UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA INDUSTRIAL:
ELECTRÓNICA INDUSTRIAL

PROYECTO FIN DE CARRERA

IMPLEMENTACIÓN HARDWARE DE ALGORITMOS CRIPTOGRÁFICOS PARA RFID

AUTOR: HUGO IZQUIERDO DONOSO
TUTOR: ENRIQUE SAN MILLAN HEREDIA

FEBRERO DE 2009

INDICE

CAPÍTULO 3: IMPLEMENTACIÓN DE ALGORITMOS

INDICE DE FIGURAS

INTRODUCCIÓN DEL PROYECTO

1. OBJETIVO DEL PROYECTO

Actualmente los Sistemas de Identificación Automáticos se están convirtiendo en algo muy popular, de aquí el nacimiento de una nueva tecnología para la transmisión de datos, entre un transmisor (Transponder) y un lector, sin contacto físico. Estos son los llamados sistemas RFID. Esta tecnología consiste en la transmisión de datos por radio frecuencia, y va a ser en uno de los aspectos de esta tecnología en lo que se va a fundamentar nuestro proyecto.

En este proyecto queremos realizar un sistema para la encriptación de los datos enviados a través de la tecnología RFID, de modo que los datos que sean enviados estén encriptados y no sea posible su manipulación. La aplicación para la que se quiere realizar la encriptación es para las tarjetas tags. Estas tarjetas envían información a un receptor u otras tarjetas a través de señales de radio frecuencia., esa información enviada es la que se quiere proteger encriptando estos datos. El sistema de encriptado el cual se quiere implementar consta de 3 partes:

a) Protocolo de encriptado el cual el receptor y el emisor establecen una comunicación de autentificación para que los datos que envíen estén protegidos. En este proyecto este protocolo no se ha implementado pero se realizará un ejemplo de protocolo de encriptación llamado SLAP.
b) Para el protocolo de encriptación como es el caso del SLAP necesita de una serie de algoritmos, una parte importante de estos algoritmos es lo que se conoce como generador de números aleatorios, en este proyecto se implementará un generador de números aleatorios de modo que se pueda utilizar en el protocolo de comunicación.
c) A su vez el generador de números aleatorios que queremos implementar necesita de un algoritmo el cual realice la multiplicación modular. En este proyecto se ha implementado diferentes algoritmos que realizan la multiplicación modular de modo que se ha podido comparar entre ello cual es el mejor para nuestra aplicación.

Por tanto la mayor parte del proyecto constará en buscar diferentes algoritmos de implementación modular y comparar todos estos en área que ocupa y el tiempo que tarda. Considerando que en nuestro circuito que queremos implementar estos algoritmos no se trata de un microprocesador ni de una fpga sino que se trata de un

circuito hecho a medida, las características que se buscará en los algoritmos será que ocupe un área limitada y que el tiempo de computo no sea excesivamente alto.
Una vez implementado los diferentes algoritmos de multiplicación modular y hechas las comparaciones, aplicaremos los algoritmos que consideremos más eficientes en un generador de números aleatorios, de modo que en esta aplicación podremos ver cuáles de los algoritmos que previamente se han seleccionado se adaptan mejor en nuestra aplicación. Por último se va a realizar también una implementación del algoritmo de exponenciación modular el cual también se puede aplicar en el generador de números aleatorios.

El proyecto quedará organizado de la siguiente manera:

a) En primer lugar se hablará de los conceptos básicos sobre la tecnología RFID, en este apartado se realizará una introducción sobre esta tecnología y sus aplicaciones.
b) A continuación se estudiará un ejemplo de un protocolo de encriptación, se desarrollará el protocolo SLAP.
c) Posteriormente se hablará de los conceptos básicos sobre el lenguaje utilizado en la programación, en este proyecto se programará en VHDL.
d) Se realizará la implementación de los diferentes algoritmos de multiplicación modular que se han considerados para este proyecto. Para cada algoritmo se implementará en vhdl, se realizará la simulación, seguidamente se realizará la síntesis para finalmente obtener los resultados en cuanto a área y tiempo. Por último una vez obtenido los resultados de los algoritmos se realizará la comparación de todos ellos, de la comparación descartaremos aquellos de los cuales no son válidos para nuestro sistema.
e) De los algoritmos que hemos considerado que se pueden aplicar al sistema de RFID los probaremos en un generador de números aleatorios. El generador de números aleatorios que se implementará será el de Blum Blum & Shub. En este caso se realizará la implementación en vhdl como en el caso de los multiplicadores modulares, después se realizará las simulaciones y posteriormente se realizará la síntesis para finalmente obtener los resultados en cuanto a área ocupada y tiempo que tarda en realizar el cálculo.
f) Por último para finalizar el proyecto se implementará un algoritmo de exponenciación modular en el cual se necesitará los algoritmos de multiplicación modular.

A continuación se muestra los algoritmos que se van a implementar.

a) Algoritmo basado en un circuito combinacional.
b) Algoritmo basado en un circuito secuencial, al que denominaremos algoritmo clásico.
c) Algoritmo Karatsuba Offman.
d) Algoritmos de Barret.
e) Algoritmo de Montgomery.
f) Algoritmo de Buckley.

De todos estos algoritmos se va a elegir el que sea más favorable para nuestra aplicación.

TECNOLOGÍA RFID

[RADIO FREQUENCY IDENTIFICATION]

RFID – PROTOCOLO SLAP – MULTIPLICACIÓN MODULAR

2. TECNOLOGÍA RFID

RFID (siglas de Radio Frequency IDentification, en español Identificación por radiofrecuencia) es un sistema de almacenamiento y recuperación de datos remoto que usa dispositivos denominados etiquetas, transpondedores o tags RFID. El propósito fundamental de la tecnología RFID es transmitir la identidad de un objeto (similar a un número de serie único) mediante ondas de radio. Las tecnologías RFID se agrupan dentro de las denominadas Auto ID (Automatic Identification, o Identificación Automática).

La tecnología RFID [1] está orientada a almacenamiento de datos y su transmisión, en el mercado existen varias tecnologías para introducir fácilmente la información de un producto en un sistema informático. La más popular es sin duda el código de barras, el cual es muy barato y fácil de implantar. No obstante presenta numerosos inconvenientes:

a) Puede almacenar poca información. Por ejemplo los códigos de barras usados en los productos alimentarios contienen sólo 12 dígitos.

b) Son relativamente difíciles de leer: es necesario orientar perfectamente el lector con el código para obtener una lectura correcta.
c) No pueden ser modificados una vez impresos.
d) Se deterioran fácilmente

Debido a esto, desde hace años se está popularizando una tecnología de identificación por radiofrecuencia (RFID) que presenta numerosas ventajas frente a los códigos impresos:

a) Es difícil destruir
b) Es fácil de leer, incluso es capaz de leer varios productos a la vez.
c) Contiene mayor cantidad de información
d) No necesita contacto directo, debido a que la lectura se realiza por radiofrecuencia.
e) Permite la lectura del objeto en movimiento.

[1] [1] [2] y [3] bibliografía

Una de las pocas desventajas que encontramos en la tecnología RFID frente al código de barras es su coste, debido a que el código de barras es un sistema que se puede realizar por impresión mientras que la tecnología RFID necesita de un circuito integrado.

2.1 FUNCIONAMIENTO RFID

El modo de uso de la tecnología RFID es similar al tradicional código de barras. Al producto que se desea identificar se le añade una etiqueta y se utiliza un lector conectado a un ordenador para obtener la información de identificación automáticamente. No obstante, las similitudes terminan ahí: tanto la etiqueta como el lector son totalmente diferentes. El principio de funcionamiento es el siguiente: el lector emite una señal electromagnética que al ser recibida por la etiqueta hace que ésta responda mediante otra señal en la que se envía codificada la información contenida en la etiqueta.

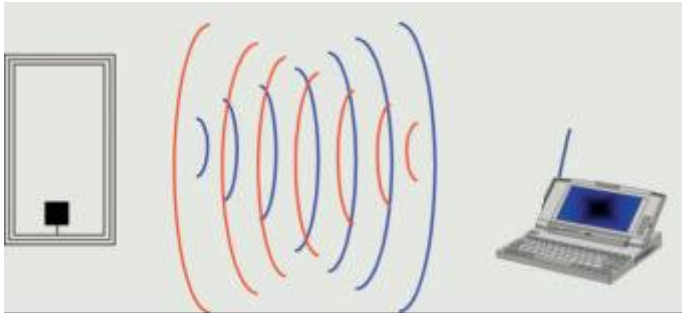


Figura 1 – TECNOLOGIA RFID

El funcionamiento de la tecnología RFID se resumen a continuación:

a) El lector manda una señal de interrogación a la etiqueta.
b) La etiqueta utiliza usa la energía de esta señal para funcionar, y su frecuencia como reloj.
c) La etiqueta lee los datos del lector, en caso de que existan.
d) El RFID contesta con su propia información.
e) Un protocolo de autentificación permite gestionar la respuesta simultánea de múltiples etiquetas.
f) Un protocolo de seguridad permite dar seguridad a la información que se transmite entre el lector y la tarjeta.
g) La información recibida se integra con el resto de sistemas de información.

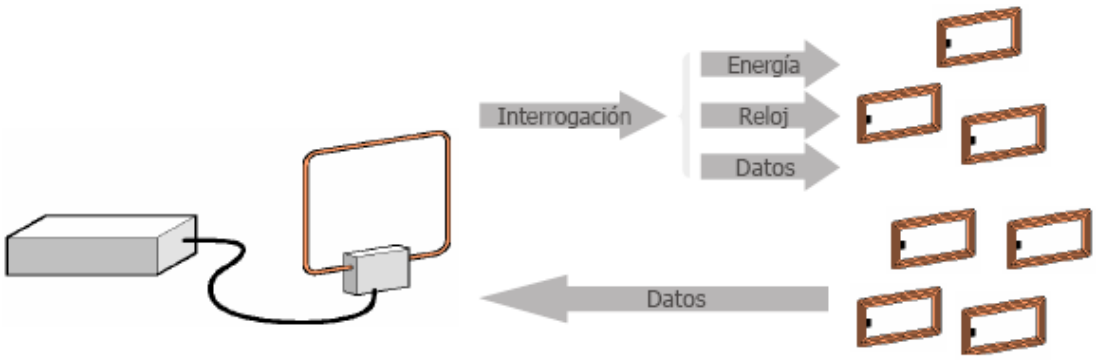


Figura 2 – TECNOLOGIA RFID 2

2.1.2 LA ETIQUETA RFID

En una etiqueta RFID podemos distinguir tres elementos: la antena, el circuito integrado y el elemento almacenador de energía.

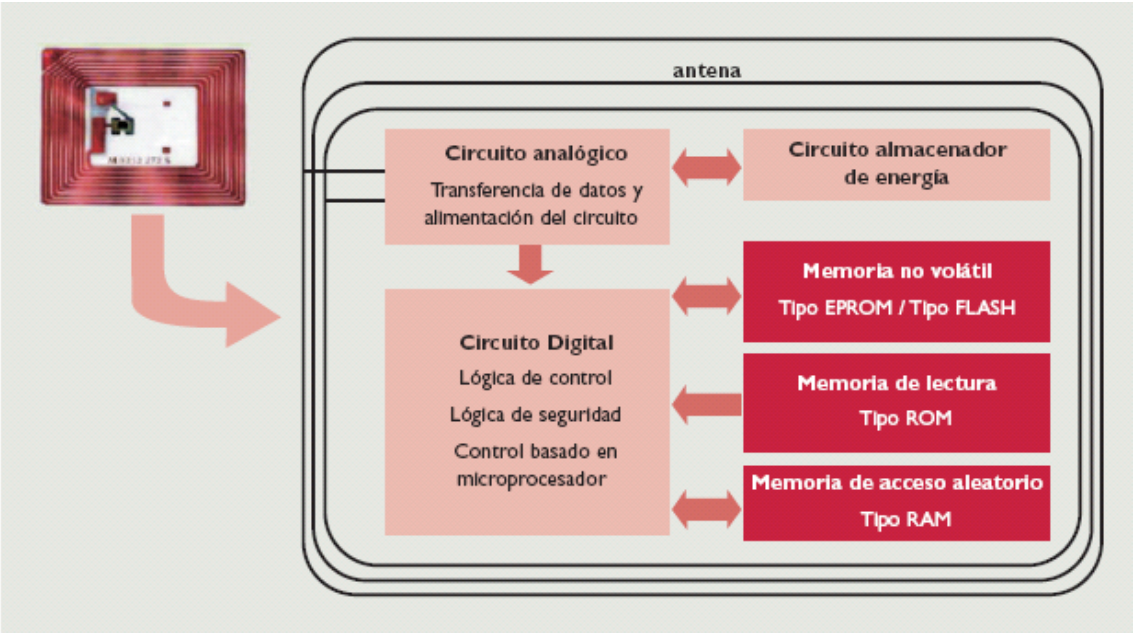


Figura 3 – TECNOLOGIA RFID 3

La antena permite realizar la comunicación entre la etiqueta y el lector. Su tamaño limita la distancia máxima a la que puede realizarse la lectura.

El circuito integrado es un circuito mixto analógico- digital. La parte analógica se encarga de controlar la alimentación y la comunicación por radiofrecuencia. Por otro lado, la parte digital gestiona la información almacenada en la etiqueta.
Por último, es necesario incluir un elemento para alimentar al circuito. En función del elemento usado existen dos tipos de etiquetas: las activas y las pasivas.

En las etiquetas pasivas, el elemento almacenador de energía es un condensador, el cual se carga con la energía emitida por el lector y luego utiliza dicha energía para responder. Por ello, la potencia de emisión está limitada, por lo que la distancia entre el lector y la etiqueta no puede ser muy elevada. La ventaja obvia de este tipo de etiquetas es el ahorro de espacio, la duración prácticamente ilimitada de la etiqueta y su menor coste. Debido a esto, éstas son las etiquetas más usadas, aplicándose en campos tan diversos como la identificación de animales, llaves de contacto de automóviles, identificación de productos en cadenas de montaje, control de accesos, cronometraje de carreras, etc. Obviamente, los componentes mencionados han de protegerse del ambiente exterior, por lo que en función de la aplicación habrá de elegirse el encapsulado adecuado. El más sencillo es el que se ha mostrado en la Figura 2 consistente en una lámina de plástico. No obstante los fabricantes ofrecen innumerables encapsulados, incluso a medida del cliente. Por ejemplo existen etiquetas en formato tarjeta de crédito para control de accesos, encapsuladas en una ampolla de vidrio para identificar animales, etiquetas en forma de clavo para palets, encapsulados resistentes a altas temperaturas para etiquetar equipos que tengan que soportar condiciones adversas, etc.

A diferencia de los tags pasivos, los activos poseen su propia fuente autónoma de energía, que utilizan para dar corriente a sus circuitos integrados y propagar su señal al lector. Estos tags son mucho más fiables (tienen menos errores) que los pasivos debido a su capacidad de establecer sesiones con el reader. Gracias a su fuente de energía son capaces de transmitir señales más potentes que las de los tags pasivos, lo que les lleva a ser más eficientes en entornos dificultosos para la radiofrecuencia como el agua (incluyendo humanos y ganado, formados en su mayoría por agua), metal (contenedores, vehículos). También son efectivos a distancias mayores pudiendo generar respuestas claras a partir de recepciones débiles (lo contrario que los tags pasivos). Por el contrario, suelen ser mayores y más caros, y su vida útil es en general mucho más corta.

Muchos tags activos tienen rangos efectivos de cientos de metros y una vida útil de sus baterías de hasta 10 años. Algunos de ellos integran sensores de registro de temperatura y otras variables que pueden usarse para monitorizar entornos de alimentación o productos farmacéuticos. Otros sensores asociados con ARFID incluyen humedad, vibración, luz, radiación, temperatura y componentes atmosféricos como el etileno. Los tags, además de mucho más rango (500 m), tiene capacidades de almacenamiento mayores y la habilidad de guardar información adicional enviada por el transceptor.

