

2.4. ATAQUES AL SISTEMA RSA.

Una gran variedad de ataquies criptoanalíticos han sido propuestos con el fin de romper el sistema de cifrado RSA. Sin embargo, hasta la fecha, ninguno de ellos parece haber conseguido su objetivo de forma efectiva. Algunos de los ataques más representativos son los conocidos como:

- Ataque cíclico.
- Ataque basado en la factorización de la clave pública.
- Ataque basado en oráculos.
- Otros ataques.

2.4.1. Ataque cíclico.

El ataque cíclico está basado en la idea de que todo sistema RSA constituye un grupo multiplicativo con un número finito de elementos. Para describir el ataque, consideremos un sistema de cifrado RSA definido por las claves públicas e y N y la clave secreta d. Sea M un mensaje tal que $1 < M < N$ y C su cifrado con la clave pública e dado por:

$$C = M^e \pmod N$$

El ataque cíclico permite descifrar C sin necesidad de conocer a clave secreta d. Para ello basta con realizar cifrados sucesivos del cifrado inicial C con la clave pública e hasta obtener nuevamente el cifrado C. Esto equivale al computo de la serie de congruencias:

$$C_i = C_{i-1}^{\ e} \pmod N \quad \text{para } i=1,2,..$$

con Co = C hasta que se verifique que Ci = C (mod N). Llegados a este punto es sencillo darse cuenta de que a su vez se debe verificar que:

$$M = C_{i-1} \pmod N$$

Es decir, si después de y cifrados consecutivos se obtiene de nuevo el cifrado inicial, entonces el cifrado (i – 1) corresponde al mensaje buscado. Este ataque cíclico puede ser evitado si los factores primos p y q de la clave pública N son números primos fuertes, es decir, suficientemente grandes y de la forma:

$$p = 2p' + 1$$

$$q = 2q' + 1$$

con p’ y q’ números primos grandes, puesto que entonces se puede fácilmente demostrar que todos los posibles subgrupos de trabajo son de un orden suficientemente elevado como para hacer impracticable el ataque.

2.4.2. Ataque basado en la factorización de la clave pública.

Otra forma de romper el sistema de cifrado RSA es mediante la factorización de su clave pública N. Por lo general esta tarea no es sencilla siempre y cuando los factores primos sean suficientemente grandes. Además, es precisamente esta dificultad la que garantiza la seguridad de este sistema de cifrado.

La factorización ha sido desde siempre uno de los problemas generales de teoría de números que ha suscitado mayor interés siendo igualmente el tema central y objeto de estudio en muchos de los procedimientos de criptoanálisis. El problema de factorización consiste en obtener la descomposición en factores primos de cualquier número entero. Un teorema fundamental del álgebra nos asegura que esta descomposición existe y es única. Sin embargo, esto no es suficiente para resolver el problema, puesto que, aunque se garantiza la existencia y unicidad de dicha descomposición, no se conoce ningún conocimiento de cálculo eficiente para obtener la de valores enteros muy elevados. Desde los tiempos de Fermat es sabido que el problema de la descomposición de un número entero N en sus factores primos se puede subdividir en otros tres problemas:

- Decidir si el número entero N es compuesto (de forma determinista) o bien si es primo con cierta probabilidad (test de primalidad probabilístico).
- Decidir si un número entero N probablemente primo realmente lo es (test de primalidad determinista).
- Una vez demostrado que el número entero N es compuesto, obtener sus factores primos.

El problema de factorización es el más importante de todos, cabe destacar que cuando se trabaja con números enteros muy grandes la mayoría de los algoritmos de factorización tienen una elevada complejidad computacional, lo que se traduce en

tiempos de computo muy elevados. Esta circunstancia hace que dichos algoritmos sean muy poco eficientes e incluso inviables cuando se trabaja con números enteros grandes.

Una forma de clasificar a los métodos de factorización es en dos grupos: los métodos de "propósito especial", quienes buscan propiedades especiales a los factores de n y 2, y los métodos de "propósito general" que dependen sólo del tamaño de n.

Entre los métodos de **"propósito especial"** están:

- El método del ensaño de divisores pequeños (conocido por la criba de Eratosthenes).
- El algoritmo $p-1$ de Pollard.
- El método que usa curvas elípticas.
- Criba de campos numéricos especial.

Entre los métodos de **"propósito general"** están:

- Criba cuadrática.
- Criba de campos numéricos general**.**

En general no existe una forma eficiente de factorizar a un número entero si no se conoce nada acerca de él, ya que algunos métodos son más eficientes en algunos casos que en otros, sin embargo podemos dar una estrategia para poder intentar factorizar un número entero n*:*

1. Aplicar el método de ensañar divisores pequeños, hasta una cota c_0.
2. Aplicar el algoritmo de Pollard, esperando encontrar un factor r, tal que $c_0 < r < c_1$ donde $c_1$es otra cota.
3. Aplicar el método con curvas elípticas, esperando encontrar un factor r_1 tal que $c_1 < r < c_2$.
4. Aplicar un método de propósito general.

Con lo que esperamos optimizar el proceso de factorizar, aplicando el mejor método conforme a la forma del número.

Si ya se conoce una estrategia para factorizar al número entero, el parámetro más importante es el tiempo que tomará para lograr el objetivo. La práctica ha permitido llegar a medir algunos algoritmos con la siguiente cota:

$$L[n,v,c] = O\left[\exp(c(\log n)^{v}(\log\log n)^{1-v})\right]$$

donde podemos suponer que la función como cota superior a considerar es:

$$L[n,v,c] = \exp(c(\log n)^{v}(\lg\log n)^{1-v}$$

donde c es una constante, n el tamaño del número, v un número tal que $0 < v < 1$ y determina la "lentitud" del algoritmo en el sentido de que si $v = 1$, entonces L se convierte a $L[n,1,c] = \exp(c\log n)$, y si $v = 0$, entonces $L[n,0,c] = \exp(c\log\log n)$.

Si tenemos un algoritmo que corre a un tiempo de $L[n, 0, c]$ diremos que corre en **tiempo polinomial,** si el algoritmo tiene una complejidad de $L[n, 1, c]$, entonces diremos que corre a un **tiempo exponencial,** y si $0 < v < 1$, diremos que el algoritmo corre a un **tiempo subexponencial.**

Ejemplos del tiempo que corren algunos algoritmos conocidos son:

Algoritmo: Criba de campos numéricos general, v=1/3.
Tiempo: $L[n, 1/3, c_0 + o(1)]$, donde $c_0 = (\frac{64}{9})^{1/3}$.

Algoritmo: Criba de campos numéricos especial, v=1/3.
Tiempo: $L[n, 1/3, c_1 + 0(1)]$, donde $c_1 = (\frac{32}{9})^{1/3}$.

Algoritmo: Criba cuadrática, v=1/2.
Tiempo: $L[n, 1/2, 1 + o(1)]$.

Algoritmo : Método de fracciones continuas, v=1/2.

Tiempo: $L[n, 1/2, c_2 + o(1)]$, donde $c_2 = \sqrt{2}$.

Algoritmo : Método de Factorización con Curvas Elípticas.

Tiempo: $L[p, 1/2, \sqrt{2}]$, para encontrar el factor p y si $n = pq$, el tiempo es $L[n, 1/2, 1]$.

Hoy en día, parece ser que el problema de factorización ha progresado enormemente, este gran impulso lo ha provocado en gran parte la criptografía y más concretamente el sistema RSA.

La notación "mips" significa millones de instrucciones por segundo y es tomada como medida estándar para registrar la potencia de cómputo con que se cuenta. Un mips tiene como origen la potencia de cómputo que realizaba una DEC VAX 11/780. Un mipsy significa el número de instrucciones que se pueden realizar en un año con un poder de cómputo de un mips, es decir 31 536 x 10^9 instrucciones.

Si hacemos una estimación del poder disponible estimado en todo el mundo usando la predicción conocida por "ley de Moore", que predijo en 1965 que la rapidez con que crecen los microprocesadores se duplica en 18 meses, partiendo de que el poder de cómputo en el mundo en 1994 era de 3x10^8 mipsy, entonces en el año 2002 alanzaríamos un poder de 9.6x10^9 mipsy, lo cual no está muy lejos de la realidad.

En la siguiente tabla se muestra la estimación de los próximos años:

Año	**mipsy**
1997	1.2 x 10^9
98-99	2.4 x 10^9
2000	4.8 x 10^9
01-02	9.6 x 10^9
2003	1.92 x 10^{10}
04-05	3.84 x 10^{10}
2006	7.68 x 10^{10}

Año	**mipsy**
2015	4.9 x 10^{12}
2021	8 x 10^{13}
2028	10^{15}
2034	10^{16}
2038	10^{17}
2043	10^{18}
2049	10^{19}

2009	3.07×10^{11}
2012	1.2×10^{12}

2053	10^{20}
2058	10^{21}

Tabla 2.2: Poder de cómputo en los próximos años.

Para poder determinar cuándo se podría contar con el poder necesario para factorizar un número entero de tamaño conocido debemos de saber qué poder de cómputo es necesario utilizar para tal factorización.

Esto lo estimaremos partiendo de que sabemos cuántos mipsy son utilizados para factorizar un número entero, por ejemplo un número de 129 dígitos (429 bits) se puede factorizar con 1000 mipsy, entonces para poder factorizar un número de 512 bits necesitaremos de T_{512}, donde T_{512} lo obtenemos de la siguiente fórmula:

$$T_n = T_{n-1} \bullet L[n] / L[n-1]$$

donde T_{n-1} es el poder de cómputo requerido para un número de menos bits y $L[n], L[n-1]$ son las respectivas complejidades que toma la criba de campos numéricos general, así por ejemplo para saber el poder de cómputo requerido para factorizar un número entero de 512 bits procedemos como sigue:

$$T_{512} = 1000 \bullet L[2^{512}] / L[2^{429}] = 29369.2 \sim 3 \times 10^4$$

Así podemos mostrar la estimación del poder de computo requerido para factorizar un número entero aleatorio en la siguiente tabla:

Dígitos	**Bits (i)**	**mipsy**	**T_i**
129	429		1000
154	512	29369.2	3×10^4
192	640	2.990×10^6	3×10^6
231	768	1.800×10^8	2×10^8
269	896	7.331×10^9	7×10^9
308	1024	2.198×10^{11}	2×10^{11}
346	1152	5.150×10^{12}	5×10^{12}
385	1280	9.835×10^{13}	9×10^{13}
423	1408	1.580×10^{15}	2×10^{15}
462	1536	2.189×10^{16}	2×10^{16}
500	1664	2.666×10^{17}	3×10^{17}

539	1792	2.898×10^{18}	3×10^{18}
577	1920	2.849×10^{19}	3×10^{19}
616	2048	2.558×10^{20}	3×10^{20}

Tabla 2.3: Poder de cómputo necesario para factorizar número primo.

Con estos datos podemos de un modo más fundamentado, estimar la seguridad de las llaves utilizadas en RSA, es decir, el poder usar una llave de longitud predeterminada sin arriesgar la seguridad, al menos evitando el ataque de la factorización (que es el ataque más seguro conocido hasta hoy).

A partir de las observaciones anteriores mostramos en la siguiente tabla el año donde es posible pensar que la longitud de la llave señalada puede ser rota:

Año límite	**Longitud de llave**
2003	896
2012	1024
2018	1152
2022	1280
2034	1408
2038	1536
2043	1664
2049	1794
2053	1920
2058	2048

Tabla 2.4: Longitud límite de llave del sistema RSA.

El año límite se refiere al año en donde el sistema debe comenzar a evaluar el cambio de la longitud de la llave a una longitud superior.

La anterior estimación no contempla varios aspectos que son muy difíciles de predecir como: el descubrir un método brillante que factorice números enteros en tiempos reducidos y colapsé al sistema RSA**,** el descubrir un nuevo tipo de tecnología que aumente la rapidez exponencialmente de las computadoras por lo que reduciría a gran escala el tiempo de factorización, u otro suceso que reduzca a gran escala el tiempo de factorización. Es prudente decir que la anterior estimación fue tomada suponiendo

que el crecimiento tanto de rapidez de las computadoras como de la eficacia de los algoritmos, se comportará como hasta hoy; que pareciera no ser de otra forma. Así mismo no se considera el descubrimiento de otro ataque a RSA que no tenga que ver con la factorización, pero elimine el interés de su desarrollo así como también factores no previstos.

2.4.3. Ataque basado en oráculos.

Este otro posible ataque al sistema RSA está basado en la existencia de un algoritmo capaz de proporcionar cierta información parcial a cerca del sistema, cuyo conocimiento permita a un eventual atacante la ruptura del mismo. Se supone además que esta información se obtiene sin ningún coste, es decir, la complejidad computacional de este ataque es nula. Una forma de representar un algoritmo de tales características es mediante una especie de “caja negra” u “oráculo” capaz de responder correctamente a las preguntas que se le formulen acerca de la información parcial del sistema antes referida. Para describir el ataque, volvemos a considerar un sistema RSA definido por las claves públicas e y N y la clave secreta d. Sea M un mensaje con $1 < M < N$, y C su cifrado que viene dado por:

$$C = M^e \pmod N$$

En estas condiciones, dos posibles oráculos que podrían ser utilizados para romper el sistema RSA son los siguientes:

- Oráculo dicotómico: toma como entrada un mensaje cifrado y su salida refleja si el correspondiente mensaje en claro es mayor o menor que N/2. Es decir, dado C el oráculo dicotómico computa un valor igual a cero si $0 \leq M < N/2$ o bien un valor igual a uno si $N/2 < N < M$.

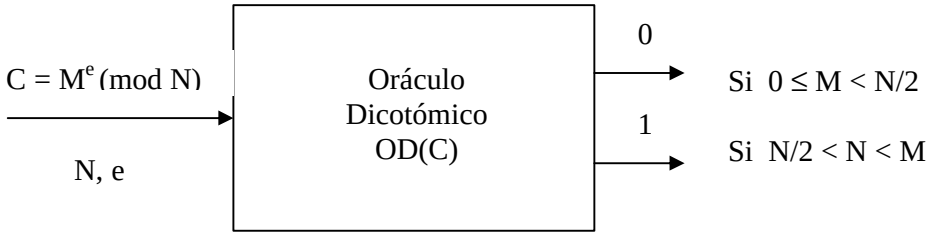

Figura 4: Oráculo dicotómico.

- Oráculo de paridad: toma como entrada un mensaje cifrado y da como salida la paridad del mensaje claro, correspondiente. Así pues, el oráculo computa el bit menos significativo de M.

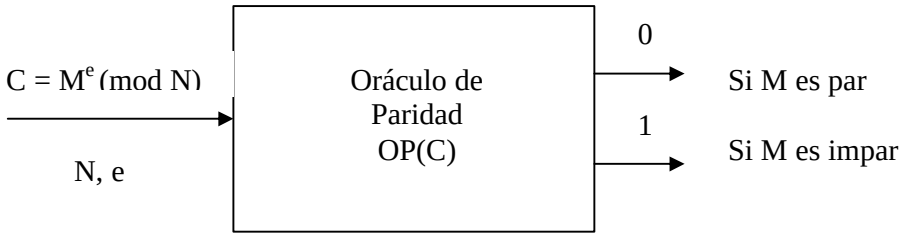

Figura 5: Oráculo de paridad.

Podemos comprobar que OP(C) y OD(C) son computacionalmente equivalentes, se puede deducir de las siguientes relaciones:

$$OD(C) = OP(C \bullet 2^e \pmod N)$$

$$OP(C) = OD(C / 2^e \pmod N)$$

Una vez llegados a este punto, tenemos que ver la forma en la que cualquiera de estos oráculos puede ser utilizado para construir un algoritmo capaz de computar el mensaje en claro M a partir del cifrado, y de esa forma romper el sistema RSA. Elegiremos para ello el oráculo dicotómico OD. Antes de presentar el algoritmo podemos comprobar que para este oráculo se verifican las siguientes relaciones:

$$OD(M^e \pmod N) = 0 \Leftrightarrow M \in [0, N/2)$$

$$OD((2M)^e \pmod N) = 0 \Leftrightarrow M \in [0, N/4) \cup [N/2, 3N/4)$$

$$OD((4M)^e \pmod N) = 0 \Leftrightarrow M \in [0, N/8) \cup [N/4, 3N/8) \cup [N/2, 5N/8) \cup [3N/4, 7N/8)$$

Y así seguiríamos sucesivamente. Un posible algoritmo para recuperar el mensaje M a partir del cifrado C se presenta a continuación:

1. Hacer $k = \lfloor \log_2 N \rfloor$
2. Hacer lo = 0
3. Hacer hi = N
4. Para i = 0 hasta k repetir pasos (5-7)
5. Hacer mid = (lo + hi)/2
6. $C_i = OD(C \bullet 2^{ei} \pmod N)$
7. Si $C_i = 1$, entonces hacer lo = mid

   Si no, hacer hi = mid
8. Hacer $M = \lfloor hi \rfloor$

El algoritmo utiliza el oráculo dicotómico en el paso 6 para calcular los valores:

$$\{OD(M^e \pmod N), OD((2M)^e \pmod N), OD((2M)^e \pmod N), ...\}$$

los cuales permiten una búsqueda dicotómica del mensaje M de acuerdo con las relaciones anteriores. También podríamos haber utilizado el oráculo de paridad sin más que cambiar el paso número 6 del algoritmo por este otro:

6. $C_i = OP(C \bullet 2^{e(i+1)} \pmod N)$

Así pues, hemos comprobado como la existencia de uno cualquiera de los oráculos anteriores supondría la ruptura del sistema RSA. Sin embargo, hasta la fecha, todavía no se ha conseguido la puesta a punto de ninguno de ellos, por lo que el sistema RSA parece seguir siendo seguro.

2.4.4. Otros posibles ataques.

A continuación describiremos las directrices básicas de otros tipos de ataques al sistema RSA, se trata de ataques más recientes, y se han incluido por su relativa inportancia actual:

- Ataque por control de tiempos (por P. C. Kocher).

Como su propio nombre indica, está basado en la idea de medir el tiempo promedio invertido por el dispositivo cifrante para la realización de las operaciones básicas de cifrado, puesto que ello permite extraer información acerca de las claves secretas de cifrado. En el caso de un sistema de cifrado RSA esto se traduce en la medida del tiempo promedio requerido por el microprocesador utilizado para el cómputo de las operaciones básicas necesarias para el cálculo de la exponenciación modular. Este ataque no tiene coste computacional alguno y es especialmente apropiado para el caso de tarjetas inteligentes u otros dispositivos criptográficos sobre los cuales el atacante puede efectuar medidas de tiempo razonablemente precisas. Existen a su vez diversas técnicas que permiten prevenir este tipo de ataques, como son la introducción de información adicional o de funciones resumen.

- Ataque por introducción de faltas.

Fue propuesto en el año 1997 por D. Boneh, R. A. De Millo y R. J. Lipton. Al igual que el anterior, este ataque fue concebido para llevarse a cabo sobre dispositivos criptográficos protegidos que operan con las claves secretas que el atacante desea obtener. El ataque está basado en la idea de que cuando un dispositivo de este tipo realiza una computación errónea, es posible aprovechar dicha falta para extraer información acerca de las claves secretas con las que opera. Así, teniendo esto en cuenta, un eventual atacante puede provocar la falta desde el exterior y obtener la información deseada comparando los valores erróneos generados por el dispositivo a consecuencia de la falta con los valores correctos, es decir, los que hubiese generado si la falta no se hubiera producido. Cabe destacar que existen numerosos procedimientos físicos capaces de provocar estas faltas en los dispositivos referidos, como pueden ser por ejemplo la aplicación de determinados tipos de radiación, las alteraciones bruscas de la tensión de alimentación, los cambios de la frecuencia de reloj del sistema, etc. Sin embargo, para poder provocar las correspondientes faltas, el atacante debe ser capaz de ejercer un cierto control sobre el dispositivo en cuestión. Ésta es la razón por la que este ataque está especialmente indicado para el caso de tarjetas inteligentes u otros dispositivos criptográficos sobre los cuales el atacante puede tener un control más directo.

Después de describir ambos métodos, podemos observar que dichos ataques tienen en común el hecho de tomar en consideración el comportamiento del microprocesador que efectúa los cálculos del dispositivo criptográfico en cuestión. Sin embargo, mientras que en el primer caso el ataque se limita a la simple observación de dicho comportamiento, es decir, en cierto modo es un ataque pasivo, el segundo lo intenta modificar de forma activa introduciendo faltas. Por esta razón, aunque todavía no haya sido llevado a la práctica, se puede pensar que éste último es algo más resistente.