Actualmente, las etiquetas activas más pequeñas tienen un tamaño aproximado de una moneda. Muchas etiquetas activas tienen rangos prácticos de diez metros, y una duración de batería de hasta varios años.

En la siguiente figura se muestra el interior de las etiquetas utilizadas para la tecnología RFID.

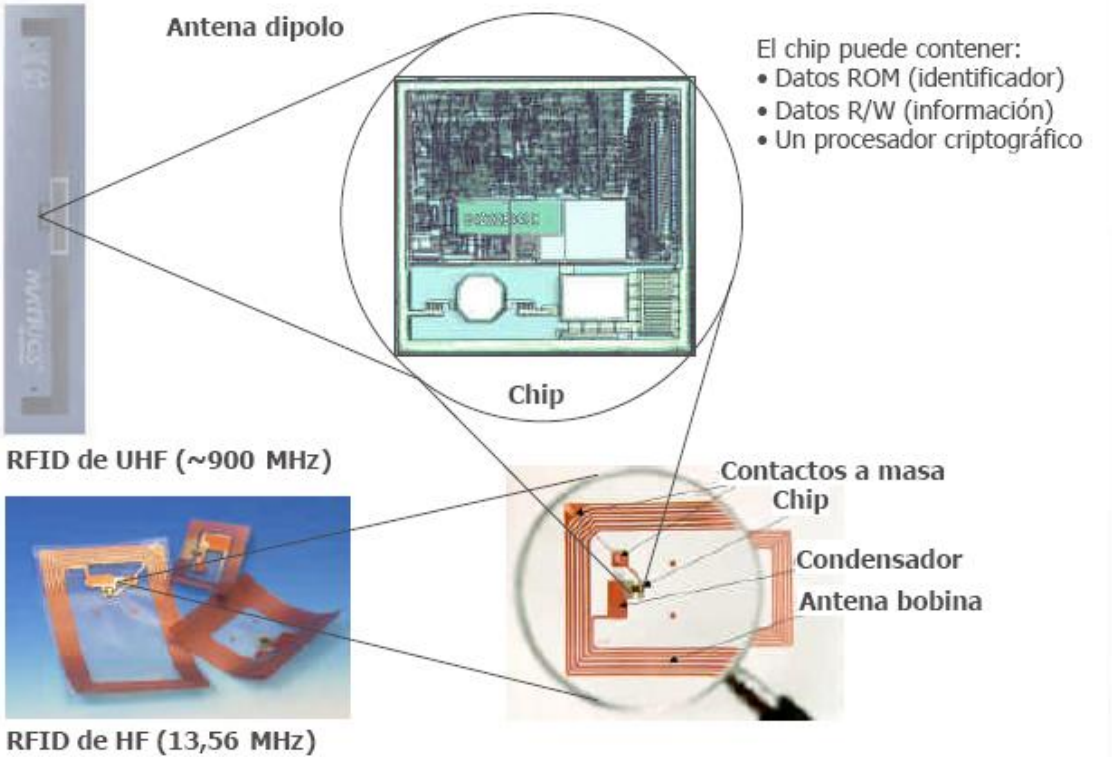


Figura 4 – TECNOLOGIA RFID 4

2.1.3 EL LECTOR RFID

La estructura del equipo de lectura es similar a la de las etiquetas: es necesaria una antena para comunicarse con la etiqueta y un circuito para gestionar la comunicación. Este circuito dispone de un interfaz estándar, como por ejemplo RS-232 o CompactFlash, para conectarse a un ordenador o a una PDA. Además existen en el mercado equipos lectores con la antena integrada y equipos que admiten antenas externas, que pueden seleccionarse en función de la aplicación. En este último caso, existen dos tipos de antenas, las antenas tipo "cuadro", similares a las usadas en los grandes almacenes para evitar robos y las antenas con núcleo de ferrita. Las primeras, al tener un mayor tamaño pueden alcanzar distancias de lectura del orden de un metro, aunque con poca direccionabilidad. Las segundas en cambio tienen un tamaño más reducido por lo que su rango se reduce a unos pocos centímetros. Sin embargo su menor tamaño permite su uso en equipos portátiles. Además son más direccionales, por lo que se pueden leer etiquetas que estén próximas entre sí.

El software

El último componente en un sistema de identificación por RFID es el sistema de proceso de datos. Las etiquetas sólo de lectura devuelven un código único grabado "a fuego" al fabricar el chip (un tamaño típico es 64 bits.) Las etiquetas de lectura/escritura, que son más caras que las anteriores, permiten leer y/o escribir una longitud mayor de información (valores típicos son 32, 256 o 2048 bits). Por tanto será necesario usar algún sistema de bases de datos que realice la correlación entre la información devuelta por la etiqueta y el resto de información del producto. A modo de conclusión, en la siguiente figura se resume en forma gráfica el sistema completo RFID.

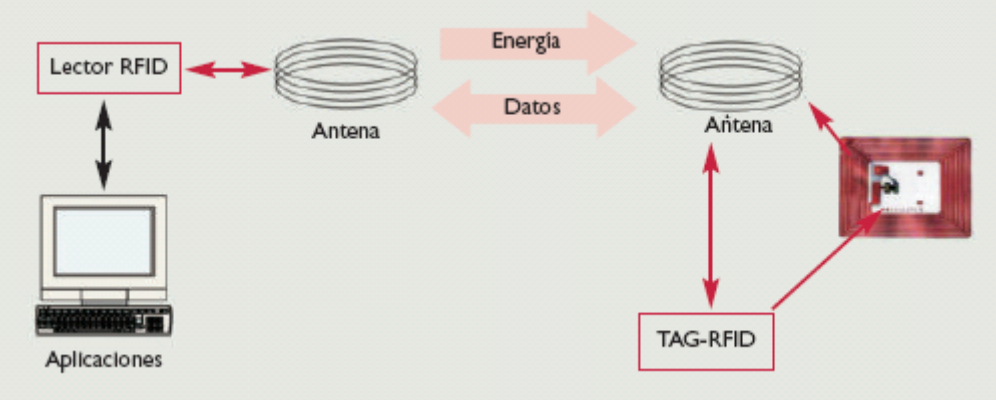


Figura 5 – TECNOLOGIA RFID 5

2.1.4 COLISIONES

Las etiquetas más sencillas necesitan ser leídas de una en una, ya que si se sitúa más de una dentro del rango de alcance de la antena del lector, ambas interferirán entre sí y no podrá realizarse la lectura. Obviamente este tipo de etiquetas son más baratas y para muchas aplicaciones son más que suficientes. Sin embargo, existen aplicaciones en las que puede ser necesario leer varias etiquetas a la vez. Ejemplos de este tipo de aplicaciones son los sistemas de cronometraje, en los que varios corredores pasan simultáneamente por la antena lectora; el etiquetado de equipajes, etc. Para este tipo de aplicaciones se han desarrollado etiquetas anticolisión que permiten la lectura de varias etiquetas situadas en el rango de alcance de la antena. Además la lectura es bastante rápida

2.1.5 FRECUENCIAS EN RFID

Las frecuencias utilizadas para esta tecnología RFID es la mostrada en la figura siguiente:

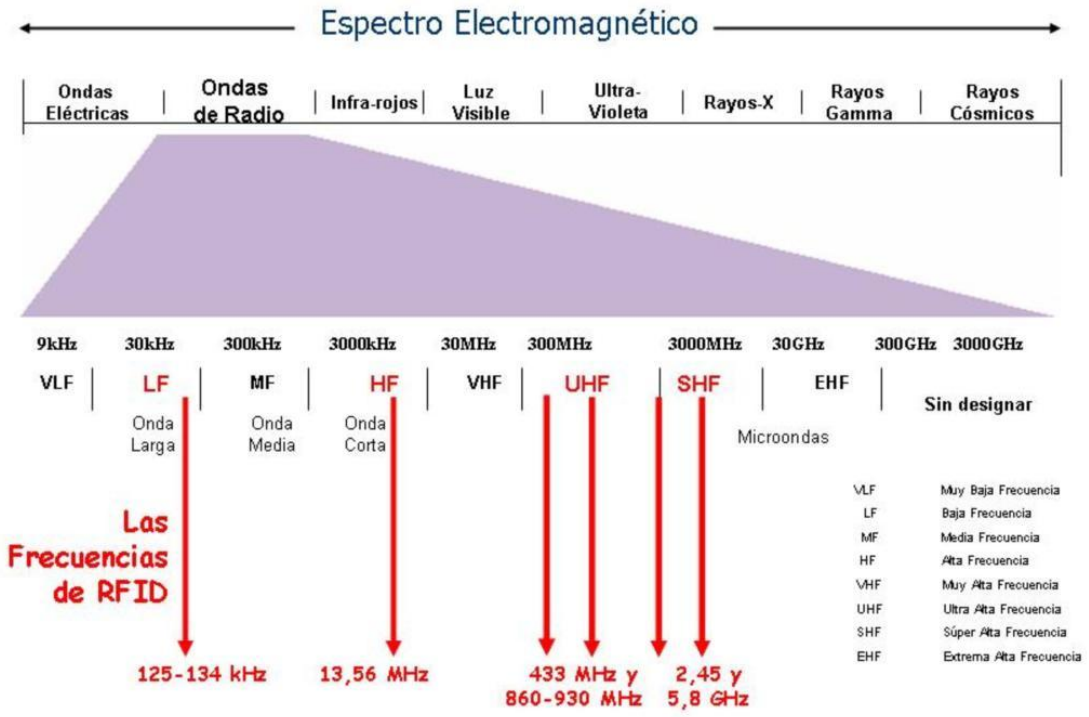


Figura 6 – ANCHO DE FRECUENCIA RFID

Los sistemas de comunicaciones están estandarizados por organismos internacionales (UIT, FCC, CCITT, etc.) que regulan su uso y operatividad. Además, estos organismos son también los encargados de repartir el espectro de frecuencia entre todas las aplicaciones que lo necesitan. Para las aplicaciones de RFID, se han asignado tres bandas principales de frecuencias, que se resumen en la Tabla 1, donde se indican además las características típicas de los sistemas que usan dicha banda, así como sus aplicaciones típicas.

Banda de frecuencia	Características	Aplicaciones típicas
Baja 100-500 KHz	Lectura para corta y media distancia Sistemas con tags económicos Velocidad de lectura baja	Control de acceso Identificación de animales Control de existencias Inmovilizadores de automóviles
Intermedia 10-15 MHz	Lectura para corta y media distancia Potencialmente barato Velocidad de lectura media	Control de acceso Tarjetas inteligentes
Alta 850-950 MHz 2.4-5,8 GHz	Lectura para corta y media distancia Velocidad de lectura alta Línea de vista requerida Tecnología cara	Supervisión en sistemas ferroviarios y automotriz.Acceso y control de peaje

Figura 7 – CARACTERISTICAS FRECUENCIA RFID

En la tabla anterior se han mostrado unos márgenes de frecuencia muy amplios en los que se encuadran los rangos asignados para RFID en cada uno de los países. Así, en función de la zona en la que estemos situados tendremos que usar un rango de frecuencia u otro, lo que implica usar etiquetas distintas. En la Figura 4 se muestran las tres regiones en las que se divide el planeta en función de las regulaciones de radiofrecuencia: Europa y África (Región 1), América (Región 2) y lejano oriente con Australasia (Región 3). Afortunadamente los fabricantes disponen de versiones de cada una de sus etiquetas para las distintas regiones. Por ejemplo las etiquetas Ucode HSL de Philips están disponibles para una banda de 869 MHz (región 1) y para una banda de 900 MHz (región 2)

Figura 8 – REGIONES RFID

Otro parámetro importante que varía entre regiones es la máxima potencia de emisión, la cual está directamente relacionada con la distancia máxima de lectura de las etiquetas. Siguiendo el ejemplo anterior, en la región 1, donde para la banda de 869 MHz se admite una potencia máxima de emisión de 0,5 W, la distancia máxima para un tamaño típico de antenas es de 4m. En cambio las mismas etiquetas en la región 2, en donde la potencia máxima permitida es de 4 W, pueden leerse desde 8,4 m.

2.1.6 ÁREAS DE APLICACIÓN PARA RFID

El uso potencial de RFID es prácticamente ilimitado en cada sector de la industria, comercio y servicios donde existen datos que deben ser leídos o comprobados. Las áreas principales de aplicación de RFID son:

a) Transporte y logística.
b) Fabricación y procesamiento.
c) Seguridad de personas.
d) Identificación y trazabilidad alimentaria animal.
e) Rastreo postal.
f) Verificación y control de equipaje.
g) Control de peaje y medios de pago electrónico.
h) Sustitución o uso simultáneo y compartido con códigos de barras.
i) Vigilancia electrónica.
j) Control de accesos y un largo etc.

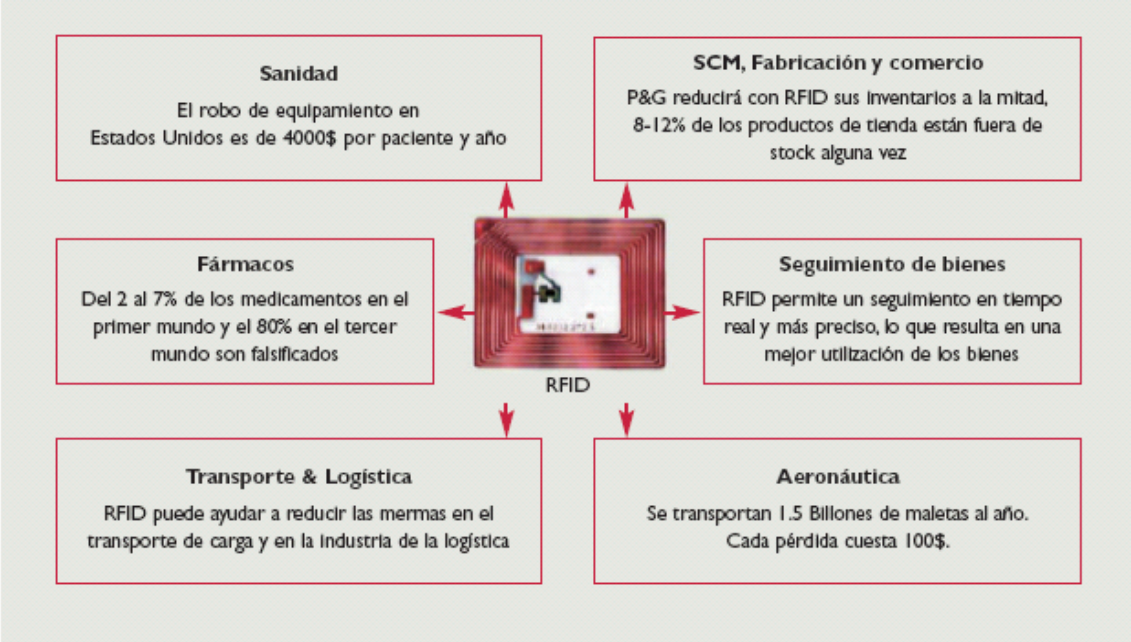


Figura 9 – APLICACIÓN RFID

2.1.7 CONCLUSIONES

Hasta ahora se ha realizado una introducción a la tecnología RFID, la cual presenta numerosas ventajas frente a los métodos tradicionales de identificación como el código de barras. Se ha mostrado que su característica fundamental consiste en la utilización de etiquetas "inteligentes" capaces de almacenar mayor cantidad de información de forma segura y perdurable. No obstante, aunque esta tecnología se vislumbra como el futuro en las aplicaciones de identificación, aún necesita superar algunas barreras para que se popularice aún más. Aparte de los problemas de estandarización que se resolverán con el tiempo, el principal escollo es el precio de las etiquetas, que es claramente desfavorable frente al coste cero del código de barras. No obstante, la mayor facilidad de uso hace que el coste total de la aplicación pueda ser menor. Por ejemplo, una aplicación que impactaría en nuestra vida diaria sería el uso de etiquetas RFID para identificar los productos de un supermercado. Con este sistema bastaría con pasar por la antena lectora para que se registrasen todos los productos del carro. El día en que el precio de las etiquetas sea menor que el coste del tiempo empleado en pasar el producto por el lector de código de barras, veremos este tipo de etiquetas cada vez que hagamos la compra.