3. IMPLEMENTACIÓN HDL DEL ALGRORITMO RSA.

En este capítulo vamos a tratar los diferentes métodos de cálculo usados para las exponenciaciones modulares utilizadas en el algoritmo, y la consiguiente elección del más adecuado. Se buscará el método óptimo para la implementación HDL del algoritmo, basándonos tanto en su simplicidad como en su coste computacional.

El algoritmo RSA consiste en la implementación de la exponenciación modular $C = M^e \pmod N$, debido a la complejidad del algoritmo cuando se utilizan palabras de gran tamaño, si realizásemos primero la exponenciación y posteriormente el resto de la división, no se podría almacenar dicha cantidad debido a que el número binario resultado de la exponenciación (M^e) es enorme.

Otra posibilidad es reducir los diferentes pasos de la exponenciación a módulo N, es decir, después de realizar una multiplicación le aplicaríamos la consiguiente reducción módulo N. Así, si consideramos palabras de 256 bits tanto para el exponente (e) como para el mensaje (M), se necesitarían:

$$\text{Log}_2 (M^e) = e \cdot \log_2 (M) \cong 2^{256} \cdot 256 = 2^{264} \cong 10^{80} \text{ bits.}$$

Este número es aproximadamente igual al número de partículas en el universo, y no hay ninguna forma posible de almacenamiento, ni incluso considerando todos los ordenadores del mundo. Por ello debemos acudir a un método alternativo para realizar la exponenciación modular de forma que los diferentes resultados intermedios sean almacenables.

A continuación se expondrán diferentes métodos de cálculo de dicha exponenciación modular, y los diferentes criterios para basarnos en la elección del método más adecuado para nuestra implementación. El criterio más importante será el relacionado con el coste computacional de las diferentes operaciones realizadas, por lo que el objetivo fundamental será el de reducir el número de operaciones tales como multiplicación y división, que son las que tienen asociado un mayor coste computacional.

3.1. MÉTODOS DE EXPONENCIACIÓN MODULAR.

La exponenciación modular es una operación básica en la mayoría de los sistemas de cifrado de clave pública. A continuación se presentan diferentes procedimientos para realizar su cálculo, algunos de ellos son de propósito general, si bien otros están especialmente indicados para su utilización en determinado tipo de aplicaciones.

Si se tiene en cuenta que se busca el resultado de la operación aritmética $a^b \pmod c$, se puede apreciar que la operación más importante para el cómputo de exponenciaciones modulares es la multiplicación, siendo a su vez esta operación la de mayor coste computacional. Así pues, para optimizar el cálculo de las exponenciaciones modulares es conveniente a su vez optimizar el de esta operación básica. Por esta razón, antes de considerar los posibles procedimientos para el cómputo de las exponenciaciones modulares, y teniendo en cuenta que la multiplicación es una operación que en sí misma permite una considerable optimización, se comienza por describir un algoritmo para conseguir esto último, en concreto el algoritmo de Karatsuby y Y. Ofman.

Posteriormente, se describirán algunos de los procedimientos más conocidos para optimizar el cálculo de las exponenciaciones modulares, estudiándose la complejidad computacional de cada uno de ellos, es decir, el número de operaciones elementales requeridas para su ejecución, con el objetivo de establecer una comparación entre los mismos.

3.1.1. Método de multiplicación de Karatsuba-Ofman.

El algoritmo de Karatsuba-Ofman es un procedimiento que permite reducir el número total de multiplicaciones elementales necesarias para el cálculo del producto de dos valores enteros. Esto se consigue dividiendo los factores en fragmentos más pequeños, los cuales se reagrupan de diversas formas con el fin de reducir el número de multiplicaciones elementales necesarias para computar dicho producto. Evidentemente, esta reestructuración de los datos requiere a su vez el cómputo de algunas operaciones adicionales, de forma que el óptimo de este método se encuentra precisamente cuando el

ahorro computacional derivado de la fragmentación de los factores compensa el coste adicional que ello supone.

Si se considera que se trabaja en una base polinómica de representación igual a B, y que la representación de los enteros u y v en dicha base son:

$$u = u_0 + u_1B + u_2B^2 + \cdots\cdots + u_{n-1}B^{n-1}$$
$$v = v_0 + v_1B + v_2B^2 + \cdots\cdots + v_{n-1}B^{n-1}$$

es decir, cada uno de los enteros tiene n coordenadas en base B. Con todo lo anterior, sea $X = B^m$ con m valor entero definido por:

$$m = \begin{cases} \dfrac{n}{2} & si\ \ n\ \ es\ \ par \\ \dfrac{n+1}{2} & si\ \ n\ \ es\ \ impar \end{cases}$$

Así podemos descomponer cada valor entero u y v en dos partes, respectivamente (U_0, U_1) y (V_0, V_1), de forma que:

$$u = U_0 + U_1X$$

$$v = V_0 + V_1X$$

es decir, siendo (U_0, U_1) y (V_0, V_1) las coordenadas respectivas de u y v en la base de representación $X = B^m$. En estas condiciones, se verifica la identidad dada por:

$$uv = U_0V_0 + [\ U_0V_0 + U_1V_1 - (U_1 - U_0)(V_1 - V_0)]\ X + U_1V_1\ X^2$$

Puesto que X es una potencia de la base de representación B, las aparentes multiplicaciones por potencias de X son en realidad simples desplazamientos lógicos y por tanto no se contabilizan como productos. Asimismo los sumandos primero y tercero de la parte derecha de la identidad anterior (respectivamente U_0V_0 y $U_1V_1X^2$) no se solapan, y por ello tampoco se contabiliza como tal su adición. De modo que existen tres multiplicaciones de longitud m y cinco adiciones/substracciones, de las cuales son dos de longitud m y tres de longitud 2m. De tal forma al multiplicar dos factores de longitud n se realizan ($2m + 3 \cdot 2\ m = 8m \cong 4n$) 4n adiciones/substracciones y tres multiplicaciones de factores de longitud $m \cong n/2$, es decir, un total de $O(3m^2) \cong O(3n^2/4)$ multiplicaciones elementales, frente a las $O(n^2)$ necesarias en el caso del método tradicional.

Se puede comprobar que si se utiliza el procedimiento de Karatsuba y Ofman par computar el producto de dos valores enteros grandes, es decir, con n elevado, entonces el coste computacional se reduce en un factor de 3 ó 4. Además este algoritmo puede ser aplicado recursivamente, consiguiendo así potenciar su efectividad gracias a la optimización dinámica derivada de la progresiva división de los factores.

Como nota negativa se observa que el ahorro computacional decrece a medida que la longitud de los factores también decrece, esto es así debido a las operaciones adicionales requeridas para la obtención de los distintos factores, de tal forma que el óptimo del método se obtiene cuando el ahorro computacional derivado de la fragmentación de los factores compensa el coste adicional que ello supone.

Se puede apreciar que el método de Daratsuba-Ofman es directamente aplicable cuando los valores enteros que van a ser multiplicados son del mismo tamaño n. Cuando esto no sucede, una posible forma de evitar el problema y poder aplicar el algoritmo es rellenar por la izquierda de forma previa el valor entero de menor longitud con el número de componentes nulas necesario para que su longitud sea n.

A partir de este punto se abandonará la optimización de la operación de multiplicación, y nos centraremos en la exposición de los diferentes métodos de cálculo de exponenciaciones modulares.

3.1.2. Método ingenuo.

Se trata de un método de cálculo muy simple para obtener la exponenciación modular basado en multiplicaciones modulares sucesivas. Para realizar el cálculo aritmético de $a^b \pmod c$ lo que se haría es:

Paso 1: Hacer $d_0 = 1$

Paso 2: Para i = 1,...,b

Hacer $d_i = ad_{i-1} \pmod c$

Paso 3: Salida d_b

Se comprueba que el número de multiplicaciones modulares requeridas es (b-1), que coincide con el número de divisiones realizadas para conseguir el resultado. No es un método muy aconsejable debido a que al realizar (b-1) productos modulares, si trabajamos con exponentes de gran tamaño, tenemos que realizar un gran número de operaciones de multiplicación y división, que son las que tienen un coste computacional mayor.

3.1.3. Métodos m-arios.

Estos métodos realizan el cálculo de la exponenciación $d \equiv a^b \pmod{c}$ partiendo de la representación binaria del exponente b. Para describirlos, se va a suponer que dicho exponente b tiene una longitud de n bits y un desarrollo de la forma:

$$b = (b_{n-1},....,b_0)$$

La idea básica de los métodos m-arios consiste en dividir el exponente b en grupos de m bits con $0<m\leq n$, de tal forma que las operaciones necesarias para el cálculo de la exponenciación modular se realizan en base a dichos grupos. Los métodos m-arios reciben distintos nombres dependiendo del valor de m. Así, por ejemplo, algunos de ellos son los siguientes:

- Método binario (m = 1): los bits se agrupan de uno en uno. Este procedimiento también se conoce con la denominación de método de izquierda-derecha.

- Método cuaternario (m = 2): los bits del exponente se agrupan por parejas.

- Método octal (m = 3): los bits se agrupan por tríos.

Para describir en detalle el funcionamiento de estos procedimientos, consideremos un método m-ario genérico tal que divide al exponente b en k grupos de m bits. El

número de estos grupos es $k = \lceil n/m \rceil$. En el caso que m no sea divisor de n, el grupo de los bits más significativos se completa por la izquierda con ceros.

Así pues, el exponente queda fragmentado de la forma:

$$b = (b_{n-1},...,b_0) = (b_{k-1\ m-1},\ldots b_{k-1\ 0},\ldots\ldots\ldots,b_{i\ m-1},\ldots,b_{i\ 0},.....,b_{0\ m-1},...b_{0\ 0})$$

donde b_{ij} es el bit j del grupo i con $0 \leq i < k$ y $0 \leq j < m$. En esta situación, se denota por h_i el número entero cuya representación binaria es:

$$h_i = (b_{i\ m-1},\ldots,b_{i\ 0}) \quad \text{con } 0 \leq i < k.$$

Con todo lo anterior, un algoritmo general para el cálculo de la exponenciación modular en función del parámetro m es el siguiente:

Paso 1: Hacer $a_1 = a$

Paso 2: Para $i = 2, 3,... ,2^m - 1$

$a_i \equiv a\ a_{i-1} \pmod c$

Paso 3: Hacer $d_k = 1$

Paso 4: Para i = k-1, ...,0

Hacer $d_i \equiv d_{i+1}^{2^m} \pmod c$

Si $h_i \neq 0$ entonces Hacer $d_i \equiv d_i\ a_{h_i} \pmod c$

Paso 5: Salida $d = d_0$

El algoritmo consta de un paso previo (paso 2), que variará según el parámetro m, y posteriormente (paso 4), se eleva al cuadrado y se realiza otro producto modular en el caso de que el grupo de bits seleccionado (h_i) sea distinto de cero.

Si designamos por k_1 al número de grupos de m bits del exponente b distintos de '0....0', las multiplicaciones requeridas para calcular la exponenciación modular $d \equiv a^b \pmod c$ suponiendo $b \neq 0$ vienen reflejadas en la siguiente tabla:

MÉTODOS M-ARIOS	**nº multiplicaciones**
Cálculos previos (paso 2)	$2^m - 2$
Cuadrados (paso 4-a)	m(k-1)
Multiplicaciones (paso 4-b)	$k_1 - 1$
Total	$\mathbf{m(k-1) + k_1 + 2^m - 3}$

Tabla 3.1: Operaciones de multiplicación de métodos m-ários.

Dependiendo del número de bits del exponente b existe un método m-ario óptimo para el cálculo de la exponenciación modular $d \equiv a^b \pmod c$. El método es óptimo en el sentido de que requiere el cómputo de un número mínimo de multiplicaciones.

Si designamos por w el número medio de multiplicaciones requeridas para el cómputo de la exponenciación, entonces el valor óptimo de m (m_{op}) se podría obtener a partir de:

$$\left(\frac{\partial w}{\partial m}\right)_{m=m_{op}} = 0$$

El valor de m óptimo obtenido es función del número de bits del exponente, puesto que w depende de k, y a su vez $k = \lceil n/m \rceil$. En la siguiente tabla se muestra el valor óptimo de m para diferentes longitudes del exponente b. Además se indica el número medio de operaciones realizadas en cada caso, comparándolo con el requerido en el caso de utilizar el método binario elemental (m = 1).

N	$\mathbf{m_{op}}$	$\mathbf{w(m=m_{op})}$	**w(m=1)**	**ahorro (%)**
1	1	0.00	0.00	0.00
2	1	1.50	1.50	0.00
4	1	4.50	4.50	0.00
8	2	10.25	10.50	2.38
16	2	21.25	22.50	5.55
32	2	43.25	46.50	6.98

64	2	87.25	94.50	7.67
128	4	167.06	190.50	12.30
256	4	325.06	382.50	15.01
512	5	638.81	766.50	16.65
1024	5	1247.62	1534.50	18.69
2048	6	2443.67	3070.50	20.41

Tabla 3.2: Valores de m óptimos según longitud de bits.

Se comprueba que con este tipo de métodos se contabiliza un ahorro de operaciones elementales con una buena elección del parámetro m, sin embargo, se observa que no se produce ningún ahorro computacional en lo que a operaciones de división se refiere.

3.1.4. Métodos m-arios adaptables.

Los métodos m-arios adaptables son una generalización de los métodos m-arios. La idea básica de estos procedimientos para computar la exponenciación $d \equiv a^b \pmod c$ consiste en divider el exponente b en cadenas de ceros de longitudes variables y en cadenas de bits de la forma '1,0,....,0,1' de longitud máxima m con $0 < m \leq n$. Así, estos métodos se van adaptando a los bits del exponente con el fin de minimizar dentro de lo posible el número de multiplicaciones requeridas para computar la exponenciación modular. A su vez, el hecho de que los grupos de bits distintos de cero sean de paridad impar (último bit igual a '1') permite reducir el número de cálculos previos.

La división del exponente $b = (b_{n-1}, ..., b_0)$ en grupos de bits distintos de '0' deben tener como tamaño máximo m, siendo m el parámetro de los métodos m-arios. Se designará por k al número de grupos de bits, y por m_i al número de bits del grupo i para $0 < i \leq k$. Así el exponente quedará dividido de la forma:

$$b = (b_{n-1},...,b_0) = (b_{k-1\ m_{k-1}-1},\ldots b_{k-1\ 0},\ldots\ldots\ldots,b_{i\ m_i-1},\ldots,b_{i\ 0},\ldots\ldots\ldots,b_{0\ m_0-1},...b_{0\ 0})$$

donde b_{ij} es el bit j del grupo i con $0 < i \leq k$ y $0 < j \leq m_i$. Si denotamos por h_i al valor entero denotado por:

$$h_i = (b_{i\ m_i-1},\ldots,b_{i\,0}) \text{ para } 0 < i \leq k$$

Así el algoritmo general de los método m-arios adaptables para el cómputo de la exponenciación modular en función del parámetro m es el siguiente:

Paso 1: Hacer $a_1 = a$

Paso 2: Hacer $a_2 = a^2 \pmod c$

Paso 3: Para $i = 1, 2, \ldots, 2^{m-1} - 1$

$a_{2i+1} \equiv a_2\, a_{2i-1} \pmod c$

Paso 4: Hacer $d_k = 1$

Paso 5: Para $i = k-1, \ldots, 0$

Hacer $d_i \equiv d_{i+1}^{2^{m_i}} \pmod c$

Si $h_i \neq 0$ entonces Hacer $d_i \equiv d_i\, a_{h_i} \pmod c$

Paso 6: Salida $d = d_0$

La diferencia fundamental con los métodos m-arios es que la longitud de los grupos de bits en los que se divide el exponente no es fija. Por esta razón el análisis del número total de multiplicaciones requeridas para el cálculo de la exponenciación es más difícil de realizar en el caso de métodos adaptables que en el de los de longitud fija.

La diferencia en el número de multiplicaciones realizadas se observa en los pasos previos (paso 2), mientras que en los de longitud fija se realizaban ($2^m - 2$) operaciones, en los de longitud variable se realizan (2^{m-1}). Esto es debido a que en los adaptables todos los grupos de bits distintos de '0...0' son de paridad impar, por lo que solo se requiere el cálculo previo del cuadrado y las potencias impares menores que 2^m de la base de la exponenciación.

Para concluir con este método, apuntar que aunque se produce un ahorro en el coste computacional comparado con los métodos m-arios normales debido a que el número de multiplicaciones es menor, sigue con el mismo problema de tener que realizar un número de divisiones elevado, por lo que sigue siendo un método de bastante coste computacional.

3.1.5. Método de reducción de Montgomery.

El método de Montgomery realiza el cálculo de las exponenciaciones modulares mediante la combinación de cualquier método m-ario (o m-ario adaptable) con una técnica de reducción aplicada al cómputo de productos modulares. Esta técnica, denominada reducción de Montgomery, tiene la característica básica de permitir el cálculo eficiente de multiplicaciones modulares sin necesidad de hacer divisiones. Esto último es de gran interés para reducir el coste computacional de los algoritmos de cálculo de exponenciacones modulares con números grandes, puesto que en todos ellos son precisamente las multiplicaciones y especialmente las divisiones (necesarias para el cálculo de los restos modulares) las operaciones de mayor coste.

Primeramente se va a describir el procedimiento denominado como reducción de Montgomery:

$$T_r = \text{Mont}(T) \equiv T\, r^{-1} \pmod{c}$$

Así, T_r es la reducción de Montgomery del entero T módulo c, respecto a r. Se debe verificar $0 \le T < c \cdot r$, y que c y r sean relativamente primos, es decir, mcd (c, r) =1. En estas condiciones, Montgomery sugirió que el cálculo del entero T_r podía realizarse sin necesidad de utilizar el método clásico de multiplicación modular (multiplicación de T y r^{-1} y posterior división por c para hallar el resto), sino que propuso un algoritmo alternativo que disminuía bastante el coste computacional.

Dicho algoritmo parte de un cálculo previo, $c' = -c^{-1} \pmod{r}$, y calcula la reducción de Montgomery del entero T de la forma que se presenta a continuación:

Mont (T) ^{o} T r^{-1} (mod c)

Paso 1: Hacer $U \equiv T\, c' \pmod{r}$

Paso 2: Hacer $T_r = \dfrac{T + Uc}{r}$

Paso 3: Si $T_r \ge c$ entonces $T_r = T_r - c$

Paso 4: Salida T_r

Se comprueba que las únicas divisiones realizadas a lo largo de la ejecución del algoritmo T_r = Mont (T) son divisiones por r. Por tanto, si suponemos que las operaciones elementales se realizan en una base numérica z con mcd (c, z) = 1, el coste computacional del algoritmo se reduce de forma substancial cuando $r = z^n$ con n entero, puesto que entonces las divisiones por r son simples desplazamientos lógicos de n posiciones a la derecha. Así, por ejemplo, si se trabaja en binario, entonces z = 2, y un valor adecuado para el parámetro r es 2^n , con n entero.

Este hecho de que el coste computacional del algoritmo se reduzca considerablemente para determinados valores de r puede ser aprovechado para acelerar el cómputo de las exponenciaciones modulares, por lo que una de las principales aplicaciones de la técnica de reducción de Montgomery es la conocida como exponenciación modular de Montgomery.

Producto de Montgomery.

Este método calcula la exponenciación modular de una forma más rápida y con menor coste computacional que otros métodos antes vistos. Para comprender mejor este algoritmo vamos a introducir el producto modular basado en la reducción de Montgomery, que consiste en la reducción modular del producto de dos valores enteros.

Sean a y b dos enteros que cumplen $0 \leq a,b < c$, y con mcd (c, r) = 1. Así la reducción de Montgomery del producto ab:

$$(ab)_r = a\ b\ r^{-1} \pmod{c}$$

Podemos calcular dicha reducción sin más que aplicar el protocolo anterior Mont (..) al producto ab, es decir, $(ab)_r$ = Mont $(a \cdot b)$.