2.2 PROTOCOLO SLAP

Para poder utilizar la tecnología RFID hay que solucionar uno de los problemas que tiene esta tecnología que es la de los ataques de intrusos. Para ello se puede utilizar un protocolo de autentificación que nos sirva para proteger la información que se envía. Uno de los protocolos que podríamos utilizar sería el protocolo SLAP [2], es un protocolo de autentificación para RFID basado en modular exponentiation (ME).

2.2.2 TABLA DE NOTACION

Simbolo	**Significado**
Sid	*El ID number de la etiqueta*
Pid	*Numero ID entre la etiqueta y el lector*
Er	*Numero de exponenciación secreto del lector en cada sesion*
Et	*Numero de exponenciación secreto de la tarjeta en cada sesión*
N	*Cambio de valor usado en la etiqueta de anonimato*
Kt	*secure ME creado por la etiqueta*
Kr*	*secure ME guardado en el lector*
Kr(i)	*secure ME creado por el lector*
Tt	*Tag-side secure ME*
Kj	*Llave de seguridad usada en la etiqueta*
R	*Numero aleatorio de 64 bits creados por el lector*
J	*Contador de 8 bits usados en la etiqueta*
I	*Numero de éxito de la sesión*
DATA	*Toda la informacion sobre la etiqueta*
Z	*Numero primo usado en la actualizacion de la llave*
X	*(sid) XOR ® XOR (Kj)*
Tr	*Reader-side secure ME*
r (8 bit)	*registro*

Figura 10 - SLAP

[2] [8] bibliografía

2.2.3 DISEÑO DEL PROTOCOLO

Lo primero que se realizará será la encriptación y desencriptación a través de ME, se recoge el proceso en la siguiente figura:

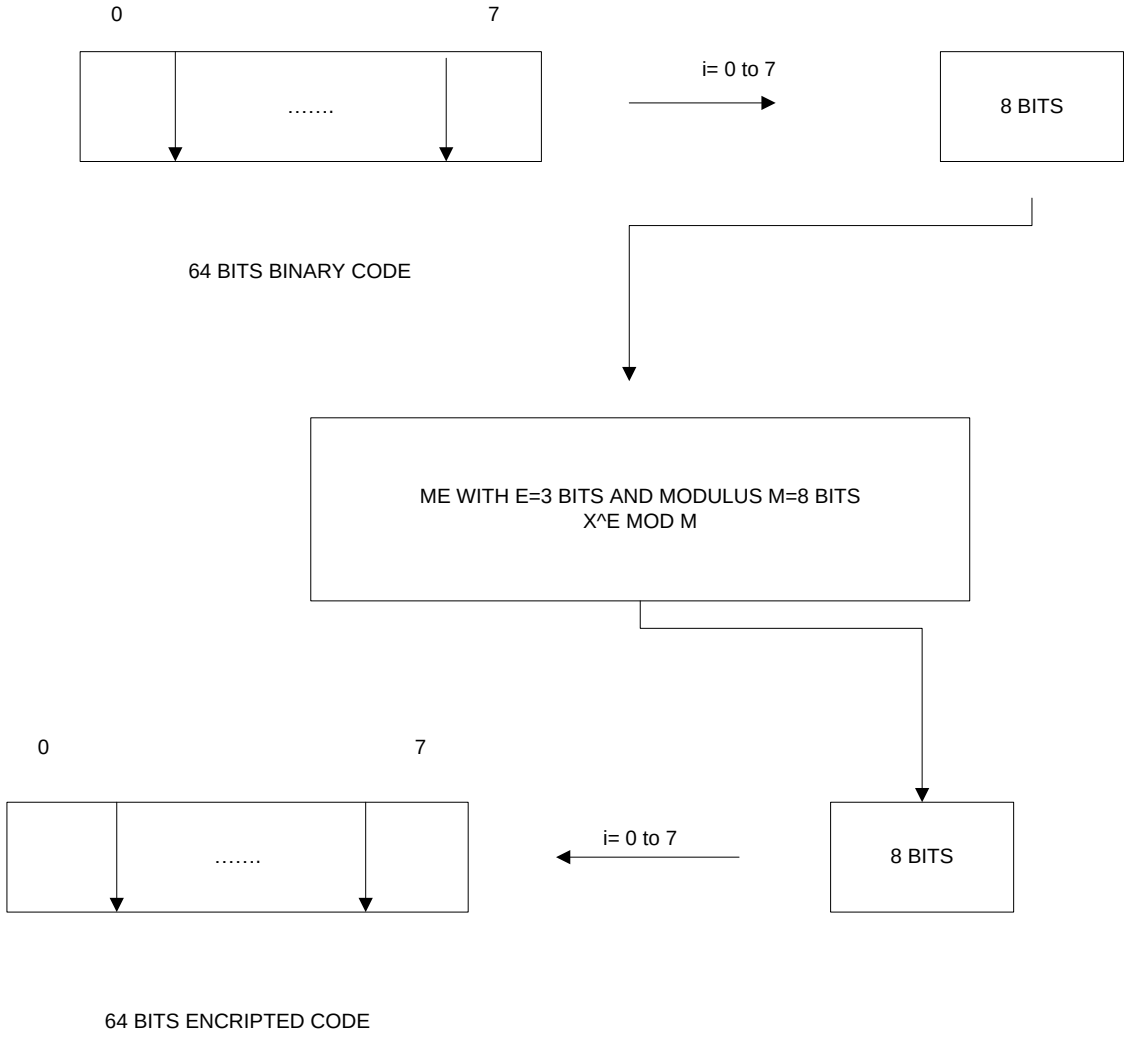


Figura 11 – SLAP 2

Después M y E deben actualizarse con un numero R aleatorio, además el modulo M debe ser numero primo de ahí que el valor encriptado del ith nunca debe ser igual a 0.

El protocolo se puede dividir en 3 partes: identificación de la etiqueta, autentificación mutua y actualización de las llaves de seguridad

1. *FASE DE INICIO*

Para cada tag, el servidor almacena inicialmente seis valores en la etiqueta: sid, pid, et, Kj,Kt, se obtiene el Tt , después de esta fase se puede pasar a la fase de autentificación mutua.

2. *FASE DE AUTENTIFICACIÓN*

Antes de empezar con la fase de autentificación el lector debe identificar a la etiqueta. El proceso es el siguiente:

a) El lector envía un numero aleatorio r a la etiqueta
b) La tarjeta calcula el valor de X usando el valor de r,sid, y Kj, y envía X y j al lector.
c) El lector recibe X y j, y consigue sid a través de X usando j. El lector compara el sid obtenido con el enviado por la tarjeta y si es igual la tarjeta ha sido identificada.

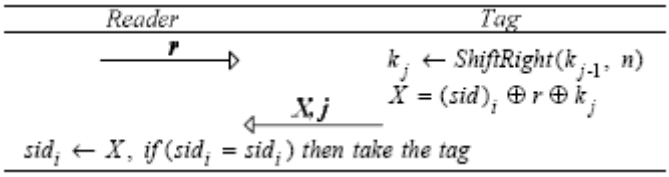


Figura 12 – SLAP 3

En la fase de autentificación se dan las siguientes fases:

a) El lector recibe el ME, Kr del DB, y lo envía a la etiqueta.
b) La tarjeta verifica que el Kr es igual al Kt de la tarjeta, si es igual la tarjeta autoriza al lector.
c) Entonces la etiqueta genera el Tt, y se lo envía al lector. Además la etiqueta debe ejecutar la llave de actualización del proceso, ya que si no el lector no estará autentificado y la tarjeta permanecerá en silencio.
d) El lector comparará el Kr con el Kr*, si son iguales el lector autentificará a la etiqueta consiguiendo el numero de identificación de la etiqueta, TagID, del DB. El lector además debe ejecutar la llave de actualización del proceso , ya que si no se realiza esto la etiqueta permanecerá en silencio.

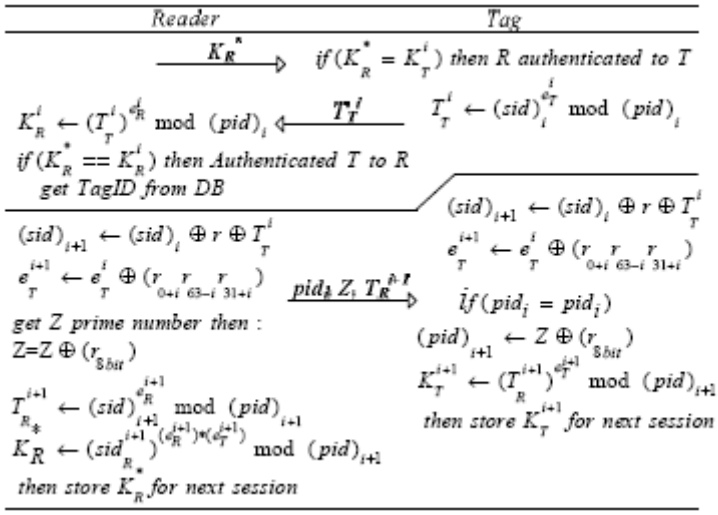


Figura 13 – SLAP 4

3. <u>ACTUALIZAR LLAVE</u>

La tercera fase que ser realiza en este protocolo sería la de la actualización de la llave. Las fases en las cuales se realiza la actualización son las siguientes:

a) Después de que el lector sea autentificado, la etiqueta debe actualizar dos valores: sid, y et de la siguiente manera:

$$(sid)_{i+1} \leftarrow (sid)_i \oplus r \oplus T_T^i$$

$$e_T^{i+1} \leftarrow e_T^i \oplus (r_{0+i}\, r_{63-i}\, r_{31+i})$$

b) Después de que la etiqueta sea autentificado el lector debe actualizar los valores de sid,pis,et,er,n,Kr* estos valores serán utilizados en la siguiente sesión. Para actualizar el pid el lector selecciona el numero primo Z, que será calculado a través de Z y el XOR del registro r. El lector envía pid, Z, Tr a la etiqueta.

$$Z \leftarrow Z \oplus (r_i\, r_{7-i}\, r_{15+i}\, r_{23-i}\, r_{31+i}\, r_{39-i}\, r_{47+i}\, r_{55-i}\, r_{63-i})$$

$$T_R^{i+1} \leftarrow (sid)_{i+1}^{e_R^{i+1}} \bmod (pid)_{i+1}$$

$$K_R^* \leftarrow (sid_R^{i+1})^{(e_R^{i+1})*(e_T^{i+1})} \bmod (pid)_{i+1}$$

c) El Kr* es el valor encriptado del lector y este lo reemplaza con valor antiguo de la sesión i, en el database. Este valor será usado como llave de autentificación en la próxima sesión.

d) Después de recibir Z,pid y Tr, la etiqueta chequea el pid para determinar si los valores recibidos son correctos. Z es un numero primo que ha sido calculado al

hacer la XOR de Z con los 8 bits del numero r, por tanto para obtener Z habría que hacer la misma XOR con los 8 bit del numero r para obtener Z. Después de obtener Tr del lector, la etiqueta realiza la ecuación =>

$$(pid)_{i+1} \leftarrow Z \oplus (r_i r_{7-i} r_{15+i} r_{23-i} r_{31+i} r_{39-i} r_{47+i} r_{55-i} r_{63-i})$$

$$K_T^{i+1} \leftarrow (T_R^{i+1})^{e_T^{i+1}} \bmod (pid)_{i+1}$$

e) El valor de Kt(i) será reemplazado en la EEPROM, el valor Kt(i+1) es el valor encriptado de la etiqueta que usará para autentificar al lector en la siguiente sesión.

2.2.4 ANALISIS DE SEGURIDAD.

El protocolo debe ser seguro, y debe asegurar la seguridad del anonimato de la etiqueta como la prevención de la escucha.

a) *Seguridad del ME*: En esta propuesta se encripta el código con una llave privada et,er y modulo pid usando byte a byte, por tanto este protocolo se puede considerar seguro.

b) *Ataques a la etiqueta de identificación*: El sid es una llave compartida usada como llave de identificación de la etiqueta. Un intruso podría escuchar la comunicación entre la etiqueta y el lector. Pero X es un valor aleatorio y j no tendría sentido para el intruso, por tanto el intruso no podría identificar la etiqueta. El Kj y sid son llaves compartidas y X es un valor que se calcula a través de estos dos, el intruso no podría distinguir entre sid y X. Ademas después de cada sesión tanto el sid, Kj como el valor de n son actualizados. De este modo el intruso no podría rastrear la etiqueta en este momento.

c) *Ataques a la mutua autentificación*: Kr* será usado para autentificar el lector a la etiqueta y del mismo modo Tt será usado al revés. A través de la SE ME encriptará estos valores a través de sus llaves privadas er,et y pid. Esos dos valores serian como un numero aleatorio para el intruso, ya que no podría distinguir pid, del las llaves privadas er y et. Después de cada autentificación et, er y pid deben ser actualizados, para que Kr* t Tr sean numero aleatorios y sin sentidos para el intruso.

d) *Ataques a la actualización de llaves*: Z es un numero aleatorio, Tr(i+1) es una llave de seguridad del lector. El lector calcula el Tr(i+1) después de actulizar er y pid, por ello el intruso no puede identificar a la etiqueta. Además el intruso podría intentar atacar a la actualización de la información secreta de la tarjeta. Sin embargo, las llaves de seguridad serán actulizadas después de que el lector autentifique a la tarjeta y el intruso no puede saber el valor del pid, porque no

conoce el bit elegido para el proceso del pid. A pesar de que el intruso no conoce el pid este valor no sería útil en la siguiente sesión. Por tanto el intruso no puede realizar el ataque con éxito.

2.2.5 CONCLUSION

Este protocolo se podría usar para la encriptación de la tecnología RFID, utiliza la exponenciación modular.. Como se observara en este método se utiliza la multiplicación modular para poder realizar las distintas operaciones, existe multitud de algoritmos para poder realizar la multiplicación modular los cuales estudiaremos y compararemos algunos de ellos a continuación. Además se observa como en este protocolo necesita de un generador de números aleatorios de ahí que en este proyecto también implementemos un generador de números aleatorios para poder utilizarlo en el protocolo.

2.3 MULTIPLICACIÓN MODULAR

2.3.2 INTRODUCCIÓN MULTIPLICACIÓN MODULAR

Los esquemas criptográficos modernos se valen de operaciones aritméticas de complejidad apreciable. Tradicionalmente estas operaciones se han implementado en software. Sin embargo, dado el crecimiento continuo de las capacidades de los sistemas de cómputo en los últimos años, ha sido necesario reajustar gradualmente la complejidad de las operaciones para cumplir con los requerimientos de seguridad específicos de cada aplicación. Normalmente estos ajustes hechos en las operaciones criptográficas implican el manejo de operandos de mayor longitud, lo que obviamente desmejora los tiempos de ejecución. Como consecuencia de esto, existe el interés de implementar estos operadores en hardware, de modo que los ajustes a los operadores se puedan hacer cumpliendo simultáneamente con tiempos de ejecución razonables.
La Multiplicación Modular [3] tiene tres operandos de entrada: A, B y M. Cada uno de estos operandos corresponde a un entero sin signo representado como un número binario de una longitud dada.