Si trabajamos con un valor adecuado para r el coste computacional se reducirá en gran medida, así para calcular el producto modular $w = a \cdot b \pmod{c}$, comenzaríamos por calcular las reducciones de a y b:

$$a' = a\ r \pmod{c}$$

$$b' = b\ r \pmod{c}$$

A continuación, se computa la reducción de Montgomery $(a'b')_r$ del producto $(a'b')$ mediante el algoritmo anterior:

$$(a'b')_r = \text{Mont}\ (a' \cdot b')$$

La reducción obtenida coincide con la reducción del producto modular w:

$$w' = w\ r = ab\ r \pmod{c}$$

En estas condiciones, el producto modular w se obtiene mediante el computo de la reducción de Montgomery del entero w', puesto que:

$$(w')_r = w'\ r^{-1} = w\ r\ r^{-1} = w \pmod{c}$$

Así pues, el procedimiento descrito permite el cómputo eficiente de productos modulares para valores del parámetro r de la forma indicada anteriormente ($r = 2^n$). Nótese, sin embargo, que la técnica de Montgomery aplicada al cálculo de productos modulares no es capaz de eliminar todas las divisiones, puesto que las requeridas para el cómputo de los valores a' y b' continúan siendo necesarias. Por esta razón no es recomendable utilizar la reducción de Montgomery para el cálculo de un solo producto modular, debido a que el coste computacional aumentaría en vez de reducirse, sin embargo, es muy aconsejable para computar un gran número de productos modulares sucesivos, como es el caso del cálculo de las exponenciaciones modulares.

Exponenciación de Montgomery.

El algoritmo de exponenciación de Montgomery calcula $d \equiv a^b \pmod{c}$ con los desarrollos binarios de $c = (c_{n-1},...,c_0)$ y del exponente $b = (b_{t-1},...,b_0)$. También utiliza como entradas $r = 2^n$, con mcd (c, r) = 1, y el entero $c' = -c^{-1} \pmod{r}$.

Paso 1: Hacer $a_1 \equiv a\ r \pmod{c}$

Paso 2: Hacer $d_t \equiv r \pmod{c}$

Paso 3: Para $i = t-1, ..., 0$

Hacer $d_i = \text{Mont}(d_{i+1} \cdot d_{i+1})$

Si $b_i \neq 0$ entonces Hacer $d_i = \text{Mont}(d_i \cdot a_1)$

Paso 4: Salida $d = \text{Mont}(d_0)$

Como se puede apreciar en el algoritmo anterior el cálculo de la exponenciación únicamente requiere el cómputo de divisiones por c en los pasos 1 y 2, puesto que en las reducciones de Montgomery, representadas por Mont (...) las divisiones por r son simples desplazamientos lógicos.

El protocolo descrito en el algoritmo anterior utiliza el método binario, no obstante el cálculo de exponenciaciones puede generalizarse para el caso de cualquier método m-ario combinado con la reducción de Montgomery. Este método requiere el cálculo del mismo número de multiplicaciones que el método m-ario con el que se asocie, sin embargo, la reducción de Montgomery permite evitar el cómputo de divisiones, las cuales son sustituidas por simples desplazamientos lógicos.

Debido a la supresión de las operaciones de división, y a que los métodos m-arios conseguían una reducción en el número de operaciones de multiplicación, hacen que este método sea el más indicado para realizar las exponenciaciones modulares. No obstante, en puntos posteriores se realizaran algunas modificaciones de dicho método con el fin de simplificarlo más, si fuera posible.

3.2. IMPLEMENTACIÓN DEL ALGORITMO DE MONTGOMERY.

P. L. Montgomery introdujo un algoritmo eficiente para computar el producto modular $R = a \cdot b \pmod{n}$, donde a, b y n son números binarios de k bits. El algoritmo es particularmente idóneo para implementaciones en microprocesadores o DSP´s de propósito general, que son capaces de realizar operaciones aritméticas de forma rápida trabajando en potencia de dos.

El algoritmo de reducción de Montgomery computa el resultado del producto R, de tamaño k bits, sin realizar ninguna división por el módulo n, así sustituye la operación de división por n, por una división por un número potencia de dos, mediante una ingeniosa representación de la clase de residuos del módulo n.

Tomando $r = 2^k$, siendo k el número de bits, y de forma que se cumpla que $\text{mcd}(2^k, n) = 1$, podemos definir el residuo-n del valor a respecto al módulo n como:

$$a' = a \cdot r \pmod{n}$$

Si elegimos dos valores $a, b < n$, calculando sus residuos módulo n, podemos aprovechar una propiedad de la reducción de Montgomery para calcular el producto modular de la forma:

$$R' = a' \cdot b' \cdot r^{-1} \pmod{n}$$

Donde r^{-1} es el inverso de r módulo n ($r^{-1} \cdot r = 1 \bmod n$), y el resultado R' es el residuo-n del producto modular $R = a \cdot b \pmod{n}$. Para completar la implementación del algoritmo del producto modular sólo falta la cantidad n', que es el entero que cumple la propiedad:

$$r \cdot r^{-1} - n \cdot n' = 1$$

Tanto r^{-1} como n' son calculados mediante el algoritmo extendido de Euclides.

El producto de Montgomery $R' = a' \cdot b' \cdot r^{-1} \pmod{n}$ se calcula mediante el siguiente pseudocódigo:

ProMont (a' ,b', n)

Paso 1: $t = a' \cdot b'$

Paso 2: $m = t \cdot n' \pmod{r}$

Paso 3: $u = \dfrac{t + m \cdot n}{r}$

Paso 4: Si $u \geq n$ entonces $R' = u - n$, si no $R' = u$

Se comprueba que tanto el módulo como el cociente se realizan por un número que es potencia de dos ($r = 2^n$), por lo que se solventan con operaciones básicas de desplazamientos.

Este pseudocódigo también puede ser utilizado para calcular el producto $R = a \cdot b \pmod{n}$, debido a que n es un número impar, por definición del algoritmo RSA, y así se cumple que mcd (r, n) = 1.

3.2.1. Algoritmo RSA basado en el producto de Montgomery.

Si nos basamos en el algoritmo anteriormente definido para el producto de Montgomery, podemos generar el pseudocódigo necesario para la implementación del algoritmo RSA, es decir, el código que realiza la exponenciación:

$$X = s^e \pmod{N}$$

RSA (s, e, N) = s^e (mod N)

Paso 1: Obtención constante $C = 2^{2k} \bmod N$.

Paso 2: X' = ProMon (1, C)

Paso 3: s' = ProMon (s, C)

Paso 4: Para I = k-1 hasta 0

X' = ProMon (X', X')

Si e(i) = 1 entonces X' = ProMon (s', X')

Paso 5: X = ProMon (X', 1)

Donde e(i) es el bit i-ésimo del exponente y ProMon es la implementación del producto de Montgomery antes definido.

Se observa que es un código fácil de implementar con sucesivas llamadas al bloque ProMon, sin embargo, tenemos una división por N en el paso previo 1, que se desarrollará con mayor detalle más adelante.

A la hora de la implementación hardware podemos comprobar que el pseudocódigo del algoritmo del producto de Montgomery contiene algunos productos que nos van a ralentizar nuestro método, así tras una nueva búsqueda, se encontró un nuevo pseudocódigo más fácilmente implementable para el producto de Montgomery:

Mont (A, B, N) = A $\cdot$ B (mod N)

Paso 1: u = 0

Paso 2: Para i = 0 hasta i = k-1

u = u + $A_i \cdot B$

Si u es impar entonces u = u + N

u = u / 2

Paso 3: Si u > N devuelve u-N,

si no devuelve u

Obviamente se trata de un algoritmo más simple, que puede ser implementado en hardware de una forma mucho más fácil. Simplemente con contadores, un registro de desplazamiento (para la división por dos), y algunos sumadores quedará totalmente implementado este nuevo algoritmo **Mont (A, B, N)** para el producto de Montgomery de dos enteros de k bits. Por todo esto, definitivamente éste será el método a usar, en el que nos vamos a basar para toda la implementación hardware del algoritmo RSA.

RSA (s, e, N) = s^e (mod N)

Paso 1: Obtención constante $C = 2^{2k} \bmod N$.

Paso 2: X' = Mont (1, C, N)

Paso 3: s' = Mont (s, C, N)

Paso 4: Para I = k-1 hasta 0

X' = Mont (X', X', N)

Si e(i) = 1 entonces X' = Mont (s', X', N)

Paso 5: X = Mont (X', 1, N)

Las ventajas de este método descrito para la implementación del algoritmo RSA se pueden enumerar:

- La condición a chequear en el algoritmo **Mont** es si el resultado es par o impar, que se verifica rápidamente comprobando si un bit está a cero o uno.
- El único producto que realiza es en el paso 2, donde se multiplica un bit por k bits, que es una operación muy fácil a realizar comprobando si el bit es uno o cero.
- La división se puede sustituir por un desplazamiento al ser una potencia de dos.

Como desventajas podemos señalar las siguientes:

- Este algoritmo no puede trabajar si el módulo es par, cosa que no ocurrirá nunca por la propia naturaliza de un sistema RSA, N siempre es impar.
- El número de periodos de tiempo que tarda en realizar la exponenciación será variable, esto es debido a que está relacionado con el número de bits a uno que tenga el exponente, así el número de intervalos de tiempo variará entre k^2 y $2k^2$.

3.2.2. Diagramas de bloques.

En este apartado se mostrará un esquema general de bloques que determinan el funcionamiento del algoritmo utilizado para la implementación del sistema RSA. Como ha quedado claro anteriormente, se implementará un algoritmo de exponenciación modular basado en el método de Montgomery antes descrito.

Primeramente se va a centrar el estudio en el diagrama de bloques utilizado para la implementación del producto de Montgomery. En el citado algoritmo se observó que se necesitaba un registro para almacenar el resultado, algunos sumadores y un registro de desplazamiento para realizar la división. En el siguiente esquema de bloques se detallan los elementos necesarios para el funcionamiento del algoritmo:

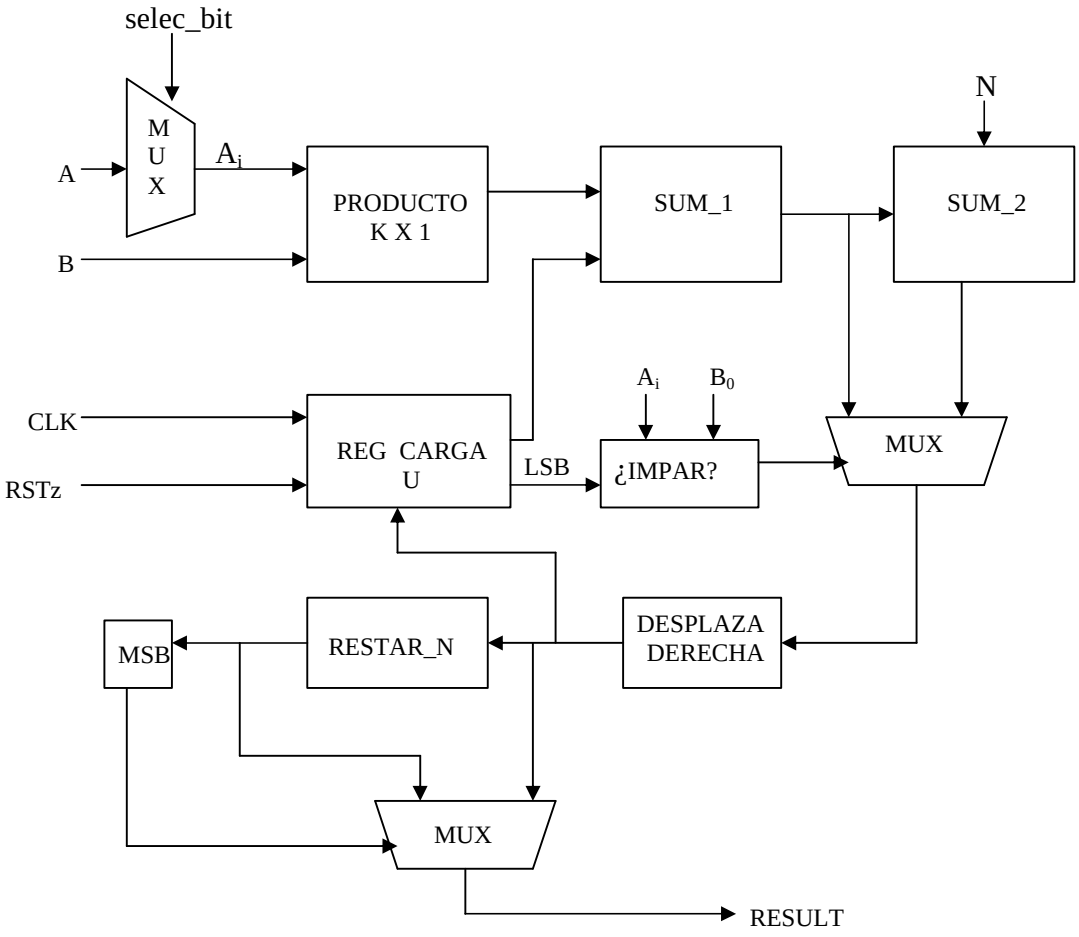

Figura 6: Esquema del bloque de Montgomery.

En este esquema se representa el proceso del producto modular de Montgomery que se realiza en cada periodo de tiempo. Así en cada ciclo de reloj se requiere de un bit del primer operando A que se multiplica por el segundo operando B, y el resultado se suma al registro de carga U, que servirá para ir almacenando el resultado. El resultado de la suma (SUM_1) se pasa al registro de desplazamiento si es impar, en caso contrario lo que se le pasa será lo anterior más N (SUM_2). Esta condición se conoce antes de la primera suma debido a que se calcula mediante un bloque que nos indica si es par o impar partiendo de los bits menos significativos de U y de B, y el bit i-ésimo del operando A, realizando la operación:

$$(U_0)\ \text{XOR}\ (A_i\ \&\ B_0)$$

Además tenemos un bloque que realiza la división entre dos, es decir, realiza un desplazamiento hacia la derecha, y el bloque que resta el módulo N, en el caso que fuese necesario, realizando el complemento a dos del segundo operando.

En el diagrama de bloques anterior también observamos tres multiplexores, uno de k x 1, que selecciona el bit i-ésimo de A con el que se opera, y dos de 2 x 1, que seleccionan la suma u + N y la resta u – N según si se cumplen las condiciones U impar, y U > N respectivamente (pasos 2 y 3 del algoritmo del producto de Montgomery).

Después de k ciclos de reloj obtenemos el resultado del producto modular de Montgomery de dos enteros (A y B) de k bits, respecto del modulo (N) también de k bits.

Ahora nos centraremos en el diagrama de bloques del algoritmo RSA implementado mediante la exponenciación modular de Montgomery:

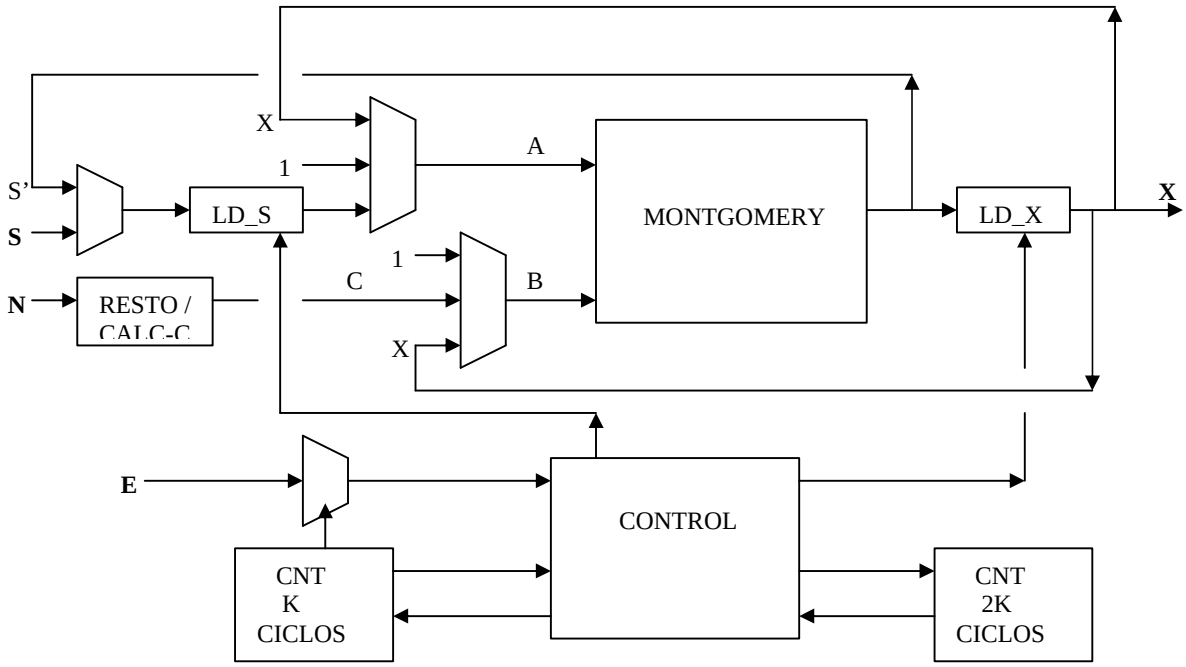

Figura 7: Esquema sistema RSA.

En el esquema anterior aparece un bloque de control, que gestiona las entradas al bloque que realiza el producto de Montgomery, con la ayuda de dos contadores. Las entradas van cambiando dependiendo del instante de tiempo, por lo que se utilizan diferentes registros para ir almacenando los valores de los mismos (S, X, C, 1). El bloque de control también gestiona los diferentes multiplexores y las señales para activar los diferentes registros de carga.

Primeramente hay que calcular la constante C definida como: $C = 2^{2k} \bmod N$, para el cálculo de esta constante se emplearon diversos métodos, que más adelante se describirán. Después comienza el algoritmo en sí, es decir, el controlador principal empieza a gobernar los diferentes multiplexores, los cuales le dan las entradas al bloque de Montgomery. Se dispone de dos contadores que le indican al controlador cuando tiene que cambiar las entradas a dicho bloque de Montgomery.

Además existen otras señales que son globales a todo el sistema y que por simplificar no aparecen en el esquema anterior, se trata de las entradas de reloj y reset

(CLK y RSTz), de una entrada que habilita el calculo de la exponenciación (ENABLE), y una señal de salida que indica el final del proceso (FIN).

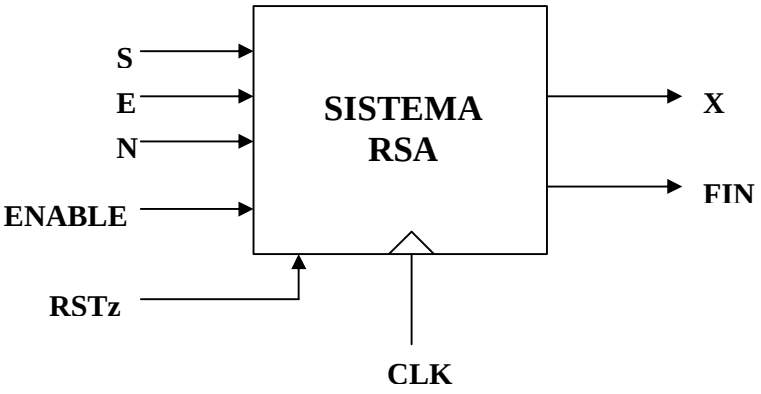

Figura 8: Diagrama del sistema RSA

3.2.3. Descripción del código Verilog del sistema RSA.

Ahora nos centraremos en el funcionamiento interno del sistema RSA, es decir, se estudiará en detalle los bloques más importantes del algoritmo usado, así como sus principales ventajas y desventajas.

Al final de este documento se incluye en el anexo el listado del código Verilog que compone todo el sistema RSA, con todas las modificaciones realizadas en la evolución de la generación de dicho código. Cada bloque del sistema se asocia con un módulo del programa realizado en Verilog, de tal forma que existirán bloques más importantes que engloben a otros subbloques o módulos menos importantes formando una estructura jerárquica. El siguiente dibujo muestra dicha estructura jerárquica, en la que partiendo de un módulo_top, que se ha denominado **codec_rsa**, se observan los diferentes submódulos que componen el código final.

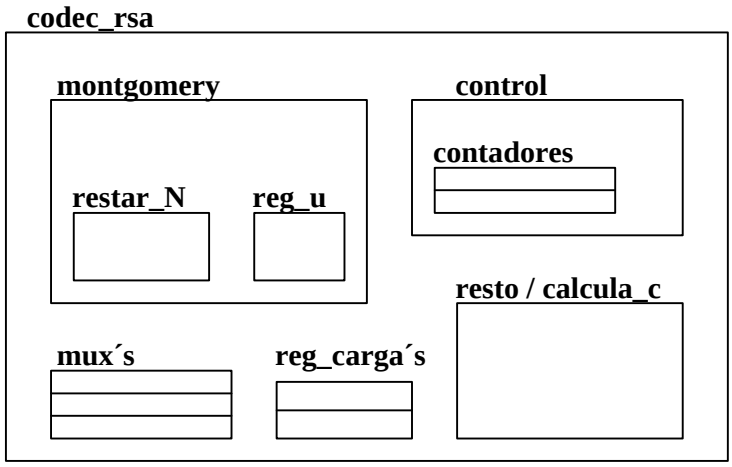

Figura 9: Estructura de módulos del sistema RSA.