Para tres números enteros A, B y M, se define la multiplicación modular como: $P \equiv A \cdot B \bmod M$, tal que: $A \cdot B = M \cdot k + P$, siendo k cualquier número entero.

Figura 14 - MODULAR MULTIPLICATION

Para la multiplicación modular pueden existir varios métodos. Entre los más importantes se encuentra los métodos Karatsuba-ofman y Booth como algoritmos de multiplicación, para obtener el modulo con estos métodos habrá que realizar un segundo método a ambos como puede ser el de Barret, y los métodos de Montgomery y Brickell como métodos en los que se utiliza la multiplicación y la reducción en el mismo método. Estos métodos se explican a continuación.

[3] [4] bibliografía

2.2.3 METODO DE KARATSUBA-OFMAN

El algoritmo de Karatsuba-Ofman [4] está considerado como uno de los algoritmos más rápidos a la hora de multiplicar enteros de gran tamaño. En este algoritmo se realizará: una multiplicación de 2-n dígitos que se reducirá en 2 multiplicaciones de n dígitos, una multiplicación de (n+1) dígitos, 2 restas de n-dígitos, 2 operaciones de desplazamiento a izquierda, 2 sumas de n-dígitos y 2 sumas de 2n-digitos.

En cuanto al desarrollo del algoritmo es el siguiente:

Se coge dos valores X e Y los cuales pueden ser dos números cualquiera enteros. Queremos calcular el producto de X * Y, los cuales podemos descomponer respectivamente en partes iguales en sus zonas altas y bajas X_H , X_L, y Y_H , Y_L respectivamente. Tomamos K=2n, (siendo n el numero de bits) si fuese impar lo completaríamos con 0 a la izquierda.

$X = XH\ 2n + XL$
$Y = YH\ 2n + YL$

*P=X*Y se calcula de la siguiente manera:*

$P = XY = (X_H\ 2^n + X_L)(Y_H\ 2^n + Y_L) = 2^{2n}(X_H Y_H) + 2^n(X_H Y_L + X_L Y_H) + X_L Y_L$
Usando esta ecuación se necesitan 4 multiplicaciones de N bits para calcular el producto P.

Desarrollando la ecuación anterior el cálculo de P se puede mejorar de la siguiente manera:

$$X_H Y_L + X_L Y_H = (X_H + X_L)(Y_H + Y_L) - X_H Y_H - X_L Y_L$$

En el algoritmo de Karatsuba-Ofman's observando la ecuación de arriba la multiplicación de 2-n bits se pueden reducir en 3 multiplicaciones de n-bits, $X_H Y_H$, $X_L Y_L$ and $(X_H + X_L)(Y_H + Y_L)$. The Karatsuba-Ofman's puede ser expresado como en el siguiente algoritmo, donde la function Size (x) devuelve el numero de bits de X, la function High(X) devuelve la parte alta de X y Low(X) devuelve la parte baja de X.

[4] [9] [10] bibliografía

Además la función RightShift(X) devuelve $X2^n$ y *OneBitMultiplication* (*X*, *Y*) devuelve XY donde X e Y serán calculados por un bit.

```
Algorithm KaratsubaOfman(X, Y)
   If (Size(X) = 1) Then KaratsubaOfman= OneBitMultiplier(X, Y)
   Else Product1 := KaratsubaOfman(High(X), High(Y));
        Product2 := KaratsubaOfman(Low(X), Low(Y));
        Product3 := KaratsubaOfman(High(X)+Low(X), High(Y)+Low(Y));
        KaratsubaOfman := RightShift(Product1, Size(X)) +
                          RightShift(Product3-Product1-Product2,     Size(X)/2)     +
Product2;
End KaratsubaOfman.
```

Figura 15 - MODULAR MULTIPLICATION 2

2.3.4 MULTIPLICACIÓN DE BOOTH

En este algoritmo [5] la multiplicación normalmente se genera en 2 pasos: el primero genera un producto parcial y el segundo que en el cual se va acumulando los productos parciales. Lo más básico en el algoritmo de multiplicación se basa en el método de add-and-shift: la operación de desplazamiento genera el producto parcial mientras que el sumador los suma todos ellos.

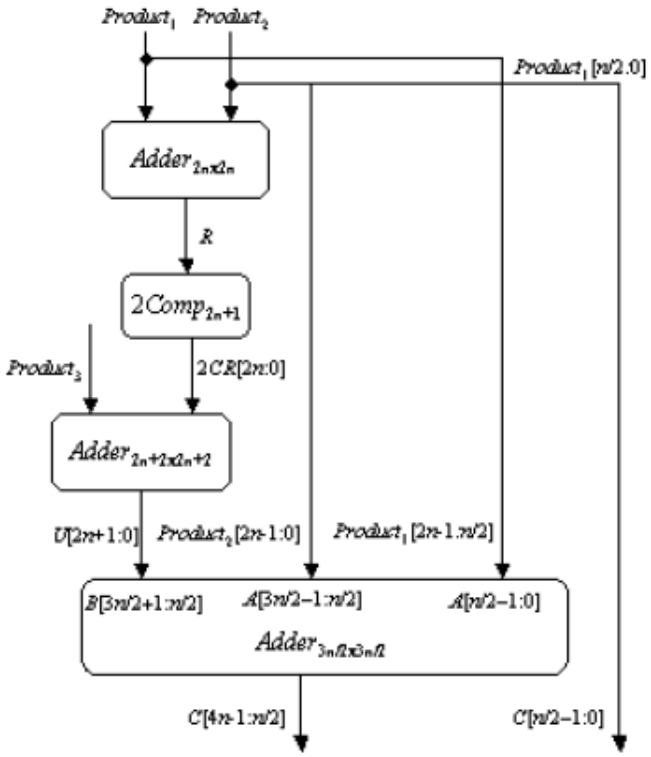


Figura 16 - MODULAR MULTIPLICATION 3

[5] [4] [9] bibliografía

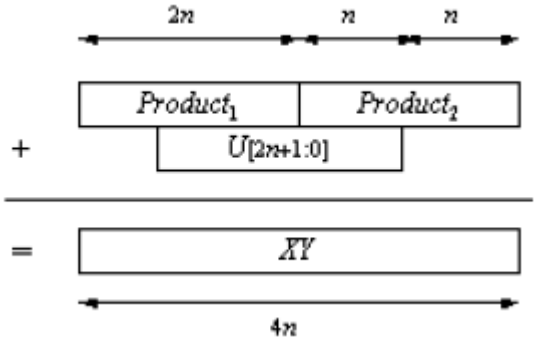


Figura 17 - MODULAR MULTIPLICATION 4

La manera de implementar un multiplicador es la basada en un método iterativo del adder-accumulator para generar los productos parciales mostrados en la siguiente figura. Sin embargo esta solución es bastante lenta ya que el resultado final solo se daría cuando pasasen n ciclos de reloj(siendo n el tamaño de los elementos)

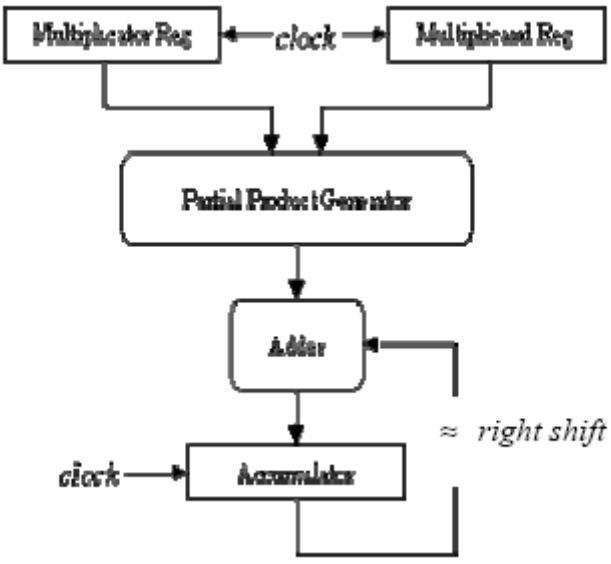


Figura 18 - MODULAR MULTIPLICATION 5

Para una versión más rápida del multiplicador iterativo se tendría que añadir varios productos parciales al mismo tiempo. Esto se podría hacer con un circuito

combinacional el cual consista en varios productos parciales con sumadores que operan en paralelo.

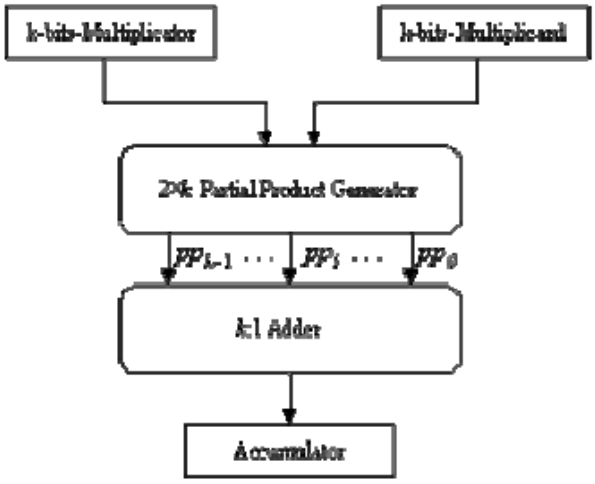


Figura 19 - MODULAR MULTIPLICATION 6

El diagrama de bloques de este algoritmo sería el siguiente:

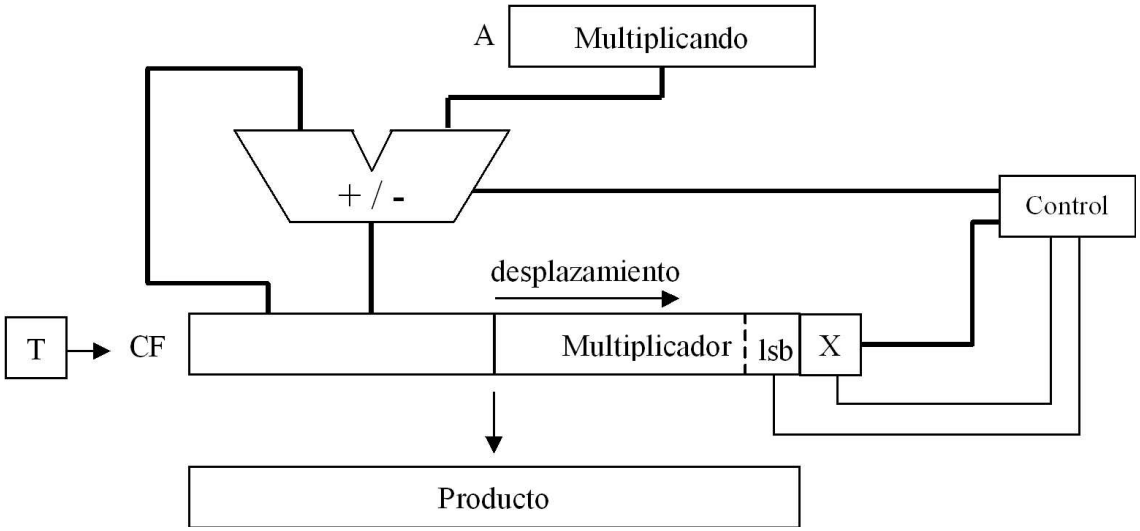


Figura 20 - MODULAR MULTIPLICATION 7

El algoritmo de Booth será el siguiente:

Tomamos X e Y como números enteros, y tomamos n y m como el tamaño de los operandos respectivamente.

Siguiendo el método add-and-shift se genera el cada producto: $Xi*Y$, $0<=i<n$. Cada producto parcial es desplazado a la izquierda o derecha dependiendo de si el bit por el que empieza es el mas o menos significativo.

El numero de productos parciales dependerá de tamaño de los operandos, en criptografía los tamaños de los operandos suele ser bastante grande.

Algorithm Booth($x_{2\times i-1}, x_{2\times i}, x_{2\times i+1}, Y$)
  Int product := 0;
  Int pp[$\lceil (n+1)/2 \rceil - 1$];
  pp[0] := $(\tilde{x}_0 \times Y + 4\times 2^{n+1}) \times 2^{2\times i}$;
  For i = 0 **To** $\lceil (n+1)/2 \rceil - 1$ **Do**
    pp[i] := $(\tilde{x}_i \times Y + 3\times 2^{n+1}) \times 2^{2\times i}$;
    product := product + pp[i];
  Return product mod $2^{n+\lceil (n+1) \rceil - 1}$;
End Booth

La arquitectura del generador de los productos parciales y su implementación es la siguiente:

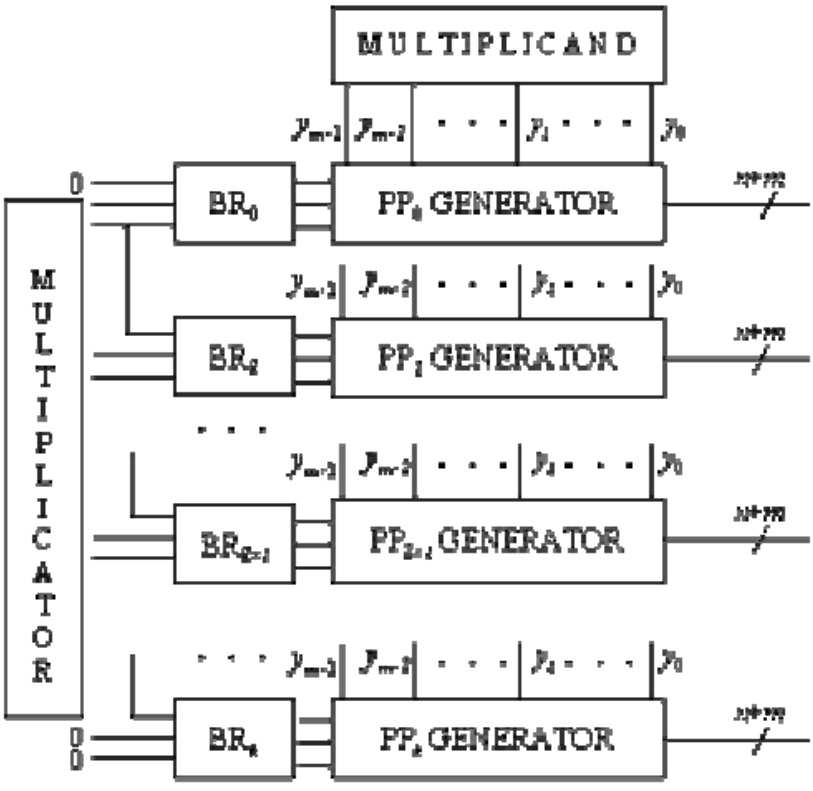


Figura 21 - MODULAR MULTIPLICATION 8

Arquitectura de los generadores de productos parciales

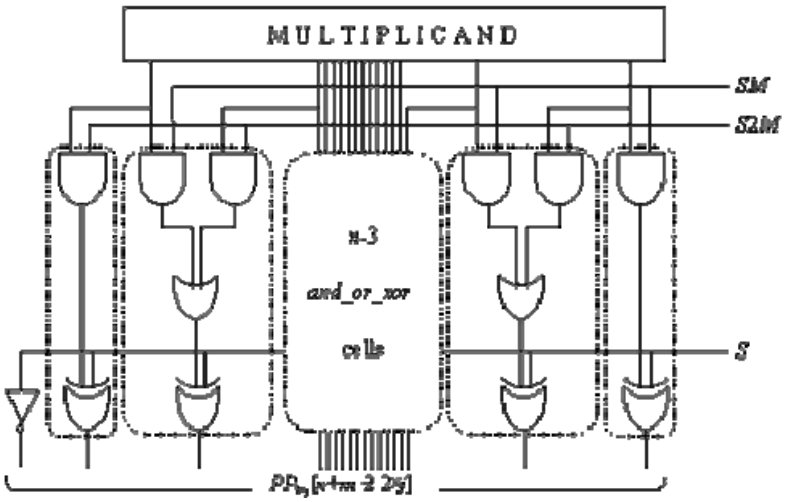


Figura 22 - MODULAR MULTIPLICATION 9

Implementación de los generadores de los productos parciales

2.3.5 COMPARACIÓN BOOTH - KARATSUBA

A continuación se muestra una tabla y gráficos con resultados en cuanto al retraso del tiempo y área que necesita cada multiplicador. La K0 es del algoritmo de Karatsuba-ofman, las demás pertenecen al algoritmo de Booth con diferentes implementaciones de hardware.

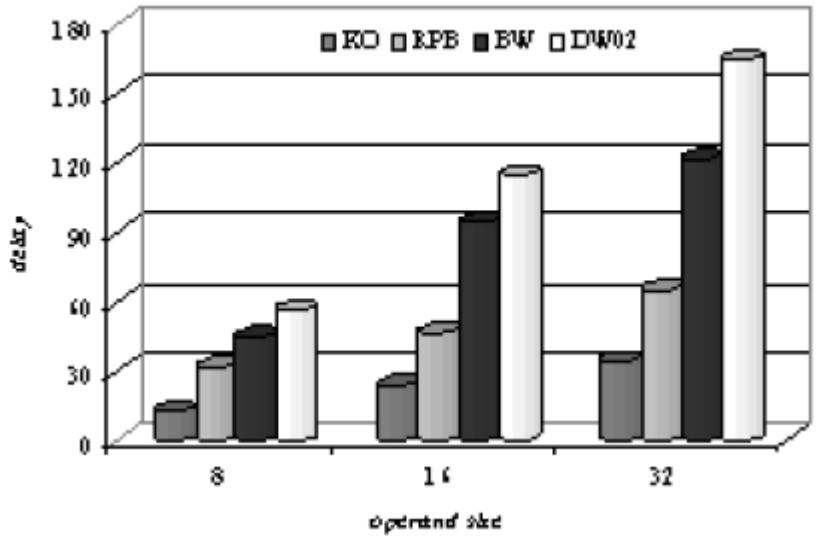


Figura 23 – RETRASO KARAT_BOOTH

TIEMPO REQUERIDO

size	*KO*		*BW*	
	delay	*area*	*delay*	*area*
8	12.6	1297	44.6	1092
16	22.8	6300	93.9	5093
32	29.1	31740	121.5	20097

RETRASOS Y AREAS

El retraso se mide en nanosegundos, mientras que el área se mide en CLBs.

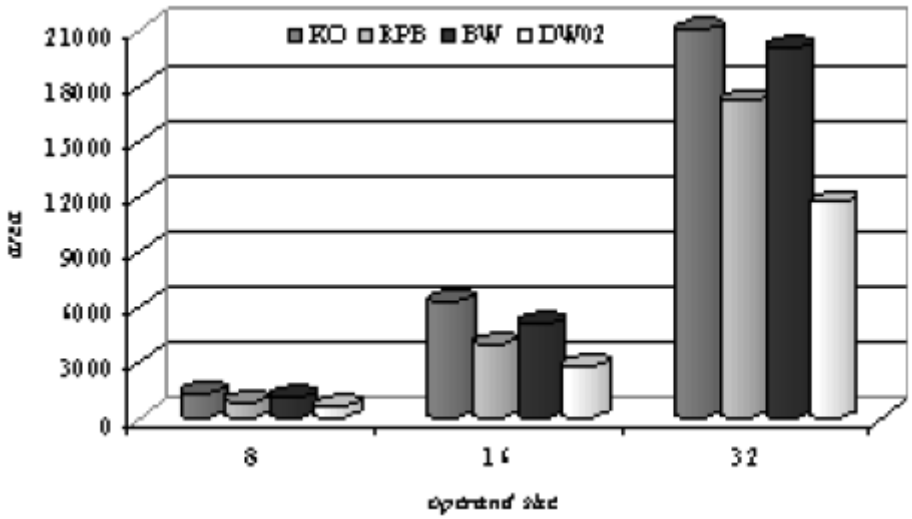


Figura 24 – AREA KARAT_BOOTH

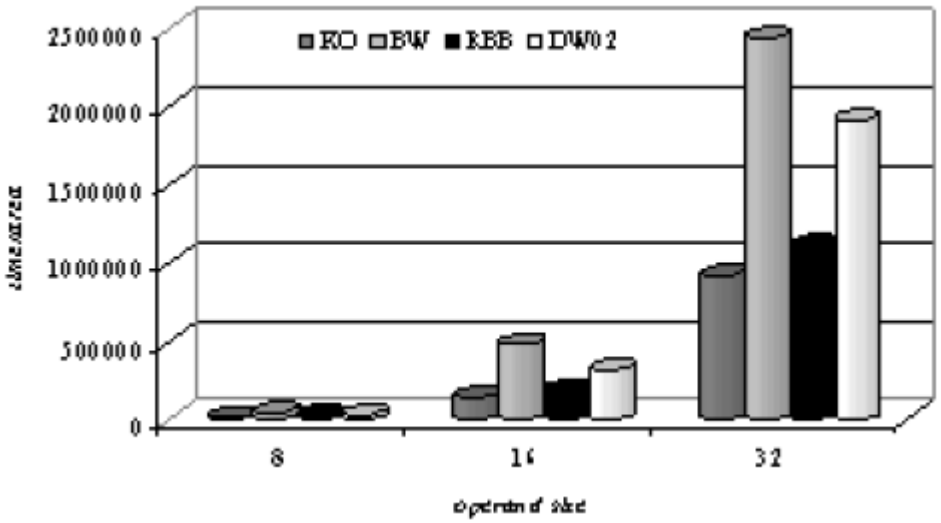


Figura 25 – AREA X TIEMPO KARAT_BOOTH

En resumen se puede considerar observando los resultados que para grandes operandos es mas adecuado el algoritmo de Karatsuba-Ofman, mientras que para operandos pequeños que no excedan de 256 bits sería más convenientes utilizar el método de Booth

2.3.6 METODOS DE REDUCCIÓN DE BARRET

Una vez que se obtiene la multiplicación para la multiplicación modular se necesita una reducción, esto es lo que realiza el método de Barret . Una reducción es simplemente el cálculo del resto de una división.

$$X \bmod M = X - \left\lfloor \frac{X}{M} \right\rfloor \times M$$

Sin embargo una división es mucho más complicada que una multiplicación. Un algoritmo para división es el de NaiveReduction, el cual se muestra a continuación.

```
Algorithm NaiveReduction(P, M)
   Int R := P;
   Do R := R - M;
   While R > 0;
   If R ≠ 0 Then R := R + M;
   Return R;
End NaiveReduction
```

Este algoritmo es muy ineficiente ya que requiere 2n−1 restas, 2n comparaciones y una suma extra.

Otra alternativa a este algoritmo puede ser el siguiente:

```
Algorithm RestoringReduction(P, M)
   Int R0 := P;
   Int N := LeftShift(M, n);
   For i = 1 To n Do
      Ri := Ri-1 - N;
      If R < 0 Then Ri := Ri-1;
      N := RightShift(N);
   Return Ri;
End RestoringReduction
```

Trata de restar el valor más grande de 2 múltiplos de M al resto actual. Cada vez que el resultado de esta operación da negativo se restaura el resto anterior y se repite el cálculo con todos los posibles valores de 2 múltiplos de M: $2nM$, $2n-1M$, …, $2M$, M. Para este algoritmo se requiere n restas, r comparaciones y algunas 2n desplazamientos así como algunas operaciones de restablecimiento. Esto es mucho más eficiente que el algoritmo de Naive

Una alternativa al RestoringReduction es NonRestoringReduction :

```
Algorithm NonRestoringReduction(P, M)
   Int R0 := P;
   Int N := LeftShift(M, n);
   For i = 1 To n Do
      If R > 0 Then Ri := Ri-1 - N;
      Else Ri := Ri-1 + N;
      N := RightShift(N);
   If Ri < 0 Then Ri := Ri-1 + N;
   Return Ri;
End RestoringReduction
```

Este algoritmo permite el resto negativo. Cuando el resto es no negativo se suma a la potencias de dos múltiplos de M. De lo contrario el múltiplo de M los resta. Mantiene hacerlo en repetidas ocasiones para todos los posibles potencias de 2 múltiplos de M: $2nM$, $2n-1M$, …, $2M$, M.

Usando el método de Barret's [6] podemos calcular el resto usando 2 simples multiplicaciones. El cálculo aproximado seria:

$$\left\lfloor \frac{X}{M} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{X}{2^{n-1}} \right\rfloor \times \left\lfloor \frac{2^{n-1} \times 2^{n+1}}{M} \right\rfloor}{2^{n+1}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{X}{2^{n-1}} \right\rfloor \times \left\lfloor \frac{2^{2\times n}}{M} \right\rfloor}{2^{n+1}} \right\rfloor$$

Figura 26 – RESTO BARRET

Con Esta ecuación se puede calcular de forma muy eficiente para divisiones de potencia de 2, para este último método en el caso en el que las divisiones no sean exactas el resultado final sería una aproximación. Para solucionar esto se podría representar los números a través de representaciones en las cuales se pueda obtener decimales como es el caso de coma flotante o punto fijo.

[6] [4] bibliografía

2.3.7 MÉTODO BOOTH - BARRET

Se calcula la multiplicación modular en 3 pasos: X*Y mod M

a) Calcular el producto P=X*Y
b) Calculo de la estimación del cociente: $Q=P/M$
   $Q=P/2^{n-1}\, x[2^{2xn}/M]$
c) Calcular el resultado final

Durante el primer paso, primero se carga en el registro 1 y registro 2 los valores de X e Y. Después espera para que PPG obtenga los productos parciales y finalmente espera que el sumador sume todos los productos. Durante el paso 2, se carga en los registros1,2 y 3 con los valores obtenidos del producto P, el pre-calculado constante 2^{2xn} /M y P respectivamente., entonces espera a que PPG calcule los productos parciales y finalmente los suma todos ellos. Durante el tercer paso, primero se carga en los registro 1 y 2 , con los valores Q y modulo M respectivamente, entonces espera a que el PPG genere los productos parciales y finalmente espera a que el sumador sume todos ellos. Y finalmente espera para que el reductor calcule el resultado final del resultado P-QxM, el cual es cargado en el acumulador acc.

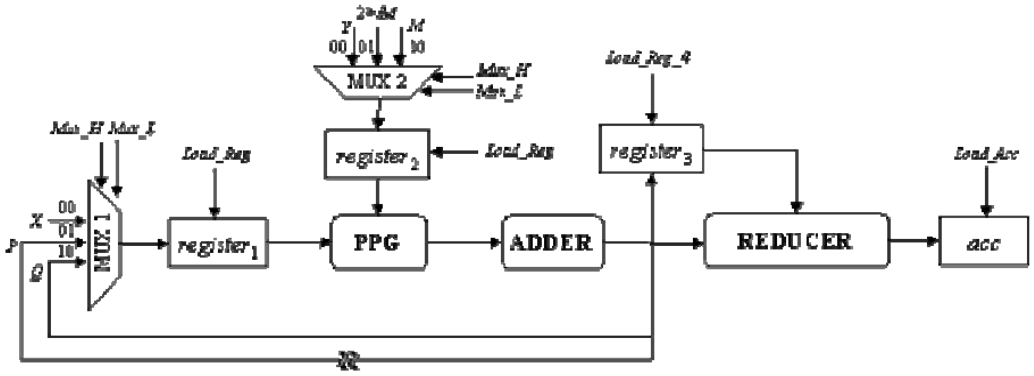


Figure 22: The modular multiplier architecture

Figura 27 - ARQUITECTURA M.MODULAR

2.3.8 ALGORITMO DE MONTGOMERY

Este algoritmo de multiplicación modular se basa principalmente en 2 pasos: el primero en el que se calcula el producto P=AxB y el segundo el cual se realiza la reducción de P mod M.

Para este algoritmo se toma A,B como dos enteros cualquiera y M que será el modulo, siendo n el numero de bits de los operandos. Se define A,B y M de la siguiente manera:

$$A=\sum_{i=0}^{n-1} a_i \times 2^i, \quad B=\sum_{i=0}^{n-1} b_i \times 2^i \text{ and } M=\sum_{i=0}^{n-1} m_i \times 2^i$$

Las condiciones del método de Montgomery [7] son las siguientes:

a) M necesita ser número primo.
b) El multiplicando y el multiplicador necesitan ser más pequeños que M.
c) Como se usa la representación binaria en los operandos, el modulo M necesita ser impar para satisfacer la primera condición.

El algoritmo utilizado para la multiplicación será el siguiente:

```
algorithm Montgomery(A, B, M)
      int R = 0;
      1: for i= 0 to n-1
      2:    R = R + a_i×B;
      3:    if r_0 = 0 then
      4:       R = R div 2
      5:    else
      6:       R = (R + M) div 2;
      return R;
end Montgomery.
```

En la arquitectura utilizada se usan 2 multiplexores, 2 sumadores, 2 registros de desplazamiento, 3 registros y un controlador, en la siguiente figura se muestra esta arquitectura:

[7] [5] [6] [11] bibliografía

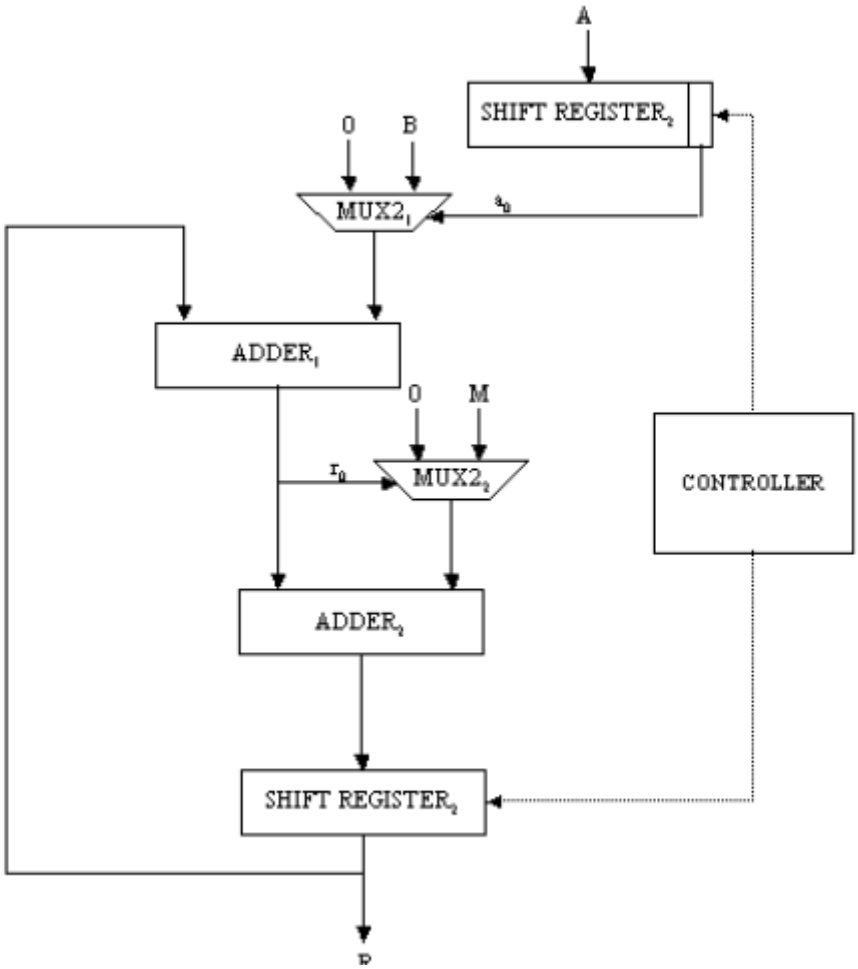


Figura 28 - ARQUITECTURA MONTG 1

La función del controlador consiste en la sincronización de los desplazamientos y cargar las operaciones en los registros de desplazamiento 1 y 2. El controlador usa simplemente un contador descendente. El controlador estará compuesto de la siguiente manera:

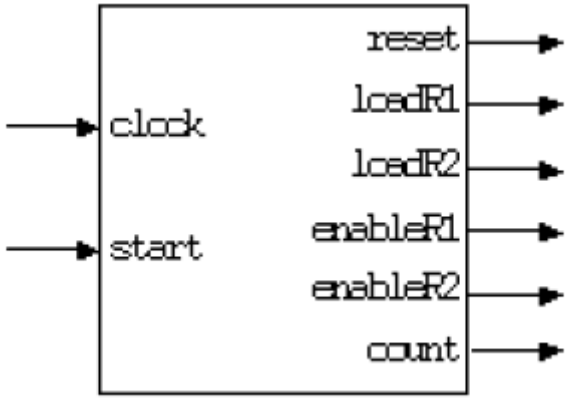


Figura 29 - CONTROLADOR MONTGOMERY

Este controlador es una máquina de estados, los cuales son 6 y son:

a) S0: inicializa la máquina de estados.
   Go to S1

b) S1: Carga el multiplicando y modulo en sus registros
   Carga el multiplicador en el registro de desplazamiento1
   Go to S2
c) Espera al sumador1
   Espera al sumador2
   Carga el multiplicador dentro del registro de desplazamiento2
   Incrementa el contador
   Go to S3
d) Enable al registro de desplazamiento2
   Enable al registro de desplazamiento1
   Go to S4
e) Verifica el contador
   Si es 0 va al S5 sino al S2
f) Detener

Viendo el algoritmo de Montgomery se observa que se necesita una multiplicación modular extra de Montgomery para calcular 2^n mod M, pero debido a que el objetivo de este algoritmo es el cálculo de la exponenciación, es preferible multiplicar antes los operandos de Montgomery por 2^{2n} y después del algoritmo multiplicar el resultado por 2^{-n}. Se necesita una multiplicación extra de Montgomery para obtener la constante r^{2n} mod M. El algoritmo utilizado es el siguiente:

```
algorithm ModularMult(A, B, M, n)
   const C := 2^n mod M;
   int R := 0;
   R := Montgomery(A, B, M);
   return Montgomery(R, C, M);
end ModularMult.
```

La arquitectura es:

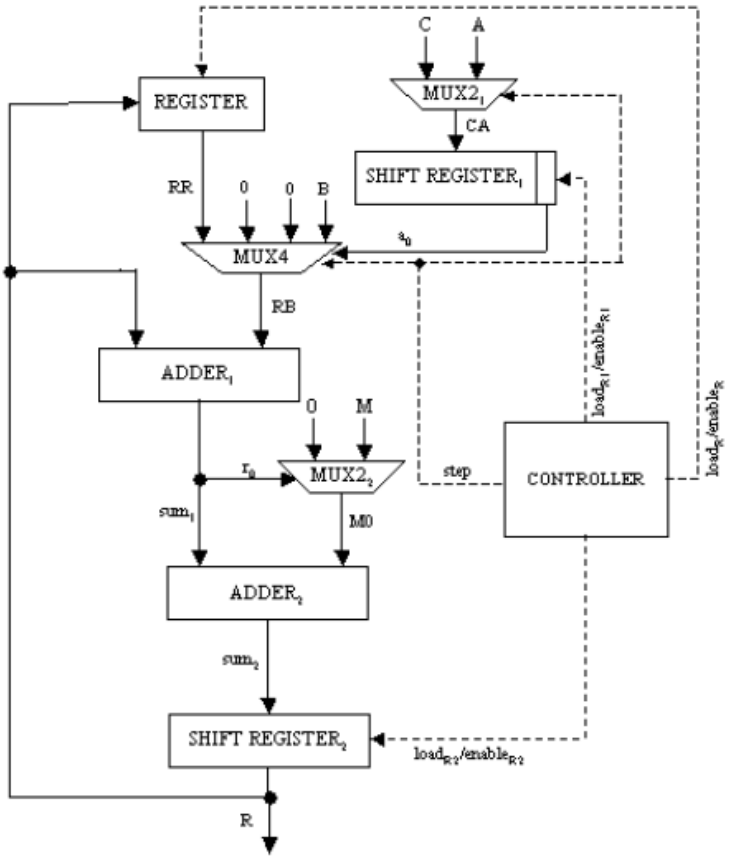


Figura 30 - ARQUITECTURA MONTG 2

La máquina de estados se compone de 10 estados:

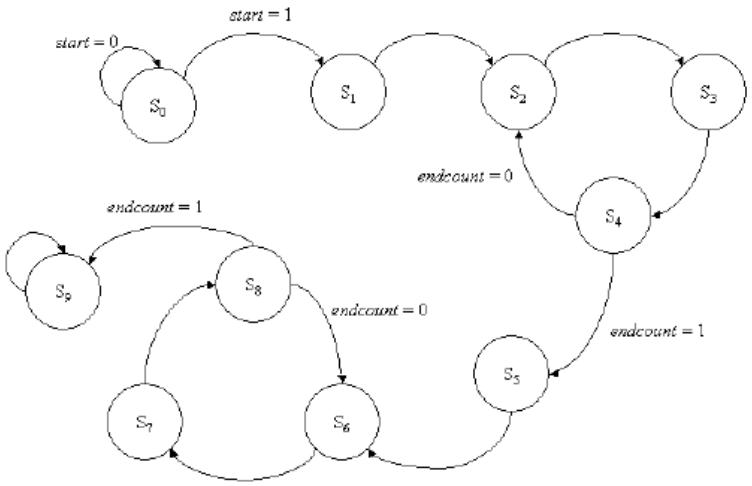


Figura 31 - ESTADOS MONTGOMERY

a) S0: Inicializa la máquina de estados
   Go to S1
b) S1: Carga el multiplicando y modulo, carga el multiplicador dentro del registro1
   Go to S2
c) S2: Espera al sumador1; espera al sumador2
   Carga el resultado parcial en el registro 2
   Incrementa el contador
   Go to S4
d) S3: Enable al registro de desplazamiento2
   Enable al registro de desplazamiento1
   Go to S4
e) S4: Carga el resultado parcial del paso 0 en el registro
   Actualiza el contador
   Si es 0 go to S3 sino al S2
f) S5: Carga constante en el registro1
   Resetea el registro
   Set step to 1, Go to S6
g) S6: Espera al sumador1; espera al sumador2
   Carga el resultado parcial en el registro2
   Incrementa el contador, Go to S7
h) S7: Enable registro de desplazamiento2
   Enable al registro de desplazamiento 1; go to S8
i) S8: Actualiza el contador
   Si es 0 go to S9, sino vuelve al S6
j) Detener

El interface del controlador sería el siguiente:

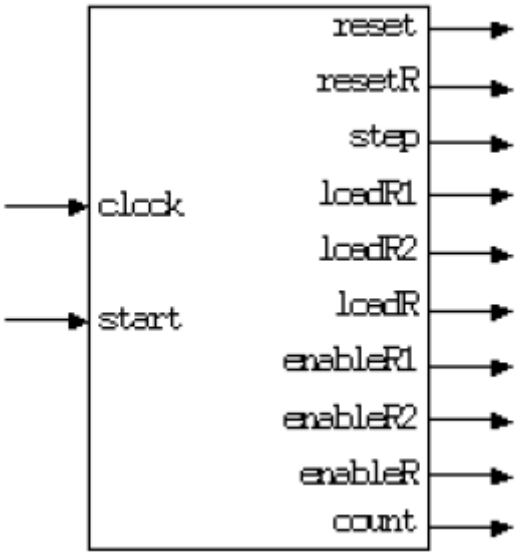


Figura 32 - CONTROLADOR MONTG 2

A continuación se muestra las simulaciones realizadas en la FPGA para los algoritmos 1 y 2.

Montgomery (15,26,47)=34

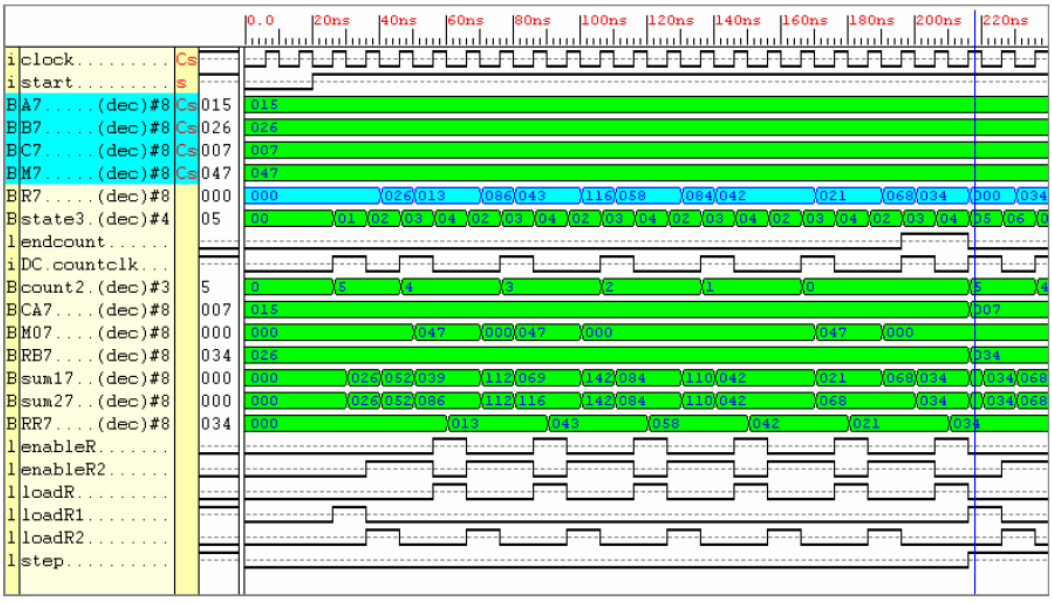


Figura 33 - SIMULACION MONTG 1

Montgomery (7,34,47)=14

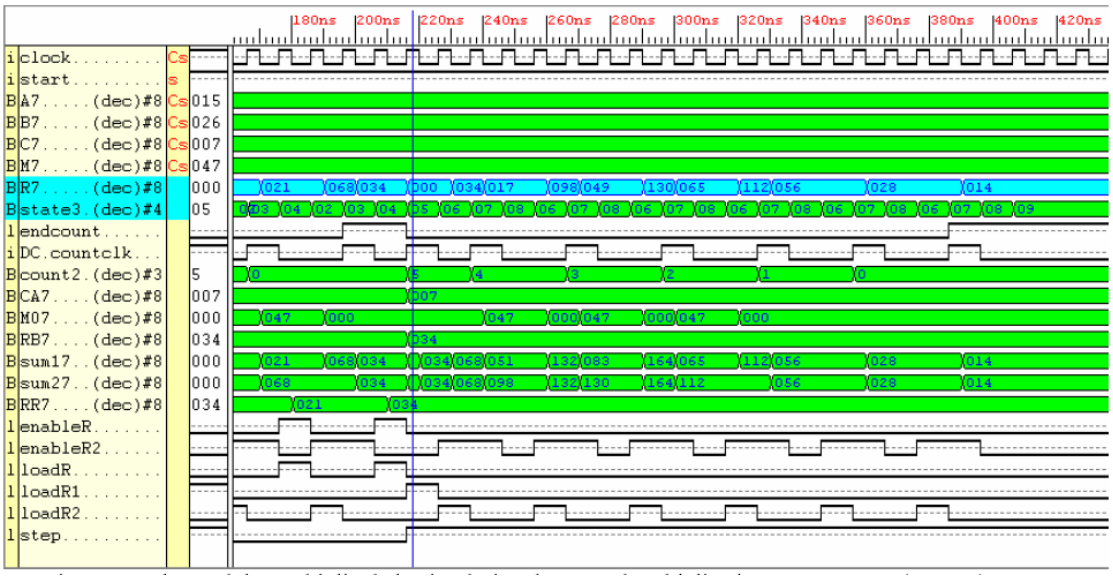


Figura 34 - SIMULACION MONTG 2

2.3.9 ALGORITMO SISTOLIC-MONTGOMERY

Existe una modificación en el algoritmo de Montgomery que sería una alternativa al método iterativo, el cual sería la systolic Montgomery. El algoritmo es el siguiente:

```
algorithm ModifiedMontgomery(A, B, M)
      int R := 0;
      1: for i := 0 to n-1
      2:    q_i := (r_0 + a_i×b_0) mod 2;
      3:    R  := (R + a_i×B + q_i×M) div 2;
      return R;
end ModifiedMontgomery.
```

A través de este algoritmo se puede generar el algoritmo systolic de Montgomery :

```
algorithm SystolicMontgomery(A,B,M,MB)
   int   R := 0;
   bit   carry := 0, x;
   0: for i := 0 to n
   1:    q_i := r_0^(i) ⊕ a_i.b_0;
   2:    for j := 0 to n
   3:       switch a_i, q_i
   4:          1,1: x := mb_i;
   5:          1,0: x := b_i;
   6:          0,1: x := m_i;
   7:          0,0: x := 0;
   8:       r_j^(i+1) := r_(j+1)^(i) ⊕ x_i ⊕ carry;
   9:    carry:=r_(j+1)^(i).x_i+r_(j+1)^(i).carry+x_i.carry;
   return R;
end SystolicMontgomery.
```

La arquitectura que da lugar a este algoritmo es el siguiente:

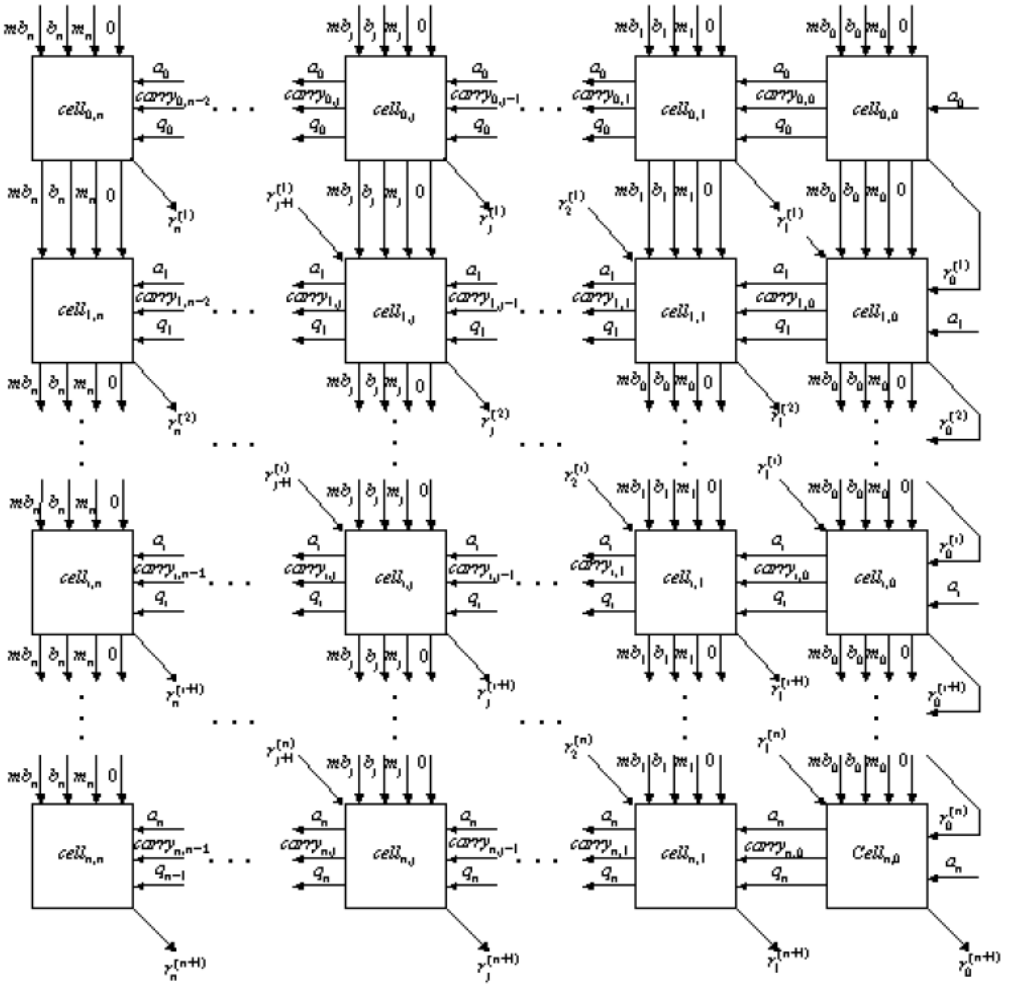


Figura 35 - ARQUITECTURA SISTOLIC

Cada celda se denomina como PE (procesing element) en los cuales se ejecuta la línea 8 del algoritmo.