Podemos observar que el módulo **codec_rsa** realiza las llamadas a los diferentes submódulos o bloque que utiliza, como es el caso de los módulos de **Montgomery, control, resto, mux** y los diferentes **reg_carga**, también estos módulos pueden contener a otros como se puede apreciar.

Anteriormente se detalló el funcionamiento del bloque que realizaba el producto de Montgomery en cada intervalo de tiempo, ahora vamos a detenernos en el bloque de control, el modulo controlador es el más importante de todo el conjunto porque lleva todo el proceso de encriptación/decriptación, es decir, manda las diferentes señales de control a los demás bloques para poder realizar todo el proceso. Dichas señales de control son las siguientes:

- Incremento y reset del contador de los bits del exponente: Inc_T3 y RST_T3.
- Reset del contador de ciclos del proceso de multiplicación de Montgomery: RST_T5.
- Señal de carga de los registros: LD_X, LD_S.
- Selección de los multiplexores de entrada al bloque de Montgomery: Selec_mux_A y Selec_Mux_B.
- Señal que indica el fin del proceso: Fin.

Así, el controlador irá dominando todas estas señales en función de las salidas de los contadores (T3 y T5) y de la señal de ENABLE, de forma que consiga el mismo comportamiento que el pseudocódigo mostrado para el algoritmo RSA. A continuación se muestra la máquina de estados que simula el funcionamiento del controlador:

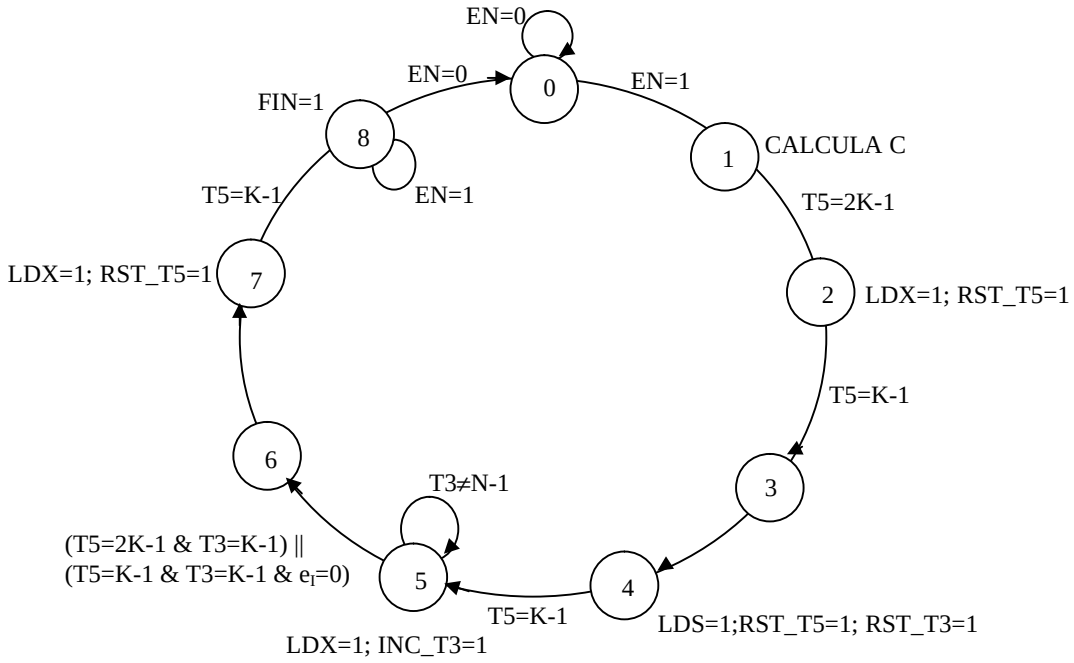

Figura 10: Máquina de estados del controlador principal.

En la máquina de estados anterior podemos comprobar que el proceso comienza cuando el controlador recibe la señal de ENABLE a nivel alto, y comienza a cambiar de fases en función de los contadores T5 y T3 (de 2K y K estados respectivamente) y de los bits del exponente. Se puede apreciar que las diferentes fases de la máquina de estado están directamente relacionadas con los pasos del algoritmo RSA:

FASE 1 ⟷ Paso 1: Obtención constante $C = 2^{2k} \bmod N$.

FASE 2 ⟷ Paso 2: X’ = Mont (1, C, N)

FASE 3,4 ⟷ Paso 3: s’ = Mont (s, C, N)

FASE 5 ⟷ Paso 4: Para I = k-1 hasta 0
X’ = Mont (X’, X’, N)
Si e(i) = 1 entonces X’ = Mont (s’, X’, N)

FASE 6,7 ⟷ Paso 5: X = Mont (X', 1, N)

Otro bloque importante sobre el que todavía no se ha comentado nada es el que realiza el cálculo previo de la constante C:

$$C = 2^{2k} \bmod N$$

Para calcular esta constante se utiliza un módulo que realiza la división y halla el resto de la misma. Como se trata de una operación no sintetizable se ha realizado un algoritmo basado en sucesivos desplazamientos y sumas que consiguen el resto de la división de una forma eficaz.

Como nota negativa podemos destacar que tarda un determinado número de ciclos en realizar dicha operación y además al sintetizar ocupa un área considerable como se comprobará posteriormente cuando examinemos los resultados.

Debido a lo anteriormente comentado se optó por cambiar dicho módulo que calcula el resto por un componente de la librería de Synopsys (**DW_div)** que calcula esta operación matemática de una forma más rápida y eficaz. No obstante, todo esto se contrastará a continuación cuando se examinen todos los resultados conseguidos con nuestro programa tanto a nivel funcional como a nivel de síntesis.

4. RESULTADOS.

En este apartado se van a mostrar los diferentes resultados obtenidos tras realizar las pruebas tanto funcionales como de sintetización. Una vez elaborado todo el código se procedió primero a comprobar su funcionalidad, es decir, comprobar que realmente realizaba lo que se pretendía, para ello se compararon los resultados obtenidos con el código Verilog con otros resultados obtenidos con Matlab, y después se pasó dicho código por el sintetizador, con el fin de ver su comportamiento y sus características hardware tales como áreas, retrasos, número de puertas, biestables, etc.

Antes de entrar en resultados cabe destacar que nuestro sistema RSA está parametrizado de forma que el tamaño de la palabra a encriptar o desencriptar, así como del exponente y del módulo, son totalmente variables. Se tomará un amplio rango de valores de dicho parámetro (‘N’ en nuestro código) con el que obtendremos las diferentes pruebas y resultados.

En el apartado anterior se vio que se tienen diversas modificaciones del programa con el fin de obtener los resultados óptimos, así se mostrarán dichos resultados para las diferentes variaciones del programa, tanto las realizadas anteriormente como las que se expondrán después, con el fin de comparar los diferentes resultados obtenidos.

Se va a estructurar este apartado en la exposición de los resultados obtenidos con el programa mostrado como hasta ahora, es decir, con la utilización del módulo Montgomery que efectúa cada operación en un ciclo de reloj, y en una variante del programa en la que se cambiará dicho módulo, de forma que se disminuya el número de ciclos de reloj en obtener el resultado final.

4.1. RESULTADOS PARA EL MÓDULO MONTGOMERY.

Hasta ahora se ha descrito el funcionamiento del sistema RSA en función a un módulo que realiza el producto de Montgomery de forma que en cada ciclo de reloj efectuaba una operación basándose en algunos sumadores y un registro de desplazamiento. Se trata de un módulo muy simple, que consta de una lógica muy simple en la que el retraso es bastante pequeño. En función a este bloque se definió un controlador que gestionaba las diferentes entradas para realizar dicho producto de Montgomery.

Para comprobar la correcta funcionalidad del sistema se realizó un programa en Matlab (también adjunto en el anexo) donde se generó las diferentes entradas del sistema RSA, de forma que cumplieran todas las especificaciones necesarias en dicho sistema, y en el que se obtienen las diferentes salidas ante dichas entradas. Este programa simula el comportamiento del sistema realizado en Verilog y trabaja con un gran número de pruebas para poder comparar todos los resultados.

Una vez conseguidos todos los valores de entrada, se generaron listas de cien pruebas en las que se obtenían los diferentes resultados, comprobando que las salidas coincidían en la simulación Matlab y en el programa Verilog. A continuación se muestra la variación del número de iteraciones para las diferentes pruebas realizadas con valores distintos del parámetro de ancho de la palabra:

N	(Max - Min) Ciclos	Media
8	111 - 147	129
16	543 - 399	471
32	1919 - 1503	1711
64	5247 - 4671	4959
128	18431 - 17663	18047

Tabla 4.1: Número de ciclos de reloj.

Podemos comprobar que el número de iteraciones varía para los diferentes resultados con el mismo parámetro. Esto es así debido a que dicho número de operaciones estaba relacionado con el número de bits a uno que tenía el exponente, por lo que el resultado tiene un número de ciclos de reloj aleatorio, por lo que se ha mostrado en la tabla anterior el máximo y mínimo, y la media de ciclos de reloj, para cada lista de resultados obtenidos para cada valor del parámetro N.

En esta tabla se observa que el número de iteraciones se dispara conforme aumentemos el parámetro N, y que sigue una relación aproximada con dicho parámetro de la forma $(1.5 \cdot N^2)$.

Se comprobó que todos los resultados coincidían para cada una de las entradas entre la simulación Matlab y el programa realizado en Verilog, en el anexo final se puede observar alguna lista de resultados obtenida para diferentes entradas.

A continuación vamos a exponer la otra parte en la que se centran los resultados de este trabajo, es decir, los resultados de sintetización del código realizado en Verilog para nuestro sistema RSA.

Estos resultados van a mostrar aspectos de la sintetización, tales como el área ocupada, número de puertos necesarios, número de biestables utilizados, o el número de celdas necesarias. En el área además se realizan divisiones según sea combinacional, secuencial, o de interconexión.

Otro parámetro importante también es el SLACK, que nos da una referencia de los retrasos producido por la lógica combinacional, debido a que muestra el tiempo restante hasta que el siguiente dado tenga que estar preparado, podemos verlo mejor en la siguiente figura:

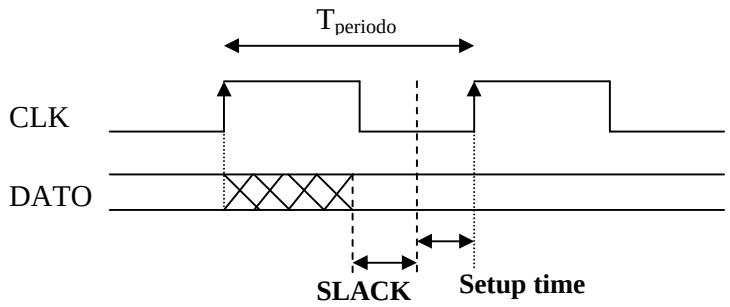

Figura 11: Definición SLACK.

Es imprescindible que aparezca un valor positivo de SLACK, por lo que deberemos modificar el valor del periodo $T_{periodo}$, de forma que sea lo suficientemente grande como para que el retraso producido por la lógica combinacional no altere el correcto funcionamiento del dispositivo. Con tal fin, lo ideal es conseguir un valor intermedio para el periodo del ciclo de reloj que sea lo suficientemente grande para que el SLACK sea positivo, pero sin hacer el dispositivo muy lento.

A continuación se muestra una tabla en la que aparecerán todos estos datos procedentes de la síntesis de nuestro código:

		ÁREA (mm^2)							
N	T_{per}	Combinac	Secuenc	Interconex	TOTAL	Puertos	Celdas	Biestab	SLACK
8	20	92164	38001	23327	153494	36	1001	84	2.14
16	20	191009	69342	46407	306758	68	2044	158	- 1.85
16	40	125179	68959	21244	225383	68	1139	158	3.72
32	50	434725	129820	111726	676271	132	4467	304	0.61
64	100	467503	236800	116488	820792	270	4289	560	9.43
128	200	1027244	465246	257698	1750190	516	9416	1022	41.1

Tabla 4.2: Resultados de sintetización.

En esta tabla se aprecia fundamentalmente que los valores de las áreas conseguidos son muy elevados, principalmente su parte combinacional, asimismo, el valor de celdas necesario es extremadamente grande para los valores mostrados del parámetro N.

No se procedió con sintetizaciones para valores mayores de N porque el tiempo necesario para la obtención de estos resultados ya era excesivo, y los valores obtenidos para el área y el tamaño del periodo de reloj se hacían muy grandes.

En lo referente al tamaño del periodo del ciclo de reloj, se comenzó con un valor de $T_{periodo}$=20 ns, se aprecia que se obtiene un SLACK positivo para el valor N=8, no siendo así para los sucesivos valores de dicho parámetro. Esto es debido a que el tiempo del ciclo necesario para que el sistema funcione es insuficiente con esa cantidad, por lo

que se incrementó a $T_{periodo}$=40 ns con N=16, $T_{periodo}$=50 para N=32, $T_{periodo}$=100 para N=64 y $T_{periodo}$=200 ns para N=128, aunque podría reducirse porque el slack es muy grande. En lo que se refiere a nuestro diseño, lo que es afectado debido a este parámetro es la frecuencia de funcionamiento, que tendrá los siguientes valores:

N	Frecuencia (MHz)
8	50
16	25
32	20
64	10
128	5

Tabla 4.3: Frecuencias de funcionamiento.

Podemos apreciar como la frecuencia de funcionamiento del sistema decrece gradualmente según vamos aumentando el parámetro N, lo cual resulta comprensible, debido a que cuantos más bits tenemos, más lógica combinacional será necesaria, por lo que se producirá un mayor retraso asociado a dicha lógica, con lo que el tiempo del periodo del reloj necesario será mayor.

En los resultados de sintetización anteriores se aprecia que el $T_{periodo}$ tiene una influencia muy grande, ya que cuando tomamos valores elevados del tamaño del periodo para un determinado valor del parámetro N, es decir, el slack es relativamente alto, los valores conseguidos del área ocupada se reducen. Esto es así porque el programa encargado de realizar la síntesis utiliza como restricción el valor del tiempo de periodo, con lo cual, si dicho tiempo es bajo el sintetizador busca las condiciones de área que cumplen con dicho valor, que serán mayores que las necesitadas para un periodo de reloj más holgado.

Lo anteriormente expuesto lo podemos constatar con la tabla de resultados de la sintetización (Tabla 4.2), donde se aprecia que cuando aumentamos el tamaño del periodo se reduce considerablemente el área ocupada, especialmente el área combinacional.

Dicho esto, se aclara que se han tomado valores que consigan un punto intermedio entre restricciones de tiempo y de área, ya que se están obteniendo resultados generales. Si buscásemos que nuestro sistema funcionase a una frecuencia determinada y de menor valor que las obtenidas, se comprobaría que el área aumentaría en gran tamaño, y se conseguiría un diseño peor del mencionado.

Con el fin de descubrir la causa del elevado valor de área ocupado por nuestra síntesis, se muestra a continuación los valores porcentuales del área ocupada por los diferentes bloques o módulos en las celdas obtenidas tras la sintetización de nuestro programa:

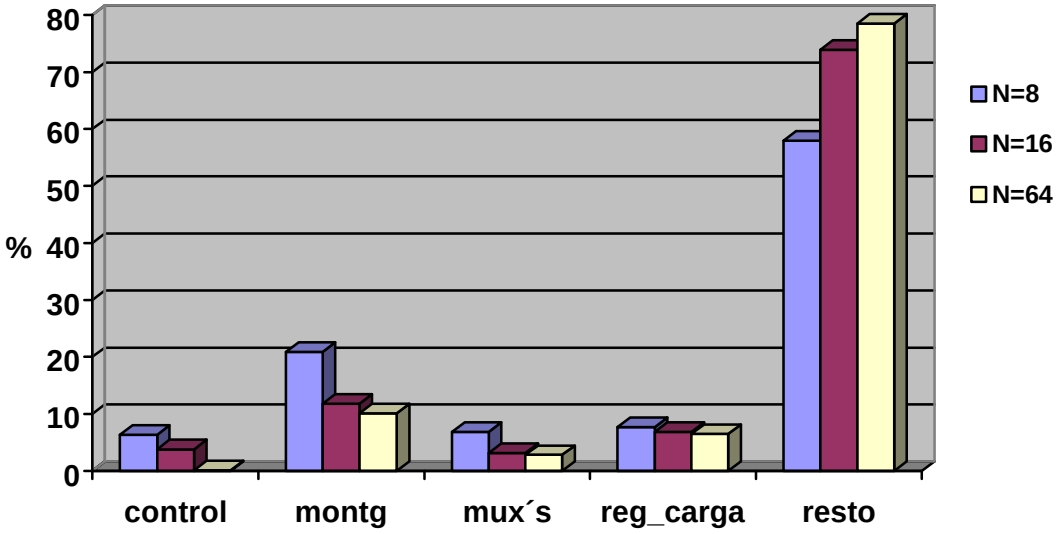

Figura 12: Área ocupada por cada bloque.

Podemos observar que casi todo el área es aportada por el bloque que calcula el resto en el paso previo (2^{2k} mod N), además, según aumenta el tamaño de los datos la influencia de este bloque es mayor con respecto al resto, por lo que nos lleva a pensar que es este bloque **resto** el responsable del aumento del periodo de reloj también, ya que el único valor que aumenta según vamos incrementando el parámetro N es el área ocupada debido a dicho bloque.

Todo esto nos lleva a sustituir este bloque por el que se referenció anteriormente de la librería de Synopsys, para obtener unos mejores resultados, tanto en restricciones de área, como de frecuencias.

4.1.1. Módulo de la librería de Synopsys.

Para realizar este cambio del módulo **resto** por el módulo **DW_div** de la librería de Synopsys no es necesario ningún cambio en otras partes del programa, sólo tenemos que realizar la llamada a este módulo de librería en nuestro bloque principal, sin tener que realizar la síntesis del mismo. El nuevo resto se calculará de forma más rápida que con el módulo anterior, y por supuesto necesita menos área para su implementación.

A continuación se mostrarán los nuevos resultados obtenidos para esta nueva composición de nuestro programa, se puede observar que la parte funcional no cambia mucho, ya que el programa sigue funcionando correctamente, los resultados coinciden con la simulación Matlab, y el número de ciclos necesarios es parecido al anterior, disminuye algo debido a que el nuevo bloque insertado precisa menos ciclos para obtener dicho resultado.

N	(Max - Min) Ciclos	Media
8	133 – 97	115
16	513 – 369	441
32	1857 – 1441	1649
64	5121 – 4545	4833
128	18177 - 17409	17793

Tabla 4.4: Número de ciclos con módulo DW_div.

Volvemos a comprobar que el número de ciclos necesitados es muy elevado, debido a que tan solo se ha reducido en el mismo valor que el que toma el parámetro N en cada caso.

Ahora veremos el comportamiento de la nueva síntesis utilizando el módulo anteriormente descrito:

		ÁREA (μm²)							
N	T_{per}	Combinac	Secuenc	Interconex	TOTAL	Puertos	Celdas	Biestab	SLACK
8	20	9919	14450	3985	28354	36	181	36	12.76
16	20	16652	27318	7106	51077	68	320	68	12.73
32	20	28319	53016	12884	94220	132	573	132	12.25
64	50	51468	105356	21821	178645	270	1146	270	21.69

Tabla 4.5: Síntesis con módulo DW_div.

En la comparación de estos resultados con los anteriormente obtenidos destaca fundamentalmente la reducción del área, principalmente el área combinacional, aunque también la secuencial y la de interconexión. Podemos ver que se produce una reducción de hasta siete veces menor que en el caso anterior, esto es así debido a que la lógica combinacional se ha reducido en gran medida al quitar el módulo que realizaba la división mediante desplazamientos. También se aprecia que el número de celdas también se ha reducido bastante.

La otra diferencia con los resultados anteriores es que la frecuencia de funcionamiento del sistema es mayor que en el caso anterior, ya que con T_{per}=20 ns se consiguen resultados muy favorables del parámetro slack, sin embargo se procedió a aumentar el periodo a 50 ns para ver como cambiaban las características del sistema.

Estos resultados son bastante coherentes, ya que van asociados a una disminución de la lógica, y con ello del retraso provocado por las puertas, que es lo que se pretendía insertando dicho módulo de la librería.