Cada celda está compuesta como se muestra en la siguiente figura, que es la arquitectura básica de cada PE

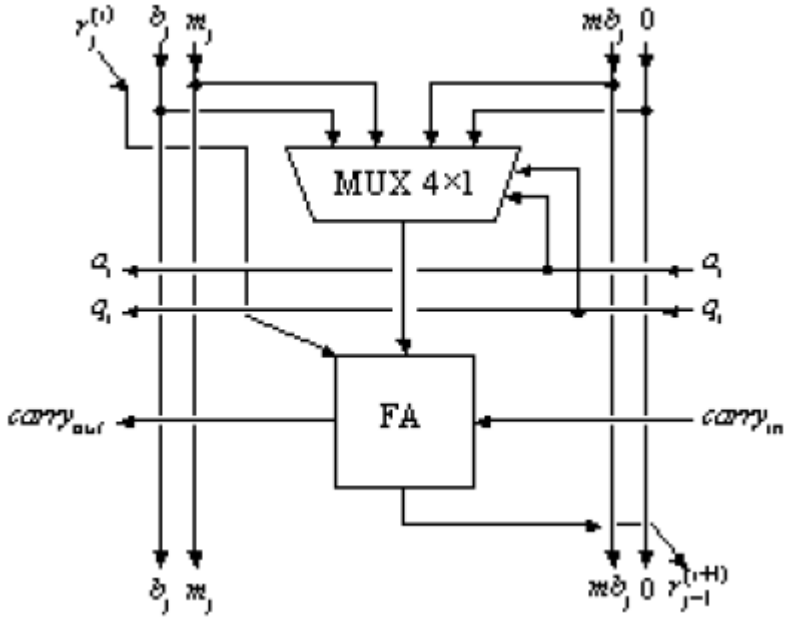


Figura 36 - CELDA PE

2.3.10 COMPARACIÓN SISTOLIC - ITERATIVO

En cuanto tiempo necesario y área requerida para cada método a continuación se muestra una tabla con los resultados obtenidos en una FPGA, en función del número de bits. IM (método iterativo), SM (systolic método)

operand size	*Area* (CLBs)		*clock cycle time* (ns)		*area×time*	
	IM	*SM*	*IM*	*SM*	*IM*	*SM*
128	89	259	46	23	4094	5957
256	124	304	102	42	12648	12767
512	209	492	199	76	41591	37392
768	335	578	207	82	69345	47396
1024	441	639	324	134	142884	85626

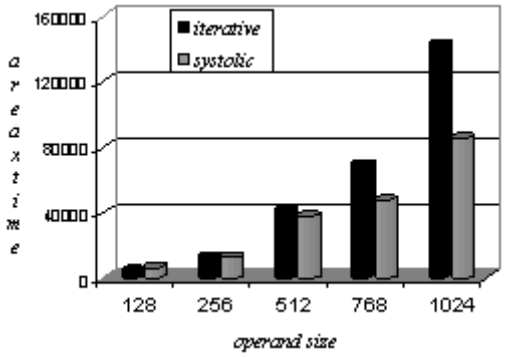


Figura 37 - GRAFICA 4

Como se puede observar en el gráfico a medida que el tamaño de los operandos es mayor sería mejor utilizar el método systolic que el iterativo. Por tanto y apoyándonos en los resultados en el caso de tener un tamaño de los operandos hasta 512 bits sería mejor utilizar el método iterativo que el factor tiempos x area sería más ventajoso, por el contrario si tuviéramos un tamaño de los operandos de mas de 1024 bits el mejor método a utilizar sería el de systolic. Teniendo en cuenta en los procesos criptográficos el tamaño de los operandos es bastante grande (mayor de 512 bits) en esta ocasión para poder sacar el mayor rendimiento al circuito se implementaría el método de systolic.

2.3.11 RESUMEN

En este apartado se han explicado varios métodos para el cálculo de multiplicación modular.

En un primer caso se ha desarrollado los métodos en los cuales se hace una multiplicación primero (Karatsuba-offman y el de Booth) y luego se ha hecho el modulo de esa multiplicación a través del método de Barret. En contraste con este método se ha descrito un algoritmo en el cual se hacía tanto la multiplicación como el módulo dentro del mismo método como es el algoritmo de Monrgomery. Este método en comparación con los del primer caso es más utilizado y además se puede observar en las simulaciones que para operandos de gran tamaño el factor tiempo x área es mejor que en los anteriores métodos. Estos algoritmos y resultados han sido obtenidos de varios documentos en los que la síntesis y la simulación se ha realizado sobre una fpga concreta, en los siguientes apartados lo que se va a realizar la implementación de estos algoritmos y además de estos otros algoritmos de modo que todos los podamos comparar en una misma fpga y de esta forma los resultados que obtengamos se pueda realizar una conclusión de cuál es la implementación más favorable.

IMPLEMENTACIÓN VHDL

[LENGUAJE VHDL]

INTRODUCCIÓN – IMPLEMENTACIÓN – PROGRAMAS UTILIZADOS – TEST BENCHZ

2.4 LENGUAJE VHDL

2.4.2 INTRODUCCIÓN VHDL

Durante el proyecto para la realización de las diferentes implementaciones se ha realizado a través del lenguaje vhdl.

A mediados de los años setenta se produce una fuerte evolución en los procesos de fabricación de los circuitos integrados, y junto a las tecnologías bipolares, surge la MOS (metal oxide semiconductor), principalmente la NMOS, promoviendo el desarrollo de circuitos digitales hasta la primera mitad de los años ochenta. En aquellas épocas, el esfuerzo de diseño se concentraba en los niveles eléctricos para establecer características e interconexiones entre los componentes básicos a nivel de transistor. El proceso de diseño era altamente manual y tan solo se empleaban herramientas como el PSPICE para simular esquemas eléctricos con modelos previamente personalizados a las distintas tecnologías. A medida que pasaban los años, los procesos tecnológicos se hacían más y más complejos. Los problemas de integración iban en aumento y los diseños eran cada vez más difíciles de depurar y de dar mantenimiento. Inicialmente los circuitos MSI (Medium Scale Integration) y LSI (Low Scale Integration) se diseñaron mediante la realización de prototipos basados en módulos más sencillos. Cada uno de estos módulos estaba formado por puertas ya probadas, este método poco a poco, iba quedándose obsoleto. En ese momento (finales de los años setenta) se constata el enorme desfase que existe entre tecnología y diseño. La considerable complejidad de los chips que se pueden fabricar, implica unos riesgos y costes de diseño desmesurados e imposibles de asumir por las empresas. Es entonces, cuando diversos grupos de investigadores empiezan a crear y desarrollar los llamados "lenguajes de descripción de hardware" cada uno con sus peculiaridades. Empresas tales como IBM con su IDL, el TI - HDL de Texas Instruments, ZEUS de General Electric, etc., así como los primeros prototipos empleados en las universidades, empezaron a desarrollarse buscando una solución a los problemas que presentaba el diseño de los sistemas complejos. Sin embargo, estos lenguajes nunca alcanzaron el nivel de difusión y consolidación necesarias por motivos distintos. Unos, los industriales, por ser propiedad de la empresa permanecieron encerrados en ellas y no estuvieron disponibles para su estandarización y mayor difusión, los otros, los universitarios, perecieron por no disponer de soporte ni mantenimiento adecuado. Alrededor de 1981 el Departamento de Defensa de los Estados Unidos desarrolla un proyecto llamado VHSIC (Very High Speed Integrated

Circuit) su objetivo era rentabilizar las inversiones en hardware haciendo más sencillo su mantenimiento. Se pretendía con ello resolver el problema de modificar el hardware diseñado en un proyecto para utilizarlo en otro, lo que no era posible hasta entonces porque no existía una herramienta adecuada que armonizase y normalizase dicha tarea, era el momento de los HDL's

En 1983, IBM, Intermetrics y Texas Instruments empezaron a trabajar en el desarrollo de un lenguaje de diseño que permitiera la estandarización, facilitando con ello, el mantenimiento de los diseños y la depuración de los algoritmos, para ello el IEEE propuso su estándar en 1984. Tras varias versiones llevadas a cabo con la colaboración de la industria y de las universidades, que constituyeron a posteriori etapas intermedias en el desarrollo del lenguaje, el IEEE publicó en diciembre de 1987 el estándar IEEE std 1076-1987 que constituyó el punto firme de partida de lo que después de cinco años sería ratificado como VHDL. Esta doble influencia, tanto de la empresa como de la universidad, hizo que el estándar asumido fuera un compromiso intermedio entre los lenguajes que ya habían desarrollado previamente los fabricantes, de manera que éste quedó como ensamblado y por consiguiente un tanto limitado en su facilidad de utilización haciendo dificultosa su total comprensión. Este hecho se ha visto incluso ahondado en su revisión de 1993. Pero esta deficiencia se ve altamente recompensada por la disponibilidad pública, y la seguridad que le otorga el verse revisada y sometida a mantenimiento por el IEEE. La independencia en la metodología de diseño, su capacidad descriptiva en múltiples dominios y niveles de abstracción, su versatilidad para la descripción de sistemas complejos, su posibilidad de reutilización y en definitiva la independencia de que goza con respecto de los fabricantes, han hecho que VHDL se convierta con el paso del tiempo en el lenguaje de descripción de hardware por excelencia.

2.4.3 PROGRAMAS UTILIZADOS

Los programas los cuales se ha usado en el proyecto para la realización del mismo son: ModelSim e ISE. El primero nos ha servido para la implementación del código vhdl y su posterior simulación. El segundo lo hemos utilizado para realizar la síntesis de los diferentes circuitos implementados y de esa manera poder comparar los diferentes algoritmos tanto en tiempo requerido como en área utilizada. Se ha elegido una FPGA para la síntesis del circuito SPARTAN 3 (XC3S50), se ha elegido esta FPGA debido a que es una de las cuales no son grandes pero tiene el suficiente tamaño para comparar de una manera clara los diferentes circuitos. Para la síntesis se ha eliminado la opción de bloque multiplicador el cual incorpora la FPGA, de este modo podremos comparar las diferentes implementaciones en puertas lógicas las cuales denominaremos LUT.

Una vez hecha la síntesis y la comprobación de que carece de errores se ha procedido a la post-síntesis, la cual nos va a permitir simular el circuito a través del ModelSim una vez sintetizado para comprobar que el circuito se comporta correctamente.
La realización del trabajo en el proyecto es el siguiente:

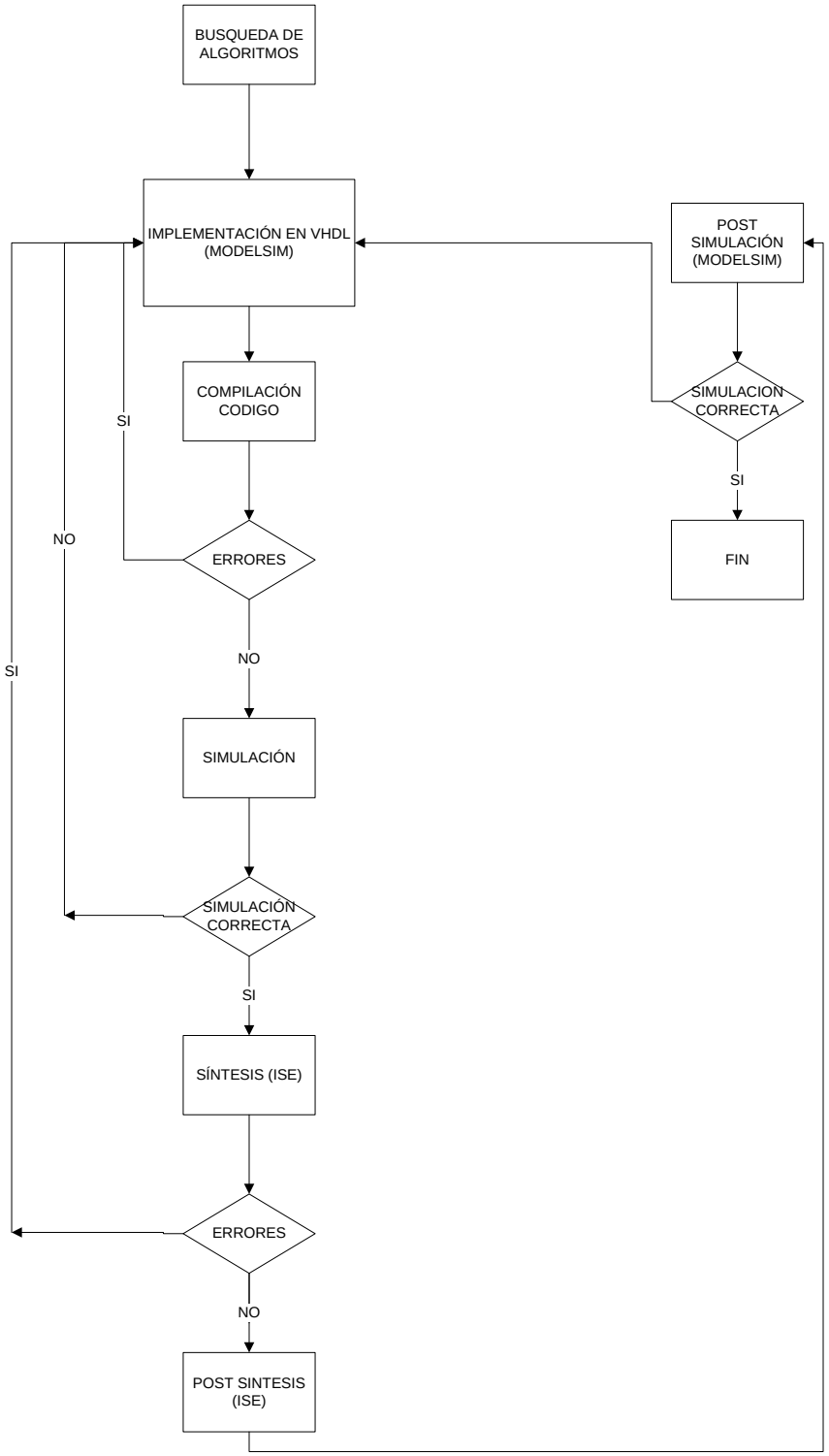


Figura 38 - D.ESTADOS PROYECTO

2.4.4 TEST-BENCH

Para la realización de la simulación de los diferentes códigos se he realizado un código estándar para todos en el cual se estimulan las señales de entrada con ciertos valores. Las señales las cuales se estimulan a través del test-bench son las señales de reloj, reset, inicio, y de las entradas A,B y M (este último en caso de que tenga esta entrada). El código se muestra a continuación, para probar diferentes implementaciones lo único que se ha hecho es cambiar el componente a instanciar.