Hasta ahora se ha conseguido una reducción drástica del área utilizada, pero seguimos teniendo un problema que no se ha solucionado, se trata del número de ciclos que tarda el sistema en conseguir el resultado. Este número de ciclos es muy elevado debido a que el algoritmo de reducción de Montgomery hasta ahora planteado realiza una iteración en cada ciclo, por lo que cuando el tamaño de las palabras va aumentando el número de ciclos llega a una cifra bastante grande.

El propósito de la continuación de este capítulo será exponer alguna modificación de dicho algoritmo con el fin de conseguir más rápido el resultado final, esta variación del programa llevará implícita un aumento de la lógica combinacional con toda seguridad, debido a que el algoritmo hasta ahora utilizado es bastante simple y ocupa muy poca memoria, como hemos observado a lo largo de este apartado, por lo que perderemos optimalidad en cuanto a lo que área y frecuencia de funcionamiento se refiere.

4.2. MEJORAS DEL ALGORITMO.

En este apartado se pretende una reducción del número de ciclos que se necesita para la realización total del proceso de exponenciación modular. Hasta ahora se ha comprobado que dicho número de periodos de reloj está relacionado con la anchura de la palabra de datos de entrada mediante la forma:

$$T_{total} \approx 1.5\ k^2 = 1.5\ N^2$$

Siendo 'k' el ancho de los datos, que coincide con el parámetro N de nuestro código. Esto es así debido a la propia estructura del bloque que realiza el producto de Montgomery, donde en cada periodo de tiempo se realiza el producto de un bit por la consiguiente palabra de entrada, y su posterior suma y desplazamiento. Se recuerda que la estructura de dicho algoritmo era de la forma:

Paso 1: $u = 0$

Paso 2: Para $i = 0$ hasta $i = k-1$

$u = u + A_i \cdot B$

Si u es impar entonces $u = u + N$

$u = u / 2$

Paso 3: Si $u > N$ devuelve $u-N$,

si no devuelve u

Por lo que en 'k' intervalos de tiempo se realiza el producto de Montgomery de dos entradas de 'k' bits de tamaño. Para valores grandes de 'k' (o del parámetro N) el algoritmo RSA se vuelve muy lento, debido a que hay que realizar todo este proceso otras $1.5 \cdot k$ veces de media, ya que dependerá del número de bits a uno del exponente. Así lo que se pretende ahora es presentar una modificación de dicho algoritmo, de forma que sacrificando su estructura simple, podamos conseguir una mayor velocidad en la obtención del dato de salida. Para ello se realizó una máquina de estados que consigue decrementar los 'k' ciclos necesarios para la obtención del producto de Montgomery por tan sólo cuatro ciclos de reloj, independientemente del tamaño de los datos de entrada ('k'):

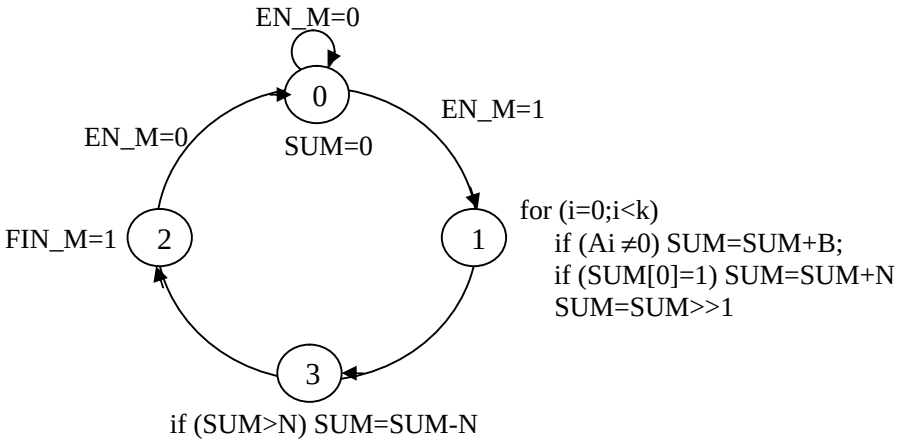

Figura 13: Máquina de estados del nuevo bloque Montgomery.

Como se observa en el diagrama de estados anterior se reduce el número de ciclos de reloj a cuatro, pero se carga mucho más la lógica asociada a cada ciclo. Ahora tenemos dos señales asociadas al principio y fin del proceso de realizar el producto de Montgomery (EN_M y FIN_M), son una señal de entrada y otra de salida que se utilizaran para la comunicación de este módulo con el controlador principal que lleva a cabo todo el proceso.

Como consecuencia de la modificación de este módulo se tiene que variar el diagrama de estados del controlador principal. Además ahora sólo necesitamos un contador en vez de los dos anteriores debido a que el nuevo bloque de Montgomery lo gobernaremos con las señales EN_M y FIN_M antes mencionadas.

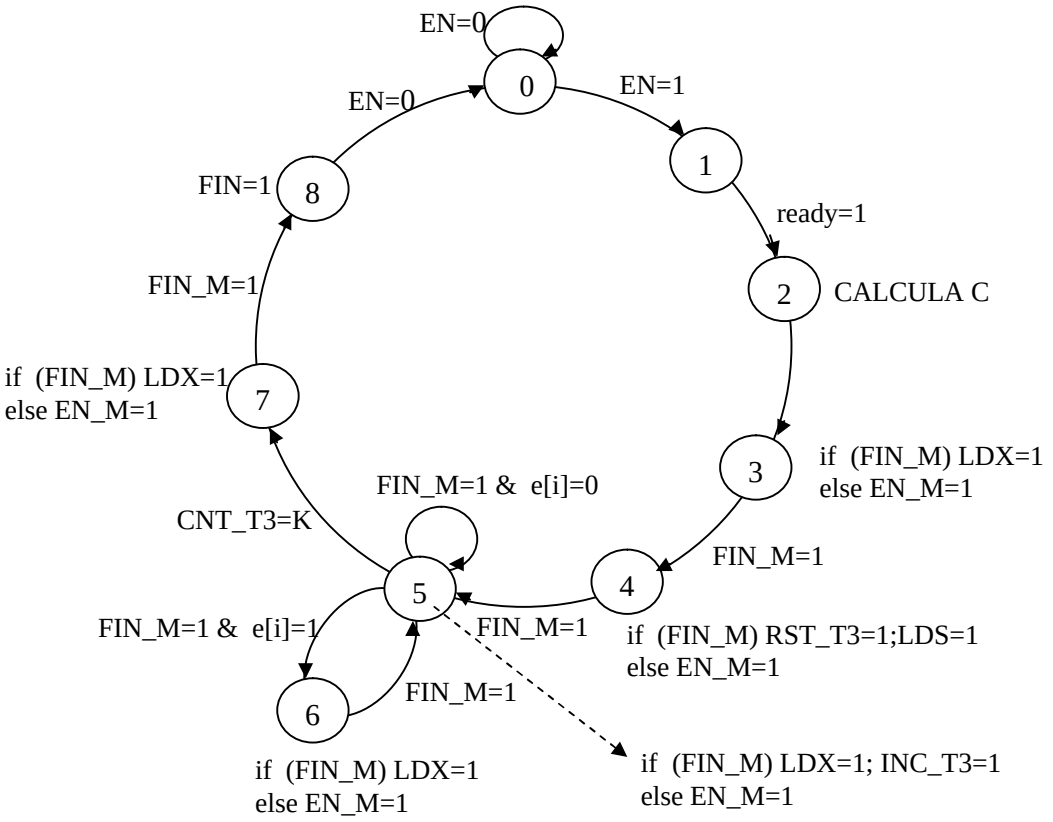

Figura 14: Máquina estados sistema RSA.

Así quedaría la máquina de estados del controlador de nuestro sistema RSA, en ella podemos observar que hacemos referencia al bloque que realiza el producto de Montgomery con las señales EN_M y FIN_M. Además se ha añadido un estado (FASE 2) que su única misión es esperar que los registros que almacenan las entradas capturen dichos valores, avisando con la señal ‘ready’. Esta máquina de estados tiene un funcionamiento más simple que la anterior, debido a que ahora no tenemos que comparar todos los bits de un contador para pasar de un estado a otro, sino que en la mayoría de los casos comprobando el estado de un único bit ya es suficiente.

Todo el código debidamente explicado se encuentra en el anexo a este documento, donde se podrá observar en detalle todo lo anteriormente mencionado.

Tras realizar las pruebas de someter al código anterior a listas de valores de entrada, con diferentes valores del parámetro N, se obtuvieron los siguientes resultados:

N	(Max - Min) Ciclos	Media
8	71 - 63	67
16	121 - 85	103
32	225 – 173	199
64	433 – 357	395
128	823 - 767	795
256	1607 – 1459	1533
512	3113 – 2945	3029

Tabla 4.6: Número de ciclos para nuevo sistema RSA.

Podemos ver en la tabla anterior que el número de ciclos ha disminuido en gran medida comparado con los resultados anteriores, siendo la progresión con el parámetro N bastante inferior a la que se obtenía con el antiguo código de Montgomery. Ahora el número de ciclos tiene una relación aproximada en media (debido a que el tiempo total está asociado con el número de bits a uno del exponente), con el parámetro N de la forma:

$$T_{total} \approx 4 \cdot (1.5 \cdot N + 3)$$

Hemos pasado de una relación cuadrática a una relación lineal con el tamaño de los datos, por lo que el tiempo total en obtener el dato de salida se reduce considerablemente, sobre todo para valores grandes del parámetro N. A continuación se muestra una gráfica donde se aprecia la diferencia entre los tiempos totales de los algoritmos antiguo y nuevo, donde podemos ver las escalas lineal (nuevo código) y cuadrática (antiguo).

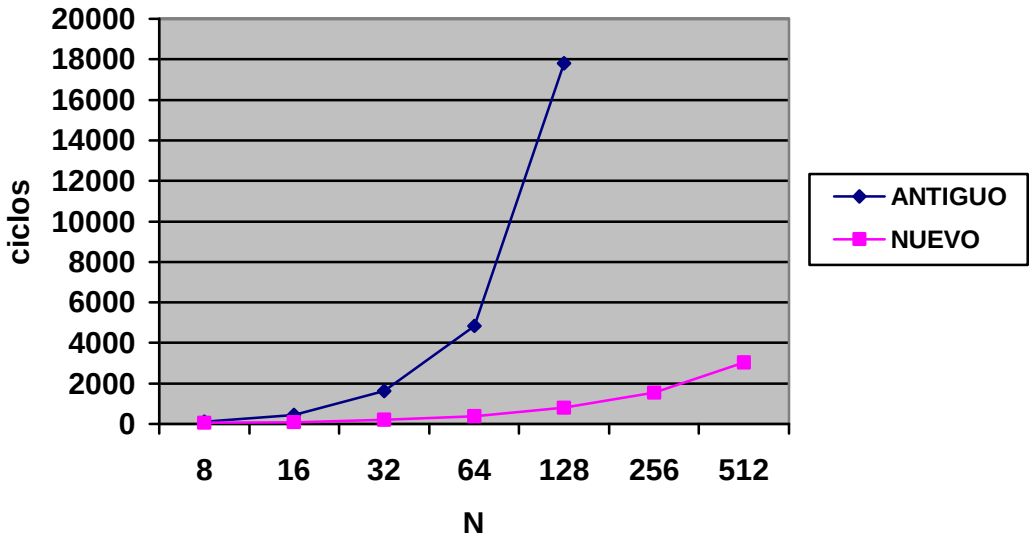

Figura 15: Comparación del número de ciclos.

Una vez conseguida la reducción del número de ciclos, vamos a ver la repercusión en el resto de resultados, es decir, como afecta la modificación efectuada en la sintetización, para ello se procede a mostrar los resultados de la síntesis:

		ÁREA (μm^2)							
N	$T_{periodo}$	Combinac	Secuenc	Interconex	TOTAL	Puertos	Celdas	Biestab	SLACK
8	20	337373	30812	75783	443969	36	3532	80	-31.33
8	60	254982	30758	62434	348174	36	2721	80	0.21
16	100	1420873	58276	324734	1803884	68	15184	152	-17.68
16	150	909090	57912	242449	1209451	68	10534	152	0.11
32	300	1919482	115269	526315	2961066	132	21435	286	1.87

Tabla 4.7: Resultados de sintetización.

Como se observa en la tabla anterior, la primera diferencia que se aprecia es que se ha tenido que variar el tiempo del periodo, es decir, con el código de Montgomery anterior y el módulo de la librería de Synopsys DW_div, se hicieron todas las pruebas con un periodo de reloj de 20 ns, y ahora se aprecia que con dicho periodo no se obtiene

un diseño correcto, ya que obtenemos un slack negativo, es decir, el retraso debido a la lógica es mayor que el tamaño del periodo de reloj. Así la principal consecuencia que observamos es que ha variado la frecuencia de funcionamiento del sistema, por ejemplo para N=8 se hizo una prueba con T_{per}=20 ns, pero se tuvo que variar hasta T_{per}=60 ns para conseguir un slack positivo. Igualmente con N=16, donde después de probar con T_{per}=100 ns se consiguió un diseño correcto con T_{per}=150 ns.

Con este diseño conseguimos frecuencias de funcionamiento que varían entre 16.6 y 3.3 MHz, necesitando para valores superiores del parámetro N frecuencias aún menores.

Otra característica que también se aprecia es que el área ocupada por el diseño también ha aumentado en gran medida con respecto al diseño anterior, como ocurre con los biestables y el número de celdas. Podemos contrastar las diferentes áreas ocupadas por nuestro diseño en los dos casos hasta ahora vistos, es decir, para el diseño anterior con el bloque de Montgomery que necesitaba un mayor número de ciclo de reloj y para el diseño que se acaba de exponer:

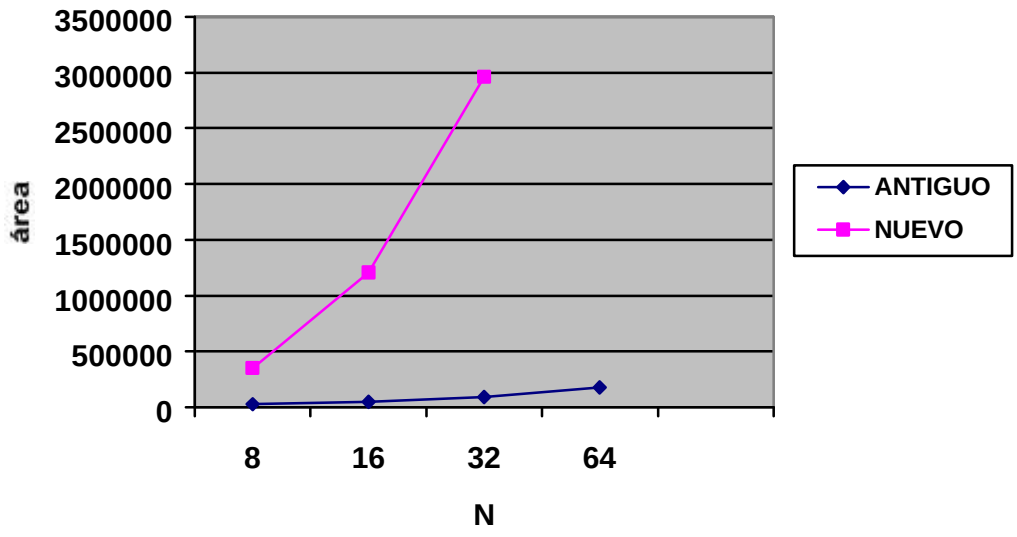

Figura 16: Comparación entre áreas ocupadas.

Se observa que según va aumentando el parámetro N, la diferencia entre áreas se va multiplicando.

El aumento del área estará íntimamente relacionada con la configuración del nuevo bloque que realiza el producto de Montgomery, para ver como influje la aportación de cada bloque al área consumida por cada celda, se expone el siguiente gráfico:

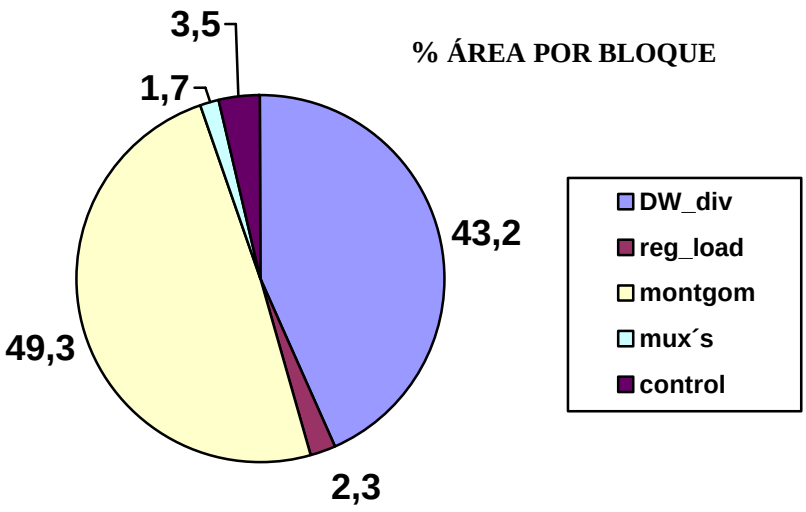

Figura 17: Porcentaje del área de celda consumida por cada bloque.

Se observa que los porcentajes más altos son los ocupados por el bloque de Montgomery (49.3%) y el bloque divisor DW_div (43.2%). Era lógico pensar que el módulo de Montgomery contribuiría en gran medida al aumento del área, no obstante también se aprecia que el módulo divisor sigue ocupando un valor alto de área ocupada.

Así podemos concluir diciendo que con la modificación realizada en este bloque se ha conseguido una reducción considerable en el número de ciclos de reloj del tiempo total en realizar el proceso de la exponenciación modular, sin embargo, el área ocupada y el número de puertas ha vuelto a aumentar, y además se ha tenido que variar la frecuencia de funcionamiento del sistema, debido a que se necesita un periodo más grande del requerido antes.

4.2.1. Nuevo bloque divisor.

En esta sección se va a tratar sobre uno de los bloques que más problemas nos ha dado a lo largo de la realización de nuestro sistema, se trata del bloque encargado de calcular la constante del paso previo del algoritmo, y que consiste en el resto de la siguiente división:

$$C = 2^{2k} \bmod N$$

Donde k es número de bits de las palabras y N es el módulo de entrada. Esta división se ha realizado hasta ahora con un módulo llamado resto basado en sucesivos desplazamientos, y con el fin de que disminuyera el área ocupada, también se ha utilizado un módulo de la librería de Synopsys denominado DW_div.

Se trata de una división que tiene un carácter especial, ya que uno de los términos es una potencia de base dos. Precisamente aprovechando esta circunstancia, existe un método que consigue optimizar dicha operación sin tener que realizar ninguna división, y que se va a exponer a continuación.

$C = 2^{2k} \bmod N$

Paso 1: Calcular m\ $2^m > N$

Paso 2: $q = 2k - m$; $r = 2^m$

Paso 3: Si $r > N$ entonces $r = r - N$
Si no $r = 2 \cdot r$; $q = q - 1$;

Paso 4: Si $q = 0$ entonces $C = r$;
Si no ir al Paso 3;

Se aprecia que se trata de un algoritmo muy simple que sólo contiene productos por dos (desplazamientos) y restas, con lo que se ha suprimido la dificultad de la división con operaciones muy sencillas.

Ahora vamos a mostrar los diferentes resultados conseguidos con la incorporación de este nuevo módulo, tanto a nivel funcional, donde se comprobará que el número de ciclos no aumenta en gran medida ya que se trata de un paso previo que se desarrolla en pocos ciclos, como a nivel de sintetización, donde se comprobará si se mejoran las condiciones de frecuencia y área ocupada conseguidas con el módulo anterior.

Primeramente se mostrarán los resultados del tiempo total invertido (nº de ciclos) para diferentes listas de entrada (doscientas para cada valor del parámetro N), donde observamos el máximo y mínimo número de ciclos de reloj, así como la media para cada valor de dicho parámetro:

N	(Max - Min) Ciclos	Media
8	78 - 71	75
16	136 - 100	118
32	256 - 204	230
64	496 - 420	458
128	950 - 894	922
256	1862 - 1714	1788
512	3624 - 3456	3540

Tabla 4.8: Número de ciclos con bloque calcula_c.

Como ya se había comentado, se observa que el número de ciclos no ha aumentado en gran medida, la razón de crecimiento es lineal al parámetro N, por lo que hace que el sistema sea algo más lento, pero sin llegar al extremo del sistema mostrado al principio de este documento, donde se obtuvieron unos resultados de tiempos muy grandes.

Ahora se mostrarán los resultados conseguidos mediante la sintetización de este bloque, donde apreciaremos si se producen optimizaciones respecto al programa anterior:

		ÁREA (μm^2)							
N	$T_{periodo}$	Combinac	Secuenc	Interconex	TOTAL	Puertos	Celdas	Biestab	SLACK
8	50	224315	43461	51839	319616	36	2386	107	- 0.97
8	60	138647	43334	33879	215861	36	1297	107	0.18
16	150	426717	82191	88246	597155	68	3276	203	21.26
16	130	459076	81881	99392	640351	68	3801	203	0.02
32	300	1523467	158103	288694	1970265	132	10691	395	52.35
32	250	1830992	157866	253181	2342031	132	14319	395	0.2

Tabla 4.9: Sintetización del sistema con módulo calcula_c.

Podemos sacar de la tabla varias consecuencias del uso del nuevo bloque:

- El $T_{periodo}$ necesario para que funcione el sistema también ha variado, aunque ahora obtenemos una mejora de frecuencia para los valores del parámetro N probados, debido a que dicho tiempo de periodo se ha reducido, con lo que la frecuencia del sistema será algo mayor.
- El área necesaria para la implementación también ha decrementado, al igual que el número de celdas, no siendo así con el número de biestables.

Con estos datos podemos decir que se ha producido una mejoría en las características de nuestro sistema, tanto en frecuencia como en área, ya que la inclusión de este nuevo módulo ha reducido en parte los retrasos producidos por la lógica combinacional.

El reparto en porcentaje del área por cada bloque de nuestro código sería el siguiente:

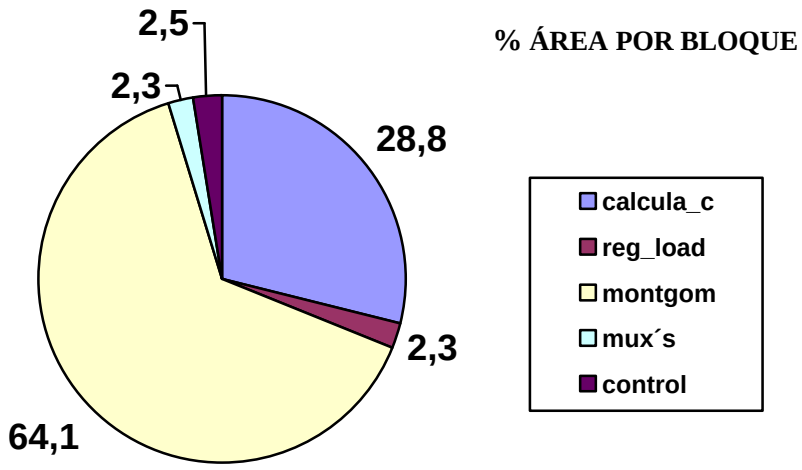

Figura 18: Porcentaje de área por cada bloque.

Como se aprecia en el gráfico anterior se ha reducido el peso específico del bloque que calcula la constante previa en el programa, siendo ahora el bloque de Montgomery el que más afecta con diferencia al área ocupada por nuestro código.

Esto corrobora nuestra exposición anterior, ya que el cambio del bloque divisor ha sido el responsable de la disminución del área y de la mejora de la frecuencia de funcionamiento.

El siguiente apartado será contrastar todos los resultados obtenidos a lo largo de las diferentes modificaciones de nuestro sistema, con el fin de obtener los resultados óptimos para nuestro diseño.

4.3. COMPARACIÓN ENTRE RESULTADOS.

A lo largo de todo este capítulo se han ido exponiendo los diferentes resultados conseguidos con cada configuración del sistema y del código utilizado, en este punto se va a hacer un resumen comparativo entre todos estos resultados, para valorar cada uno de los caminos que se han tomado.

Los resultados obtenidos han girado entorno a varios conceptos fundamentales en nuestro diseño, y que mostramos a continuación:

- Tiempo total de la obtención del resultado final: se refiere al número de ciclos necesarios desde que se habilita nuestro sistema hasta que se obtiene el dato de salida, y la consiguiente señal de finalización.
- Área necesitada para el diseño: se trata del área que ocupa la sintetización de nuestro código, y su consiguiente división en los diferentes tipos, es decir, el área consumida por la parte combinacional, el área necesitada por la parte secuencial y el área de interconexión.
- Frecuencia de funcionamiento: está relacionada con el tamaño del periodo del reloj, de forma que los retrasos producidos por la lógica no impidan un correcto funcionamiento del sistema.

A lo largo de este documento podemos observar que se ha intentado mejorar dichas características, sin embargo, se ha comprobado que la mejora de una de ellas normalmente lleva asociada alguna desventaja en nuestro sistema. Por ello, se contrastarán todos los resultados con el fin de equilibrar dichos aspectos positivos antes mencionados.

Para comenzar, podemos recordar las diferentes actualizaciones realizadas a nuestro sistema, con el fin de poder ver mejor las características de cada una de ellas. Cabe destacar una división total de dos etapas, una en la que usábamos un bloque que utilizaba un algoritmo de Montgomery lento, donde se necesitaban muchos ciclos de reloj, y otra en la que dicho algoritmo fue sustituido por otro que precisaba muchos menos ciclos para realizar el cómputo global de la exponenciación. Partiendo de esto, y si tenemos en cuenta que también se han realizado modificaciones al bloque que

calculaba el resto inicial del paso previo del algoritmo, podemos enumerar las diferentes modificaciones de nuestro código en:

- Actualización inicial o primera actualización: utiliza el bloque de Montgomery lento, el que precisa muchos ciclos, pero que es muy simple y utiliza bastante poca área, y un módulo que calcula el resto utilizando desplazamientos y sumas.

- Segunda actualización: se sustituyó el bloque que calcula el resto por un módulo de la librería de Synopsys, encargado de realizar divisiones de forma rápida y simple. El bloque de Montgomery sigue siendo el mismo que en la primera actualización.

- Tercera actualización: en este caso lo que se cambia es dicho bloque de Montgomery por otro que aplique el mismo algoritmo pero que tarde un tiempo menor. El nuevo bloque de Montgomery tendrá la misma funcionalidad que el anterior.

- Cuarta actualización: se sustituyó el bloque de librería matemática de Synopsys por otro bloque que utiliza un algoritmo especialmente diseñado para divisiones en las qué un operando es potencia de base dos.

Ahora se mostrarán los resultados obtenidos para cada una de estas actualizaciones de nuestro programa, con el fin de poder comparar con mayor claridad qué aspectos de los antes descritos se comportan mejor en cada actualización.

Se va a dividir esta comparación de resultados en dos partes, una que se encarga de la parte funcional y que está relacionada con la compilación de nuestro código Verilog, y otra parte relacionada con la sintetización del mismo, que muestra los resultados obtenidos al aplicar el programa sintetizador.

4.3.1. Resultados de compilación.

Estos resultados están basados en el correcto funcionamiento del código Verilog elaborado, así pues, partiendo de una serie de resultados obtenidos con otro programa, en nuestro caso Matlab, hay que comprobar que aplicando las mismas entradas se obtienen las mismas salidas. Esto se realizó mediante un módulo en Verilog llamado **test_rsa**, que también se adjunta en el anexo, y simplemente consiste en un bucle que va recorriendo una tabla de datos, que son las entradas y salidas del sistema, con el fin de comprobar que todas las salidas conseguidas con nuestro código Verilog son idénticas a las obtenidas con el programa en Matlab.

En el anexo también se mostrará la forma de conseguir dichas entradas al sistema, ya que se recuerda que deben cumplir una serie de especificaciones las entradas a dicho sistema, para que el sistema funcione correctamente.

El otro punto importante de este apartado es el relacionado con el tiempo total de obtención del resultado final, es decir, mide un poco la velocidad del programa, el número de ciclos de reloj necesarios para conseguir los datos de salida. Anteriormente se comprobó que se necesitaba que el algoritmo no fuese muy lento. Ahora mostraremos el comportamiento de este aspecto en cada actualización de nuestro código mediante el siguiente gráfico:

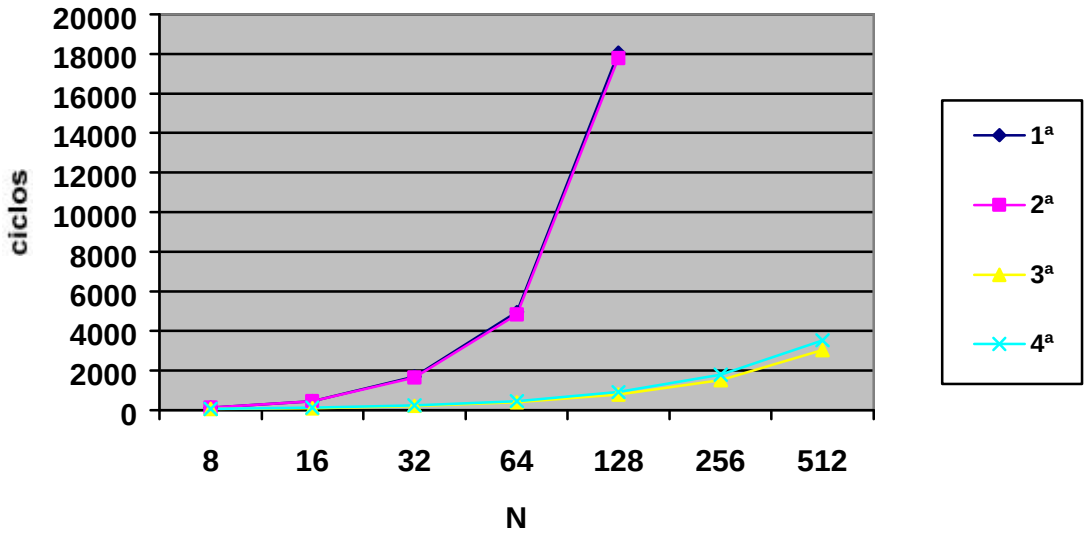

Figura 19: Comparación número de ciclos.

En el gráfico anterior se aprecia una división total entre dos grupos de actualizaciones, la primera y segunda actualización por un lado, casi con los mismos resultados, y la tercera y la cuarta por el otro. Esta división hace referencia a la utilización en un caso del bloque de Montgomery más lento, que realizaba una iteración en cada ciclo, de forma que para valores de N elevados aumentaba en gran tamaño, y la utilización del otro módulo realizado de Montgomery, que perdía su simplicidad, pero aumentaba su velocidad en gran medida, como se aprecia en el gráfico.

Se observa que la diferencia es considerable entre ambos grupos de actualizaciones, haciéndose aún más grande para valores altos del parámetro N.

Tras esta primera comparación queda claro que si lo que buscamos es rapidez del algoritmo debemos utilizar la tercera o la cuarta actualización, aunque estaremos sacrificando la simplicidad de los otros casos.

A continuación se mostrarán los cronogramas de estos dos grupos bien diferenciados de actualizaciones, tomando datos de tamaño N=16 bits, donde se aprecia la diferencia de ciclos necesarios en ambos casos. Además se puede observar la evolución de algunas señales, para poder conocer mejor su comportamiento.

En el primer cronograma, que se corresponde con las actualizaciones primera y segunda, vemos que el tiempo requerido es mayor (~ 11 μs), y tenemos un contador que aumenta cada ciclo de reloj. En el segundo cronograma, que se corresponde con la tercera y cuarta actualización, el tiempo final se reduce (~ 3.2 μs), y se puede apreciar la evolución de las fases de la máquina de estados, donde ahora el contador va incrementandose con los bits del exponente.

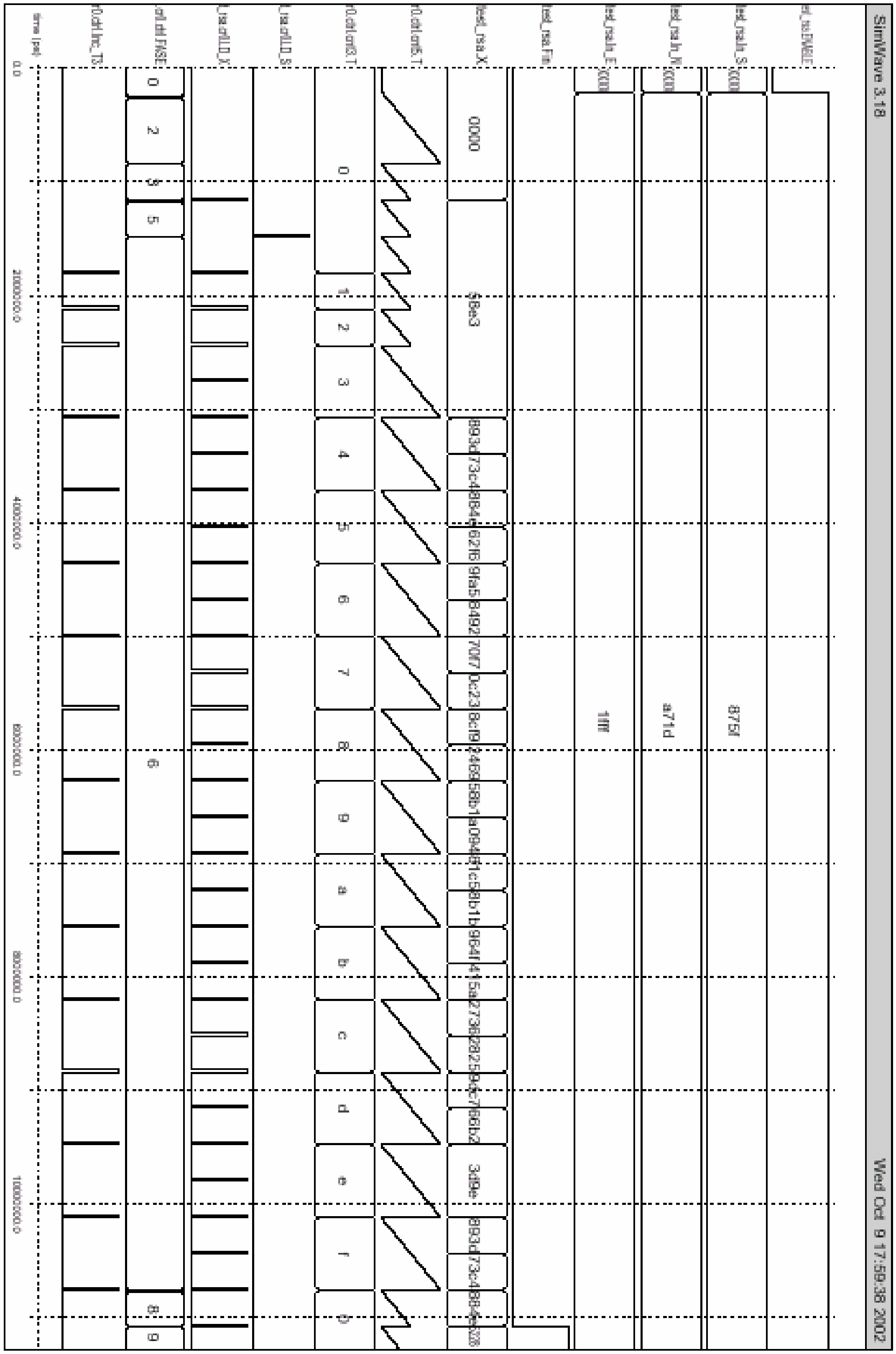

Figura 20: Cronograma de actualizaciones primera y segunda.

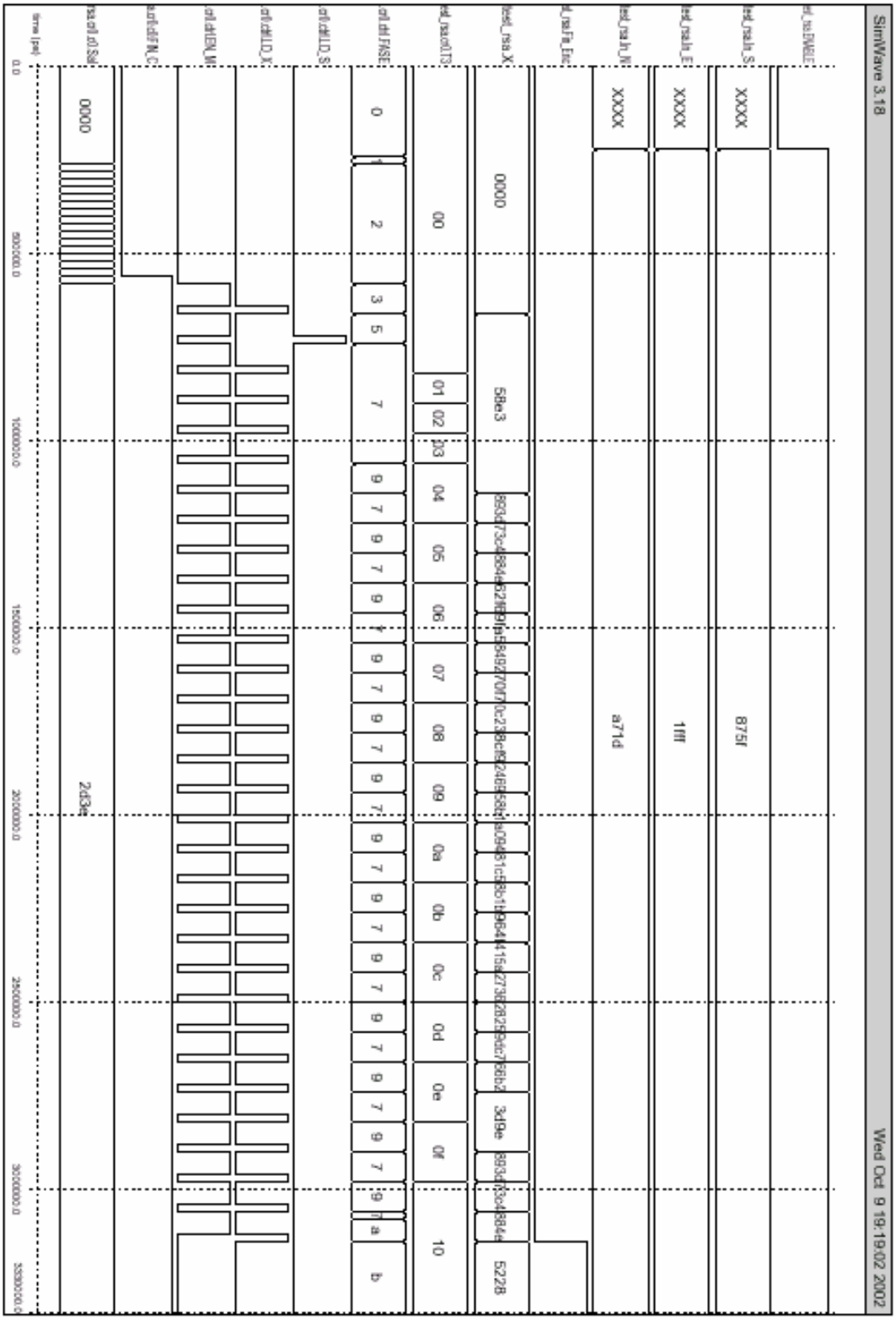

Figura 21: Cronograma de las actualizaciones tercera y cuarta.

4.3.2. Resultados de Sintetización.

El otro aspecto importante de comparación de resultados es el basado en la sintetización del código Verilog realizado, por lo que en este punto se analizarán los diferentes resultados conseguidos de área, frecuencia y demás aspectos procedentes de la síntesis de nuestro sistema.

Hasta ahora hemos comprobado los diferentes resultados obtenidos para los diferentes valores del parámetro N, que se definía como el ancho de los datos del sistema, en este punto se va a realizar una comparación de las características obtenidas por las distintas actualizaciones de nuestro código para los diferentes valores de N, es decir, vamos a comprobar que casos son favorables para cada valor del parámetro.

Primero tomaremos el valor N=8, es decir, datos con un tamaño de ocho bits, y contrastaremos los resultados de la síntesis. Nos vamos a fijar en el área que se necesita, la frecuencia de funcionamiento, y otros datos como el número de biestables y de celdas. Así, se consigue la siguiente tabla:

Actualización	$T_{periodo}$ (ns)	Área (mm^2)	Celdas	Biestab	SLACK	Frecuencia (MHz)
1ª	20	0.015	1001	84	2.14	50
2ª	20	0.028	181	36	12.76	50
3ª	60	0.348	2721	80	0.21	16.6
4ª	60	0.215	1297	107	0.18	16.6

Tabla 4.10: Comparación de síntesis para N=8.

Se observa que para la tercera y cuarta actualización se aumentó el periodo de reloj debido a la utilización del nuevo algoritmo que realiza el producto de Montgomery, al igual que aumentó el área ocupada y el número de celdas, por el mismo motivo.

Además vemos que si comparamos las dos últimas actualizaciones, en la cuarta, que es donde incluimos el bloque que calcula el resto utilizando una potencia base dos, el área ocupada disminuye, no siendo así el número de biestables. Esto muestra que el

nuevo bloque incluido reduce el área pero precisa de un número mayor de biestables para calcular el resto inicial.

Ahora contrastamos los valores para el parámetro N=16:

Actualización	$T_{periodo}$ (ns)	Área (mm^2)	Celdas	Biestab	SLACK	Frecuencia (MHz)
1ª	40	0.225	1139	158	3.72	25
2ª	20	0.051	320	68	12.73	50
3ª	150	1.20	10534	152	0.11	6.6
4ª	130	0.64	3801	203	0.12	7.7

Tabla 4.11: Comparación de síntesis para N=16.

Se aprecia que es la segunda actualización la que muestra unos valores de síntesis mejores (módulo de montgomery 'lento' y módulo de librería matemática), ya que la frecuencia de funcionamiento es más alta y los valores de áreas son bastante menores. Después podríamos elegir la cuarta actualización (módulo calcula_c), ya que presenta valores de área y frecuencia mejores que la otra que tarda el mismo número de ciclos en obtener el resultado final (tercera actualización).

Los valores de síntesis para N=32 serían los siguientes:

Actualización	$T_{periodo}$ (ns)	Área (mm^2)	Celdas	Biestab	SLACK	Frecuencia (MHz)
1ª	50	0.676	4467	304	0.61	20
2ª	20	0.094	573	132	12.25	50
3ª	300	2.96	21435	286	1.87	3.3
4ª	250	2.34	14319	395	0.2	4

Tabla 4.12: Comparación de síntesis para N=32.

Ahora se comprueba que el área aumenta en gran medida para las actualizaciones que contienen el módulo 'rápido' de Montgomery, es decir, el precio por tener un sistema RSA que obtenga antes los datos de salida, será un incremento considerable en

su área y una disminución de frecuencia de funcionamiento. No obstante, sigue cumpliéndose que ante las dos posibilidades de configuración que contienen este módulo (tercera y cuarta actualización), la última tiene mejores características de frecuencia y de área ocupada.

Para valores de N=64 no se obtuvieron resultados de las dos últimas actualizaciones debido a que se preveían unos resultados poco favorables de área y frecuencia, y además el tiempo de sintetización era inpensable. No obstante, para las primeras actualizaciones sí se consiguieron dichos resultados, que se muestran a continuación:

Actualización	$T_{periodo}$ (ns)	Área (mm^2)	Celdas	Biestab	SLACK	Frecuencia (MHz)
1ª	100	0.82	4289	560	9.43	10
2ª	50	0.18	1146	270	21.69	20

Tabla 4.13: Comparación de síntesis para N=64.

A la vista de estos valores podemos decir que cómo se preveía la utilización del módulo de la librería matemática de Synopsys alcanza unos resultados más favorables al sintetizar el programa, principalmente en lo que al área ocupada se refiere.

Una vez mostrados todos estos resultados para cada valor del parámetro N, se aprecia que la segunda configuración es la que tiene unos valores óptimos en cuanto a área y frecuencia se refiere, pero si descartamos las dos primeras debido a que vimos que el tiempo total en obtención del resultado era excesivamente elevado para valores grandes de N, nos encontramos con que la configuración que mejores valores obtiene será la última mostrada, es decir, la que constaba del algoritmo que mejoraba la división cuando aparecen potencias de dos (módulo calcula_c), ya que consigue rebajar algo el área ocupada y la frecuencia de funcionamiento, aunque necesite un número de biestables mayor.

4.4. ESPECIFICACIONES ACTUALES DE ALGUNOS FABRICANTES.

Actualmente, los grandes fabricantes de este tipo de criptosistemas (cryptocores), están utilizando tecnologías pipeline y diseños High Perfomace Hardware Acelerators para conseguir acelerar los procesos de exponenciación modular sin disparar sus números de puertas. A continuación se mostrarán las especificaciones de diseño de algunos de estos fabricantes:

- MIPS Technologies: Utiliza estructura pipeline trabajando en paralelo con datos de un ancho de 16-32 bits. Tiene una frecuencia de funcionamiento de 0-100 MHz y utiliza áreas de 1.5 mm^2 cuando trabaja con datos de 1024 bits. El tiempo estimado de funcionamiento es menor a los 100 ms, y tiene un consumo de 0.5 mW/MHz.

- XILINX XC4000E: Dispone de un core que realiza la exponenciación modular utilizando una implementación pipeline con 16 bits de ancho de datos. Utiliza lógica carry/save y consigue unos resultados de áreas de 8500 puertas a 8 MHz y 10000 puertas a 15.6 MHz para datos de 512 bits.

- IBM: High Perfomance RSA Hardware Acelerator design. Obtiene valores de área de 5 mm^2 para datos de 1024 bits de ancho. Sus principales características son el uso de operaciones paralelas internas y sumadores con High Speed Carry, el core de la exponenciación modular ocupa 4.9 mm^2 de área y su frecuencia de funcionamiento es menor de 48 MHz.

- IAIK Technologies: Dispone de un prototipo de cryptocore RSA que utiliza datos de 1024 bits trabajando con estructuras paralelas de 16 bits. Necesita un tiempo de 250 ciclos de reloj y ocupa 70 mm^2 de área de silicio. Su diseño está formado por un acumulador y seis carry save adders.

- SAFENET Technologies: Sustituye el nivel de puertas de la sintetización de Verilog por un diseño EIP-23 Hardware de Public Key Acelerator, consiguiendo

valores de frecuencia de 50-200 MHz y 34000 puertas lógicas para valores de ancho de datos de 1024 bits.

- SCI Worx: Utiliza tecnología de operaciones internas en paralelo, trabajando con estructuras de 8, 16, 32, 64 y 128 bits. Consigue los siguientes resultados de área y frecuencia para datos de 1024 bits:

Nº bits estruct.	Área (nº puertas)	Frec. Max. (MHz)	Tiempo total (ms)
8	6500	400	110-330
16	8500	350	29-75
32	15500	250	18-20
64	78000	200	2-4
128	270000	140	2

Tabla 4.14: Comparación de síntesis para N=64.

5. CONCLUSIÓN Y LINEAS FUTURAS.

En este capítulo, después de haber explicado detalladamente el funcionamiento de nuestro sistema, y de haber obtenido todos los resultados, llega el momento de obtener algunas conclusiones generales del proyecto que se ha realizado.

En base a los resultados obtenidos, tanto de la compilación como de la síntesis de nuestro programa, se comprobó anteriormente que según los requisitos que necesitemos que se cumplan, es mejor elegir una u otra actualización del programa, es decir, si buscábamos áreas de menor tamaño y frecuencias de funcionamiento mayores, la opción elegida sería la que utiliza un módulo de Montgomery que realizaba un paso del algoritmo en cada ciclo de reloj, aunque esto hiciera que el tiempo total aumentara en gran medida debido a que se precisaban muchos ciclos de reloj.

La otra opción es la de usar una actualización de dicho bloque que realizaba el producto de Montgomery de forma más rápida, consiguiendo un tiempo total bastante menor, pero que empeora las especificaciones de área y frecuencia para valores altos de ancho de los datos.

Los sistemas RSA se caracterizan por tener una baja velocidad tanto para encriptar como para decriptar grandes volúmenes de datos, si lo comparamos con otros algoritmos simétricos, como por ejemplo el DES, obtenemos una relación de 10.000 : 1 en su implementación hardware. Los resultados obtenidos se comportan de forma aceptable para anchos de datos de hasta 64 bits, pero si trabajamos con tamaños mayores las especificaciones que se obtienen se alejan bastante de las ofrecidas por algunos fabricantes en la actualidad.

Como conclusión cabría destacar que esa segunda opción, aunque ha conseguido sus propósitos de rebajar el tiempo total, ha dado unos resultados tras su sintetización que no invitan a seguir trabajando en siguientes diseños para valores de tamaños de datos mayores a los mostrados, ya que las especificaciones de área y de frecuencia empeoraban en gran medida. Es por ello, que parece que el camino a desarrollar sería uno asociado al primer modelo expuesto, con un bloque del producto de Montgomery

más simple, pero mejorándolo de forma que se obtuvieran los datos de salida en un tiempo menor.

Es por esto que se propone como una línea de trabajo futura a la realización de este proyecto, una modificación distinta a la realizada del bloque de Montgomery inicial, de forma que se mejoren los resultados de tiempos para tamaños de datos superiores a 64bits.

Si consideramos el esquema que se expuso para dicho bloque (figura 4), se aprecia que el camino crítico que ralentiza el tiempo total del sistema y provoca más retrasos es el formado por los bloques SUM_1, SUM_2, DESPLAZA Y RESTAR_N, ya que tiene que pasar por ahí para cada bit de la entrada al producto de Montgomery. Una posible solución podría ser la de usar sumadores del tipo Carry Save. Una vez calculado el producto de Montgomery en función de los vectores Carry y Sum, éstos serían incluidos en una estructura pipeline, de forma que al operar en paralelo se reduzca el tiempo total del sistema y aumente su frecuencia.

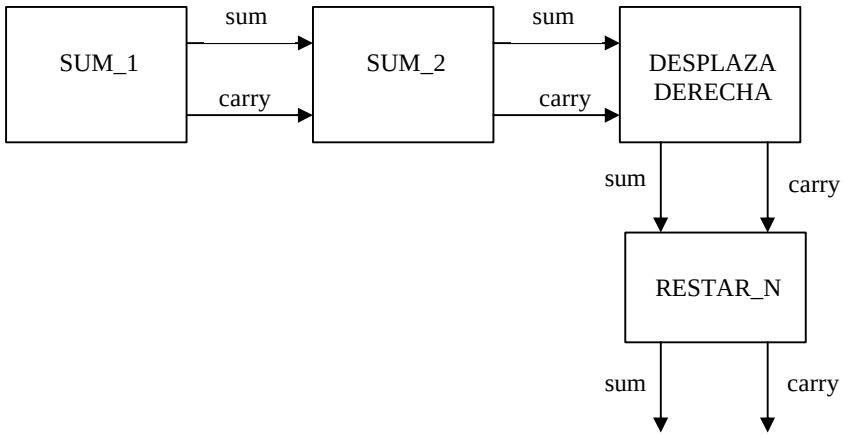

Figura 22: Estructura bloque Montgomery usando **carry sum**.

6. ANEXO.

En este capítulo se adjuntará todo el código fuente procedente de la elaboración de nuestro sistema, tanto el programa en Verilog, como su versión en Matlab que se ha utilizado para contrastar todos los resultados. Además se mostrará la forma de obtener los datos de entrada de forma que cumplan las especificaciones necesarias en todo sistema RSA. Esta última parte no forma parte de nuestro sistema, ya que sólo se encarga de la encriptación y desencriptación de datos mediante la exponenciación modular, sin embargo, es necesario para probar que el sistema funciona correctamente, por lo que se elaboró una forma de obtener las entradas en código Matlab.

Por tanto, dividiremos este capítulo en dos apartados, uno que contiene todo el código del programa en Verilog, y otro asociado al programa Matlab que genera los datos de entrada y la correspondiente lista de salida, para contrastar la correcta funcionalidad del código Verilog.

6.1. CÓDIGO EN LENGUAJE VERILOG.

El código utilizado por todos los bloques que intervienen en nuestro sistema es el siguiente:

codec_rsa.v: es el módulo más alto de nuestro sistema, contiene todos los bloques necesitados, y sólo es instanciado desde el módulo de test que sirve para probar el correcto funcionamiento de éste.

```
`include "control.v"
`include "mux_rsa_2_1.v"
`include "mux_rsa_3_1_A.v"
`include "mux_rsa_3_1_B.v"
`include "reg_carga_s.v"
`include "M_mont.v"
`include "reg_carga_x.v"
`include "Calcula_C.v"

module codec_rsa (CLK, RSTz,
                  In_N, In_E, In_S, Nuevo_X,
                  Fin_Enc, ENABLE);

parameter N=16;parameter Nc=5;     // Realiza el calculo del algoritmo RSA
input CLK,RSTz,ENABLE;             // para valores de N bits de entrada.
input [N-1:0] In_E;                // Las entradas son el mensaje a codificar
input [N-1:0] In_N;                // S, las claves E y N, y una constante
input [N-1:0] In_S;                // que depende del tamaño que utilicemos.
output [N-1:0] Nuevo_X;
output Fin_Enc;

wire [N-1:0] In_C;                 // Algoritmo RSA:   X=(S exp E) mod N
wire [N-1:0] Un_uno;               //  X=montgomery (1,C,N)
wire [N-1:0] OP_A;                 //  S=montgomery (S,C,N)
wire [N-1:0] OP_B;                 //  Desde i=k-1 hasta 0
wire [N-1:0] De_Mont;              //     X=montgomery (X,X,N)
wire [N-1:0] Nuevo_S;              //     Si exponente(i)=1
wire [N-1:0] S_load;               //        X=montgomery (S,X,N)
//wire [Nc-1:0] T5;                //  X=montgomery (X,1,N)
wire [Nc-1:0] T3;                  //  Devuelve X
wire [Nc-2:0] Selec_bit_e;
wire [1:0] SM_A;
wire [1:0] SM_B;
wire LD_S;
wire LOAD_S;
wire LD_X;
wire ei;
```

```
wire [2*N:0] Const;
wire TC;
wire [3:0] FASE;
reg ready;
wire FIN;
wire EN_M;
wire FIN_C;

assign Un_uno = 1;
assign LOAD_S = (FASE==2) | (LD_S);
assign Selec_bit_e = T3[Nc-2:0];
assign Const=1<<(2*N);


// Register Input
// --------------------------------

reg [N-1:0] DIn_N, nDIn_N, DIn_E, nDIn_E, DIn_S, nDIn_S;

always @(posedge CLK or negedge RSTz)
if(RSTz==0) begin DIn_N<=0;DIn_E<=0;DIn_S<=0;end
else begin DIn_N<=nDIn_N;DIn_E<=nDIn_E;DIn_S<=nDIn_S;end

always @(ENABLE or In_N or DIn_N or In_E or DIn_E or In_S or DIn_S)
if (ENABLE==1) begin nDIn_N<=In_N;nDIn_E<=In_E;nDIn_S<=In_S;end
else begin nDIn_N<=0;nDIn_E<=0;nDIn_S<=0;end

always @(DIn_N or DIn_E or DIn_S)
if ((DIn_N>0 && DIn_E>0 && DIn_S>0)) ready=1;
else ready=0;

assign ei = DIn_E [N-1-Selec_bit_e];

// --------------------------------
// Instantiate modules

Calcula_C #(N) c0 (
  .CLK(CLK), .RSTz(RSTz),
  .EN(ENABLE), .In_N(In_N),
  .FIN_C(FIN_C), .Sal(In_C)
);

control #(N,Nc) ctrl (
   .CLK(CLK), .RSTz(RSTz),
   .ENABLE(ENABLE), .ready(ready), .FIN(FIN), .FIN_C(FIN_C),
   .T3(T3),
   .ei(ei),
   .LD_S(LD_S), .LD_X(LD_X), .EN_M(EN_M),
   .Select_Mux_A(SM_A), .Select_Mux_B(SM_B),
   .Fin_dec(Fin_Enc), .FASE(FASE)
```

```
);

reg_carga_c regc (.CLK(CLK), .In(C), .Load(LD_C), .Out(In_C));

mux_rsa_3_1_A #(N) muxa (
  .In1(Nuevo_X), .In2(Nuevo_S), .In3(Un_uno),
  .Select_In(SM_A),
  .Out(OP_A)
);

mux_rsa_3_1_B #(N) muxb (
  .In1(In_C), .In2(Nuevo_X), .In3(Un_uno),
  .Select_In(SM_B),
  .Out(OP_B)
);

M_mont #(N) m0 (
  .CLK(CLK), .RSTz(RSTz),
  .EN(EN_M), .A(OP_A), .B(OP_B), .In_N(DIn_N),
  .FIN(FIN), .Sal(De_Mont)
);


mux_rsa_2_1 #(N) mux (
   .In1(De_Mont), .In2(DIn_S),
   .Select_In(FASE==2),
   .Out(S_load)
);

reg_carga_s #(N) regs (
   .CLK(CLK), .RSTz(RSTz),
   .In(S_load), .Load(LOAD_S),
   .Out(Nuevo_S)
);

reg_carga_x #(N) regx (
   .CLK(CLK), .RSTz(RSTz),
   .In(De_Mont), .Load(LD_X),
   .Out(Nuevo_X)
);

endmodule

**********************************************************************
**********************************************************************
```

montgomery.v: es el módulo que calcula los productos de Montgomery necesitados en el desarrollo de la exponenciación modular. Como se ha explicado a lo

largo de este documento se desarrollaron dos versiones de este bloque, la primera se denominó montgomery.v y conseguía un número de ciclos muy elevado, y la otra versión es el módulo M_mont.v, con el que se redujo bastante el número de ciclos totales para la obtención del resultado final.

```
module M_mont (
  CLK, RSTz,
  EN, A, B, In_N,
  FIN, Sal
);

parameter N=16;
input  CLK, RSTz;
input  EN;  input [N-1:0] A, B, In_N;
output FIN; output [N-1:0] Sal;

// Internal registers
reg [N-1:0] R_A, nR_A;
reg [N-1:0] R_B, nR_B;
reg [N-1:0] R_N, nR_N;


// State Machine
reg [1:0] SM, nSM;

always @(posedge CLK or negedge RSTz)
if(RSTz==0) SM<=0; else SM<=nSM;

always @(EN or SM) begin
  nSM<=SM;
  case(SM)
  0: if(EN==1) nSM<=1;
  1: nSM<=3;
  3: nSM<=2;
  2: if(EN==0) nSM<=0;
  endcase
end

// FIN
assign FIN=(SM==2);

// Register while on SM=0
always @(posedge CLK or negedge RSTz)
if(RSTz==0) begin R_A<=0; R_B<=0; R_N<=0; end
else begin R_A<=nR_A; R_B<=nR_B; R_N<=nR_N; end

always @(SM or R_A or A) if(SM==0) nR_A<=A; else nR_A<=R_A;
always @(SM or R_B or B) if(SM==0) nR_B<=B; else nR_B<=R_B;
```

```
always @(SM or R_N or In_N) if(SM==0) nR_N<=In_N; else nR_N<=R_N;


// Process
reg [N+1:0] SUM_AC, nSUM_AC;
assign Sal=SUM_AC[N-1:0];
integer i;
wire SUM_AC_gt = (SUM_AC > R_N);
wire [N:0] SUM_AC_diff = SUM_AC-R_N;


always @(posedge CLK or negedge RSTz)
if(RSTz==0) SUM_AC=0; else SUM_AC=nSUM_AC;

always @(SM or R_A or R_B or R_N or SUM_AC or SUM_AC_gt or
            SUM_AC_diff) begin
  nSUM_AC=SUM_AC;
  case(SM)
  0: nSUM_AC=0;
  1: for(i=0;i<N;i=i+1) begin
       if(R_A[i]!=0) nSUM_AC=nSUM_AC+R_B;
    if(nSUM_AC[0]==1) nSUM_AC=nSUM_AC+R_N;
    nSUM_AC=nSUM_AC>>1;
    end
  3: if(SUM_AC_gt) nSUM_AC=SUM_AC_diff;
  endcase
end

endmodule


**********************************************************************


`include "reg_u.v"
`include "impar_par.v"
`include "restar_N.v"

module montgomery (CLK, RSTz, Bus_A,
                   N_bus, Bus_B, Select_bit,
                   Mon_Result);

parameter N=16;parameter Nc=5;    //Funcion que aplica el producto de
input CLK,RSTz;                   //Montgomery a dos entradas A y B de 16
input [N-1:0] Bus_A, Bus_B;       //bits y con un modulo N del mismo tamaño.
input [N-1:0] N_bus;              //
input [Nc-2:0] Select_bit;
                                  // u=0;
output [N-1:0] Mon_Result;        // Desde i=0 hasta N-1
wire[N-1:0] Mon_Result;           //    u=Ai * B
                                  //    Si u es impar  u=u+N
```

```
wire [N-1:0] BUS_1;                 //     u=u/2
wire[N:0] BUS_2;                    // Si u mayor que N   u=u-N
wire [N+1:0] BUS_3;                 // Devuelve u
wire [N+1:0] BUS_4;
wire [N+1:0] BUS_5;
wire [N:0] BUS_6;
wire [N:0] BUS_7;
wire [N:0] BUS_8;
wire Ai;
wire Bo;
wire Uo;
wire Impar;
wire Selec_mux;

assign Bo = Bus_B [0];
assign Uo = BUS_2 [0];
assign Selec_mux = BUS_7 [N];
assign BUS_3 = ({2'b00, BUS_1}) + ({1'b0, BUS_2});
assign BUS_4 = ({2'b00, N_bus}) + BUS_3;
assign Mon_Result = BUS_8 [N-1:0];
assign Ai = Bus_A [Select_bit];
assign BUS_1 = (Ai==0) ? 0 : Bus_B;
assign BUS_5 = (Impar == 1'b0) ? BUS_3 : BUS_4;
assign BUS_6 = BUS_5[N+1:1];
assign BUS_8 = (Selec_mux == 1'b0) ? BUS_7 : BUS_6;

//   Instantiate modules
//  --------------------------

reg_u #(N,Nc) regu (.CLK(CLK), .RSTz(RSTz), .In_dato(BUS_6),
                                .Selec_bit(Select_bit), .Out_dato(BUS_2));

impar_par ip (.Ai(Ai), .Bo(Bo), .Uo(Uo), .Impar(Impar));

restar_N #(N) resta (.Operando_1(BUS_6), .Operando_2(N_bus), .Sal(BUS_7));

endmodule

*********************************************************************
*********************************************************************
```

control.v: es el bloque que lleva el control de todas las señales que interacionan en el sistema, para que todo funcione correctamente. Contiene una máquina de estados que gobierna todas las fases de nuestro programa.

```
`include "contador_T3.v"

module control (
   CLK, RSTz,
   ENABLE,
   FIN, FIN_C,
   ei,
   ready, T3,
   EN_M,
   LD_S, LD_X,
   Select_Mux_A, Select_Mux_B,
   Fin_dec,FASE
);

parameter N=16;parameter Nc=5;

input ENABLE,CLK,RSTz;
input [Nc-1:0] T3;
input ei;
input ready;
input FIN,FIN_C;
output EN_M;
output LD_S;
output LD_X;
output [1:0] Select_Mux_A;
output [1:0] Select_Mux_B;
output Fin_dec;
output [3:0] FASE;

// Internal variables
// ---------------------------------
reg Inc_T3;
reg LD_S;
reg LD_X;
reg RESET_T3;
reg EN_M;
reg [1:0] Select_Mux_A;
reg [1:0] Select_Mux_B;
reg [3:0] FASE,nFASE;

// Counter instantiation
// ---------------------------------
contador_T3 #(Nc) cnt3 (
   .CLK(CLK), .RSTz(RSTz),
   .Inc(Inc_T3),
   .ENABLE(ENABLE), .RST_T3(RESET_T3),
   .T(T3)
);
```

```
// Internals

// FASE evolution
// --------------------------------
always @(posedge CLK or negedge RSTz) begin
  if (RSTz==0) FASE<=0; else FASE<=nFASE;
end

always @(ENABLE or FASE or T3 or FIN or ei or ready or FIN_C)
   if(ENABLE==0) nFASE<=0;
   else begin
      nFASE<=FASE;
      case(FASE)
      0: if(ENABLE==1) nFASE<=1;
      1: if(ready==1) nFASE<=2;
      2: if (FIN_C) nFASE<=3;
      3: if(FIN) nFASE<=5;
      5: if(FIN) nFASE<=7;
      7: if(T3[Nc-1]==1) nFASE<=10;
        else begin if(FIN && ei) nFASE<=9;
                 else if (FIN && ~ei) nFASE<=7;end
      9: if(FIN) nFASE<=7;
      10: if(FIN) nFASE<=11;
      11: nFASE<=11;
      default: nFASE<=0;
      endcase
   end

// Fin_dec
// --------------------------------
assign Fin_dec=(FASE==11);

// RESET_T3
// --------------------------------
always @(FASE or FIN) begin
  RESET_T3<=0;
  case(FASE)
  5: if (FIN) RESET_T3<=1;
  endcase
end

// EN_M
// --------------------------------
always @(FASE or FIN) begin
  EN_M<=0;
  case(FASE)
  3,5,7,9,10: if (~FIN) EN_M<=1;
  endcase
end
```

```
// Inc_T3
// --------------------------------
always @(FASE or FIN) begin
  Inc_T3<=0;
  case(FASE)
  7: if (FIN) Inc_T3<=1;
  endcase
end

// LD_X
// --------------------------------
always @(FASE or FIN) begin
   LD_X<=0;
   case(FASE)
   3,7,9,10: if (FIN) LD_X<=1;
   endcase
end

// LD_S
// --------------------------------
always @(FASE or FIN) begin
    LD_S<=0;
    case(FASE)
    5: if (FIN) LD_S<=1;
    endcase
end


always @(FASE) begin
    case (FASE)
    3: begin Select_Mux_A=2;Select_Mux_B=0; end
    5: begin Select_Mux_A=1;Select_Mux_B=0; end
    7: if (T3[Nc-1]) begin Select_Mux_A=0;Select_Mux_B=2; end
       else begin Select_Mux_A=0;Select_Mux_B=1; end
    9: begin Select_Mux_A=1;Select_Mux_B=1; end
    10: begin Select_Mux_A=0;Select_Mux_B=2; end
endcase
end

endmodule


**********************************************************************
**********************************************************************
```

resto.v: módulo que calcula el resto del paso inicial del algoritmo de exponenciacion modular de Montgomery:

```
module resto (P, Q, RSTz, CLK, Resto, Val);

parameter N=32;parameter Nc=6;

input [Nc-1:0] Val;
input [4*N-1:0] P, Q;
input CLK,RSTz;
output [4*N-1:0] Resto;

wire [Nc-1:0] Val;
reg [4*N-1:0] Prod,nProd;
reg [4*N-1:0] Resto,nResto;


always @(posedge CLK or negedge RSTz) begin
  if (!RSTz) begin
    Prod=0;
    Resto=0;
  end
  else begin
    Prod=nProd;
    Resto=nResto;
  end
end

always @(Val or P or Q or Prod) begin
 nProd=Prod;
 if (Val==2*N-2) begin
     nProd=0;//nResto=0;
     if ((Prod + (Q << Val)) < P+1)
       nProd = (Prod + (Q <<Val));
 end
 else begin
     if ((Prod + (Q << Val)) < P+1) begin
       nProd = (Prod + (Q << Val));
     end
 end
end

always @(P or nProd) nResto=P-nProd;

endmodule
```

****
****

Calcula_C.v: módulo que calcula también el resto del paso previo, pero de forma más rápida y eficiente, aprovechando que un término de la división es potencia de dos.

```
module Calcula_C (
  CLK, RSTz,
  EN, In_N,
  Sal, FIN_C
);

parameter N=16;
input  CLK, RSTz, EN;
input [N-1:0] In_N;
output [N-1:0] Sal;output FIN_C;

// Internal registers
reg [N-1:0] R_N, nR_N;

// State Machine
reg [1:0] SM, nSM;

always @(posedge CLK or negedge RSTz)
if(RSTz==0) SM<=0; else SM<=nSM;



// Register while on SM=0
always @(posedge CLK or negedge RSTz)
if(RSTz==0) begin R_N<=0; end
else begin R_N<=nR_N; end

always @(SM or R_N or In_N) if(SM==0) nR_N<=In_N; else nR_N<=R_N;

// Process
reg [N:0] r, q, nr, nq;

assign Sal=r[N-1:0];

integer i,listo;

wire r_gt = ((r<<1) < R_N);
wire [N:0] r_diff = (r<<1)-R_N;


always @(posedge CLK or negedge RSTz)
if(RSTz==0) begin r=0;q=0;end else begin r=nr;q=nq;end

always @(SM or R_N or r or q or r_gt or r_diff) begin
  //nr=r;nq=q;
  case(SM)
  0: begin nr=0;nq=0;end
  1: begin
      listo=0;
```

```
        for(i=0;i<N+1;i=i+1) begin
         if((1<<i)>R_N && listo==0) begin nr=(1<<i)-R_N;nq=2*N-i;listo=1;end
        end
       end
    3: if(r_gt) begin nr=(r<<1);nq=q-1;end
      else begin nr=r_diff;nq=q-1;end
   endcase
  end

  always @(EN or SM or q) begin
   if(EN==0) nSM<=0;
   else begin
    nSM<=SM;
    case(SM)
    0: if(EN==1) nSM<=1;
    1: nSM<=3;
    3: if (q==2) nSM<=2;
    2: nSM<=2;
    endcase
   end
  end

  // FIN_C
  assign FIN_C=(SM==2);

  endmodule
```

****
****

reg_carga: son registros de carga, que su única misión es generar una salida cuando se activa alguna señal.

```
  module reg_carga_s (CLK, RSTz, In, Load, Out);

  parameter N=16;
  input CLK,RSTz;
  input [N-1:0] In;
  input Load;
  output [N-1:0] Out;

  reg [N-1:0] Out,nOut;

  always @(posedge CLK or negedge RSTz) if (RSTz==0) Out=0; else Out=nOut;

  always @(Load or In or Out)
  begin
      if (Load==1) nOut=In;
```

```
        else nOut=Out;
    end

    endmodule
```

****
****

```
    module reg_carga_x (CLK, RSTz, In, Load, Out);

    parameter N=16;
    input CLK,RSTz;
    input [N-1:0] In;
    input Load;
    output [N-1:0] Out;

    reg [N-1:0] Out,nOut;

    always @(posedge CLK or negedge RSTz) if (RSTz==0) Out=0; else Out=nOut;

    always @(Load or In or Out)
    begin
        if (Load==1) nOut=In;
        else nOut=Out;
    end

    endmodule
```

****
****

contador.v: es un bloque que cuenta.

```
    module contador_T3 (CLK, RSTz, Inc, RST_T3, T, ENABLE);

    parameter Nc=5;
    input CLK,RSTz,ENABLE;
    input Inc;
    input RST_T3;
    output [Nc-1:0] T;

    reg [Nc-1:0] T,nT;

    always @(posedge CLK or negedge RSTz)
    begin
        if (RSTz==0) T=0;
        else T=nT;
    end
```

```
    always @(T or Inc or RST_T3 or ENABLE)
    begin
        if (ENABLE==0) nT=0;
        else begin
          if (RST_T3==1) nT=0;
          else if (Inc==1) nT=T+1;
          else nT=T;
        end
    end

    endmodule
```

****
****

mux_rsa.v: hacen la función de multiplexores.

```
    module mux_rsa_3_1_A (In1, In2, In3, Select_In, Out);

    parameter N=16;
    input [N-1:0] In1;
    input [N-1:0] In2;
    input [N-1:0] In3;
    input [1:0] Select_In;
    output [N-1:0] Out;
    wire [N-1:0] Out;

    assign Out = (Select_In == 2'b00) ? In1 : (Select_In == 2'b01) ? In2 : In3;

    endmodule
```

****
****

```
    module mux_rsa_2_1 (In1, In2, Select_In, Out);

    parameter N=16;
    input [N-1:0] In1;
    input [N-1:0] In2;
    input Select_In;
    output [N-1:0] Out;
    wire [N-1:0] Out;

    assign Out = (Select_In == 1'b0) ? In1 : In2;

    endmodule
```

****
****

test.v: es el módulo de test para nuestro sistema, es decir, instancia al bloque superior (codec_rsa) y produce las listas de resultados.

```
`timescale 1ns / 100ps
`include "codec_rsa.v"

module test_tabla;

parameter N=32; parameter Nc=6;
parameter NDAT=2;

reg [N-1:0] tmp_mem [0:5*NDAT-1];

reg [N-1:0] mem_S [0:NDAT-1];
reg [N-1:0] mem_E [0:NDAT-1];
reg [N-1:0] mem_N [0:NDAT-1];
reg [N-1:0] mem_X [0:NDAT-1];

initial begin $dumpfile("sim.vcd");$dumpvars;end

// Netlist
reg CLK,ENABLE;
reg RSTz;
reg [N-1:0] In_E,In_N;
reg [N-1:0] In_S;
wire [N-1:0] X;
wire Fin;

codec_rsa #(N,Nc) cr0 (.CLK(CLK), .RSTz(RSTz), .In_N(In_N), .In_E(In_E),
            .In_S(In_S), .Nuevo_X(X), .Fin_Enc(Fin), .ENABLE(ENABLE));

parameter T=20;
always begin #(0.5*T) CLK=0;  #(0.5*T) CLK=1; end

// Read memories upon startup
integer i;
initial begin
  $readmemh("tabla.txt",tmp_mem,0,5*NDAT-1);
  for(i=0;i<NDAT;i=i+1) begin
    mem_S[i]=tmp_mem[5*i];
    mem_E[i]=tmp_mem[5*i+1];
    mem_N[i]=tmp_mem[5*i+2];
    mem_X[i]=tmp_mem[5*i+3];
  end
```

```
    for(i=0;i<NDAT;i=i+1)
        $display("S=0x%x E=0x%x N=0x%x X=0x%x",mem_S[i],
                mem_E[i], mem_N[i], mem_X[i] );
  end

  time t0;
  integer dt;

  integer maxIT; initial maxIT=0;
  integer minIT; initial minIT='hffffff;

  initial begin
    RSTz=0; CLK=1; ENABLE=0;
    #(10*T) RSTz=1;

    for(i=0;i<NDAT;i=i+1) begin
      In_S=mem_S[i];
      In_E=mem_E[i];
      In_N=mem_N[i];
      #(T) ENABLE=1; t0=$time;
      wait(Fin==1);  dt=($time-t0)/T;

      if(X==mem_X[i]) $display("X=0x%x (0x%x) OK %0d cycles",X,mem_X[i],dt);
      else          $display("X=0x%x (0x%x) KO",X,mem_X[i]);
      ENABLE=0; #(2*T);
      if(maxIT<dt) maxIT=dt;
      if(minIT>dt) minIT=dt;
    end

    $display("Max %0d / Min %0d cycles", maxIT, minIT);

    #(10*T) $finish;
  end

  endmodule
```

****
****

6.1. CÓDIGO EN LENGUAJE MATLAB.

Ahora se mostrará el código matlab utilizado para la obtención de las diferentes entradas al sistema, y para representar la simulación de nuestro sistema construido en Verilog, y comprobar que los resultados conseguidos son correctos.

prueba.m: función que genera las entradas al sistema, comprobando si cumple las diferentes restricciones.

```
long=2;
s = fix(65535*rand(long,1));

for i=1:1:long
  e(i) = fix(65535*rand);
  while (esprimo(e(i))==0) e(i) = fix(65535*rand);end
  n(i) = fix(65535*rand);
  while ((n(i)<s(i)) | (esprimo(n(i))==0)) n(i) = fix(65535*rand);end
  x(i,1)=rsa(s(i),e(i),n(i),16);
end

tabla=[s,e,n,x];
```

****
****

esprimo.m: función que comprueba si un número es primo. Para ello primero lo divide entre números primos menores y después aplica el teorema de Fermat de números primos.

```
function sal=esprimo(num)

primos=[2 3 5 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101
103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317
331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443
449 457 461 463 467 479 487 491 499 503 509 521 523 541 557 563 567 571 577 587
593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709
719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853
857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991
997];

sal=0;
i=1;
suma=0;
```

```
for (i=1:1:166)
 if (num==primos(i)) sal=1;
 else if (mod(num,primos(i))~=0) suma=suma+1;
 end
end

if (sal==0)
 if (suma==166)
  if (rsa(2,num,num,16)==2) sal=1; end  %% teorema de fermat
 end
end

end
```

****
****

rsa.m: función encargada de realizar la exponenciación modular, para ello utiliza la función monpro.m, que realiza el producto de Montgomery de dos entradas.

```
function F = rsa(s,e,n,k)
% s = string
% e = exponente
% n = modulo
% k = resolucion

c=modulo(2*k,n);
x = 1;
ss = monpro(s,c,k,n);
x1 = monpro(1,c,k,n);

for i = 1:k      % Desde MSB hasta LSB.
   x1 = monpro(x1,x1,k,n);
   if bitn(k-i,e)==1
      x1 = monpro(ss,x1,k,n);
   end
end

x = monpro(x1,1,k,n);
F = x;
```

****
****

monpro.m: se encarga de realizar el Producto de Montgomery de dos enteros de k bits, respecto a otro entero (módulo).

```
function x = monpro(a,b,k,n)

% Esta funcion es usada para calcular el producto
% de Montgomery de los enteros a y b, modulo n, todos de k bits.

u = 0;

for i = 1:1:k
  % From the most significant to the least significant
  u = u + b*bitn(i-1,a);
  if bitn(0,u) == 1
    u = u + n;
  end
  u = fix(u/2);
end

if u >= n
  u = u - n;
end

x = u; % Returns the value of the Montgomery product
```

****
****

gentab.m: función que genera una tabla de entradas, y sus respectivas salidas. Se utilizará para comprobar el correcto funcionamiento de nuestro sistema.

```
long=100;
s = fix(60000*rand(long,1));
for i=1:1:long

 p(i) =fix(255*rand);
 while ((p(i)<150) | (esprimo(p(i))==0)) p(i) = fix(255*rand);end

 q(i) =fix(255*rand);
 while ((q(i)<150) | (esprimo(q(i))==0)) q(i) = fix(255*rand);end

 n(i,1) = p(i)*q(i);
 fi(i,1)=(p(i)-1)*(q(i)-1);

 while (n(i,1)<s(i))
  p(i) =fix(255*rand);
  while ((p(i)<150) | (esprimo(p(i))==0)) p(i) = fix(255*rand);end
  q(i) =fix(255*rand);
  while ((q(i)<150) | (esprimo(q(i))==0)) q(i) = fix(255*rand);end
  n(i,1) = p(i)*q(i);
  fi(i,1)=(p(i)-1)*(q(i)-1);
```

```
  end

  e(i,1) = fix(65535*rand);
  while (esprimo(e(i,1))==0 | e(i,1)>((p(i)-1)*(q(i)-1)))
            e(i,1) = fix(65535*rand);
  end

  x(i,1)=rsa(s(i),e(i,1),n(i,1),16);

  d(i,1)=3;
  while (mod(e(i,1)*d(i,1),fi(i,1))~=1) d(i,1)=d(i,1)+1;end

  sout(i,1)=rsa(x(i,1),d(i,1),n(i,1),16);

end
tabla=[s,e,n,x,d,sout];
```

****
****

modulo.m: calcula el resto inicial de la división entre la potencia de dos y el módulo.

```
function x = modulo(power,modulus)
% Calcula el valor 2^power MOD modulus

done = 0;
k = power/2;
nk = modulus;
m = ceil(log2(modulus));
r = 2^m - nk;
q = 2*k - m;

while q ~= 0
  rtmp = 2*r - nk;
  if rtmp > 0
    r = rtmp;
  else
    r = 2*r;
  end
  q = q - 1;
end
x = r; % Devuelve el valor del resto.
```

****
****

7. BIBLIOGRAFÍA.

- "Técnicas criptográficas de protección de datos". Amparo Fustes Sabater. Editorial RA-MA.
- "Basic Methods of Cryptography". Jan C.A. Van del Lube. Universidad de Cambridge.
- "Criptografía digital". Jose Pastor Franco. Edición Prensas Universitarias de Zaragoza.
- "Generación de Números Pseudoaleatorios usados en Sistemas Criptográficos". J.J. Angel A. CryptoNotas Vol. 1, no. 1, 1998.
- "High-Speed RSA Implementation". Cetin Kaya Koc. RSA Laboratories, RSA Data Security, Inc.
- "Timming Attacks on implementations RSA, DSS and other systems". Kocher, P. C.
- "Fast implementations of RSA Cryptography". M. Shand, J. Vuillemin. Paris Research Laboratory.
- "Cryptanalysis of Short RSA Secret Exponents". M. J. Wiener. IEEE Transaction on Information Theory.
- www.rsa.com (RSA Data Security)
- www.seguridata.com/rsa
- http://quetzal.uis.edu.co/criptografia (biblioteca virtual de criptografía)
- www.rsasecurity.com/rsalabs (laboratorio RSA)
- www.guia.hispavista.com/informatica/seguridad (guía informática)
- www.clarkson.edu
- www.opencores.org

- www.connotech.com/MONTGOM.HTM
- http://servlet.java.sun.com/logRedirect/industry-news/http://www.mips.com.
- www.sci-worx.com/internet/designobjects/
- www.xilinx.com
- ftp://ftp.funet.fi/pub/crypt/math/ (ftp de RSA math)
- www.millersv.edu/~bikenaga/absalg/exteuc.htm (Algorit. Euclideo Extendido)
- www.dice.ucl.ac.be/crypto/dhem/these/these.htm (Librería para diseño de sistema criptográfico de clave pública).