Codigo=>

```
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
   use ieee.std_logic_arith.all;

entity test_CARD IS

end test_CARD;

architecture test_CARD_a of test_CARD IS

   COMPONENT mont_CARD
      GENERIC(N:INTEGER:=4);
      PORT(
         A,B: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
         M: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
         S: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
         INICIO: IN STD_LOGIC;
         FIN: OUT STD_LOGIC;
         CLK,RESET: IN STD_LOGIC
         );
      END COMPONENT;
```

```
 SIGNAL INICIO: STD_LOGIC;
      SIGNAL A,B: STD_LOGIC_VECTOR(3 DOWNTO 0);
      SIGNAL FIN:STD_LOGIC;
      SIGNAL S: STD_LOGIC_VECTOR(3 DOWNTO 0);
      SIGNAL M: STD_LOGIC_VECTOR(3 DOWNTO 0);
      SIGNAL CLK,RESET: STD_LOGIC;

      begin

         mi_test_CARD: MONT_CARD PORT MAP
(INICIO=>INICIO,A=>A,B=>B,S=>S,FIN=>FIN,M=>M,CLK=>CLK,RESET=>RESET);

         --SIMULACION DEL RESET
    PROCESS
      BEGIN
         reset <= '0';
         INICIO<='0';
         WAIT FOR 150 ns;
         reset <= '1';
         WAIT FOR 1000 ns;
         reset <= '0';
         WAIT FOR 10 us;
         INICIO<='1';
         WAIT;
    END PROCESS;

   -- SIMULACION CLK
    PROCESS
       BEGIN
          wait for 1 us;
          clk<='1';
          wait for 1 us;
          clk<='0';
       end process;
```

```
PROCESS
      BEGIN
         A<=CONV_STD_LOGIC_VECTOR(5,4);
         B<=CONV_STD_LOGIC_VECTOR(7,4);
         M<=CONV_STD_LOGIC_VECTOR(11,4);
         WAIT;
      END PROCESS;
   END TEST_CARD_A;
```

IMPLEMENTACIÓN DE ALGORITMOS

[MULTIPLICADOR COMBINACIONAL]

ALGORITMO – IMPLEMENTACIÓN VHDL – SIMULACIÓN – SÍNTESIS – RESULTADOS

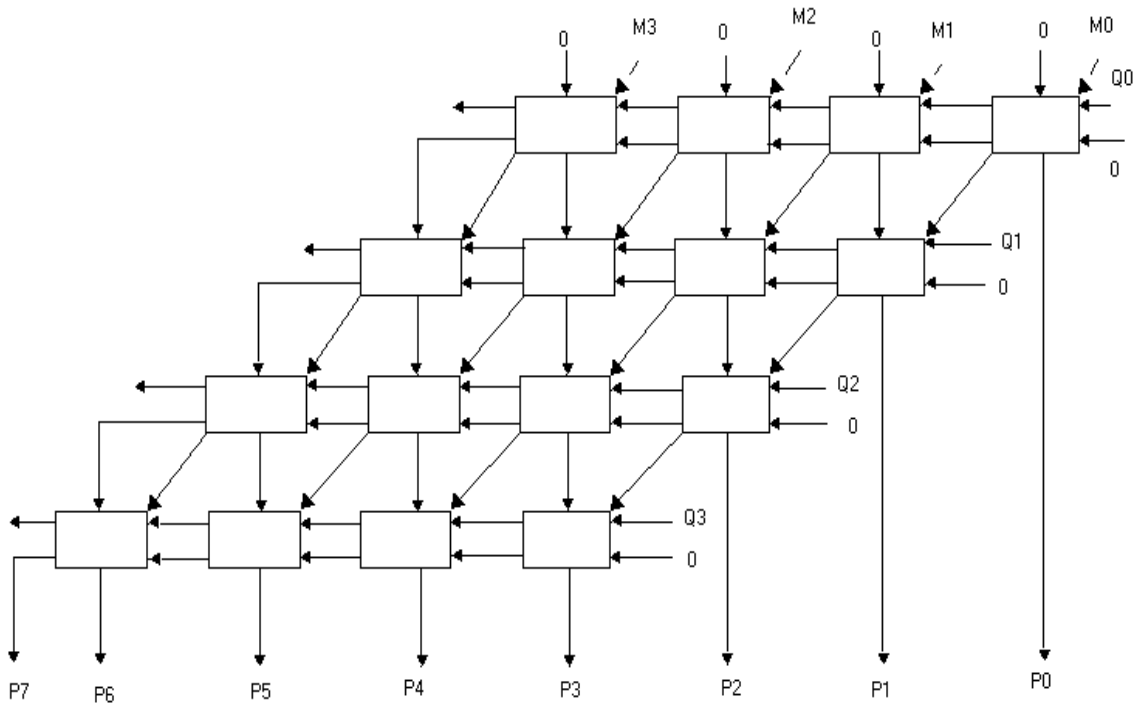
3. IMPLEMENTACIÓN DE ALGORITMOS

3.1 MULTIPLICACION COMBINACIONAL

3.1.2 ALGORITMO

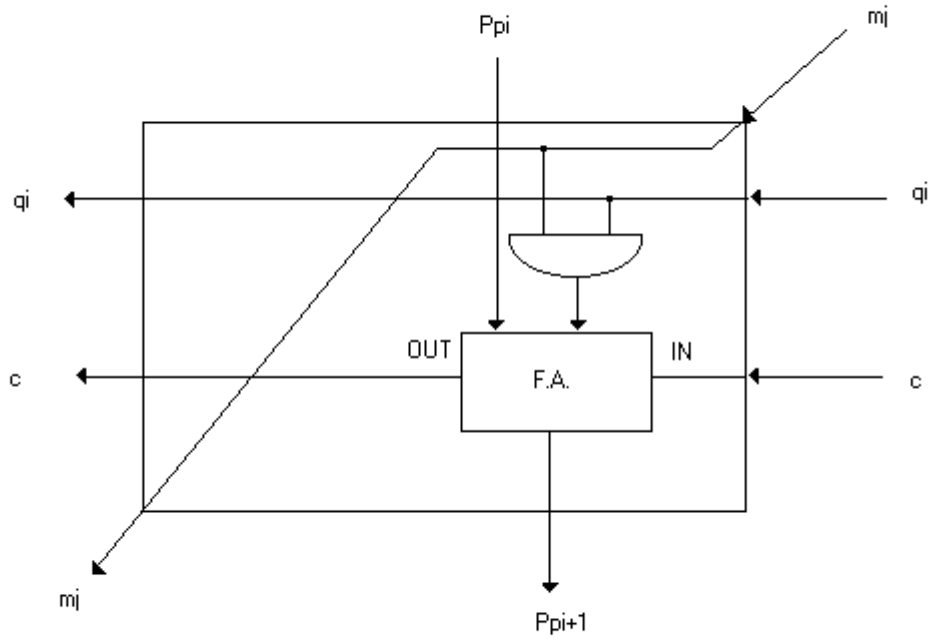
En esta multiplicación se realizará la operación de forma directa a través de un circuito totalmente combinacional. El método de multiplicación por sumas y desplazamientos tiene una implementación combinacional prácticamente directa.

En la figura se muestra un multiplicador para números binarios sin signo de 4 bits. La propia disposición de las celdas ya recuerda el proceso de la multiplicación, es decir, generación de productos parciales desplazados hacia la izquierda. Para poder llegar a comprender el funcionamiento de este circuito es necesario en primer lugar analizar el funcionamiento de cada celda básica.



La figura muestra la estructura interna de esa celda básica. Consta de un sumador completo (FA) y una puerta AND lógica. La puerta AND realiza el producto de cada bit

del multiplicador qj con el bit correspondiente del multiplicando mj generándose así un operando para el Sumador Completo. Los otros operandos son el bit correspondiente del producto parcial anterior y el acarreo generado en la etapa previa.



Si se observa el multiplicador de la figura, la primera fila de celdas elementales realiza la generación del primer producto parcial mediante la función lógica AND de q0, con los bits m0, m1, m2 y m3 del multiplicando. En la segunda fila se genera el segundo producto parcial de manera similar, ahora realizando la función lógica AND de q1, con cada uno de los bits de M; este segundo producto parcial (desplazado un lugar a la izquierda) se suma con el generado en la fila anterior. La tarea realizada en las filas tres y cuatro es similar al de las dos anteriores, en ellas se generan, respectivamente, los productos parciales relativos a multiplicar M por q2 y q3.

3.1.3 IMPLEMENTACION EN VHDL

Para la implementación de este circuito se realizará a través del sintetizador ISE, crearemos un programa vhdl en el cual lo único que se programará será la multiplicación a través de una línea de código en la cual aparecerá S<=A*B, una vez creado y compilado el código se realizará la síntesis la cual nos dará el circuito combinacional que queremos. En el sintetizador eliminamos la opción que nos da la fpga de utilizar su bloque multiplicador, de esa manera podemos ver el área que ocupa en número de LUT y poder compararlos con los demás algoritmos.

Código=>

```
PROCESS(CLK,RESET)
  BEGIN

    IF RESET='1' THEN
      S<=CONV_STD_LOGIC_VECTOR(0,2*N);
      FIN<='0';
     ELSIF CLK' EVENT AND CLK='1' THEN

        FIN<='1';
        S<= A*B ;

      END IF;
END PROCESS;
```

El diagrama de flujo se muestra a continuación:

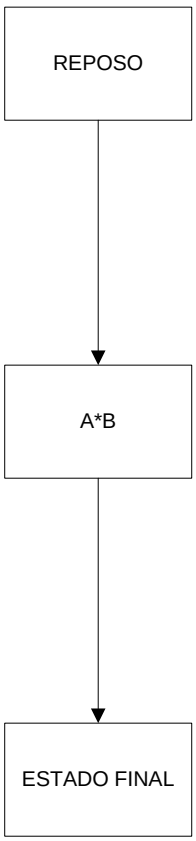


Figura 39 - D.ESTADOS COMBINACIONAL

3.1.4 SIMULACIÓN

En las siguientes simulaciones se muestra tanto en código binario como en decimal el cálculo del producto a través de esta implementación.

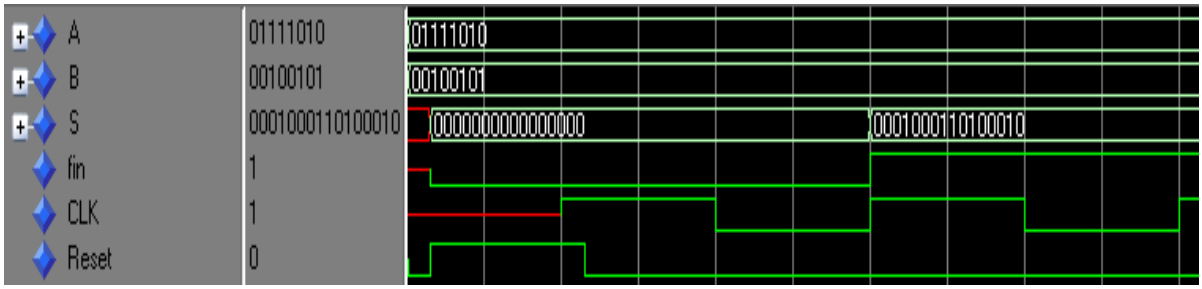


Figura 40 - SIMULACION COMB 1

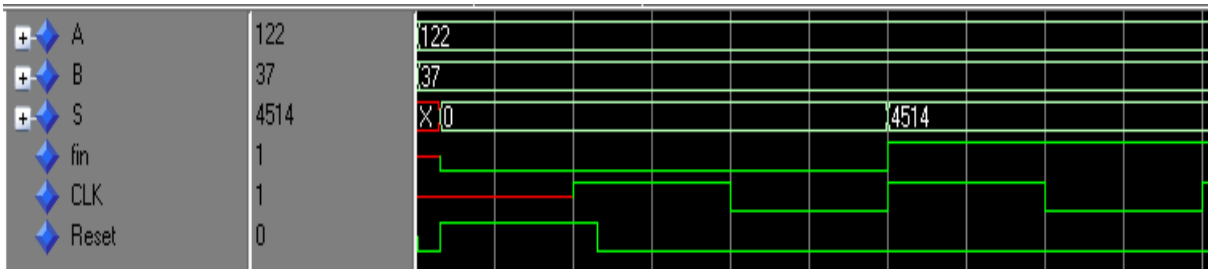


Figura 41 - SIMULACION COMB 2

Como se observa en las simulaciones para esta implementación el cálculo tarda 1 ciclo de reloj debido a que se basa en un circuito totalmente combinacional. Por tanto se puede concluir que para N bits esta implementación tarda 1 ciclo de reloj.

3.1.5 SINTESIS

Para la implementación se ha elegido un FPGA Spartan 3, la síntesis se ha realizado para 4,8,16,32,64,128,256,512 y 1024 bits. Los resultados para los distintos bits se compararán más adelante con los otros algoritmos implementados. El área se medirá en número de LUT de 4 variables, también se calculará la frecuencia máxima a la que puede ir cada implementación.

N=4

Frecuencia Máxima= 154 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	9	768	1%
Number of 4 input LUTs	15	1536	0%
Number of bonded IOBs	19	124	15%
Number of GCLKs	1	8	12%

N=8

Frecuencia Máxima = 105 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	37	768	4%
Number of 4 input LUTs	70	1536	4%
Number of bonded IOBs	35	124	28%
Number of GCLKs	1	8	12%

N=16

Frecuencia Máxima = 77 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	145	768	18%
Number of 4 input LUTs	284	1536	18%
Number of bonded IOBs	67	124	54%
Number of GCLKs	1	8	12%

N=32

Frecuencia Máxima = 55 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	559	768	72%
Number of 4 input LUTs	1109	1536	72%
Number of bonded IOBs	131	124	105%
Number of GCLKs	1	8	12%

N=64

FMAX= 41 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	2166	768	282%
Number of 4 input LUTs	4317	1536	281%
Number of bonded IOBs	259	124	208%
Number of GCLKs	1	8	12%

N=128

Frecuencia Máxima = 35 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	8471	768	1102%
Number of 4 input LUTs	16916	1536	1101%
Number of bonded IOBs	515	124	415%
Number of GCLKs	1	8	12%

N=256

Frecuencia Máxima = 30 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	33405	768	4349%
Number of 4 input LUTs	66762	1536	4346%
Number of bonded IOBs	1027	124	828%
Number of GCLKs	1	8	12%

N=512

Frecuencia Máxima = 20 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	167025	768	21748%
Number of 4 input LUTs	268120	1536	17455%
Number of bonded IOBs	4157	124	3352%
Number of GCLKs	1	8	12%

N=1024

Frecuencia Máxima = 10 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	657400	768	86992%
Number of 4 input LUTs	989000	1536	4534%
Number of bonded IOBs	16700	124	13400%
Number of GCLKs	1	8	12%

3.1.6 RESULTADOS CIRCUITO COMBINACIONAL

A continuación se muestra las tablas en las cuales se observa el área ocupada (LUT), la frecuencia máxima y el número de ciclos de reloj que tarda el circuito.

Nº BITS	ÁREA (LUT)	FMAX (MHZ)	CICLOS DE RELOJ
4	15	154	1
8	70	105	1
16	284	77	1
32	1109	55	1
64	4317	41	1
128	16916	35	1
256	66762	30	1
512	268120	20	1
1024	989000	12	1

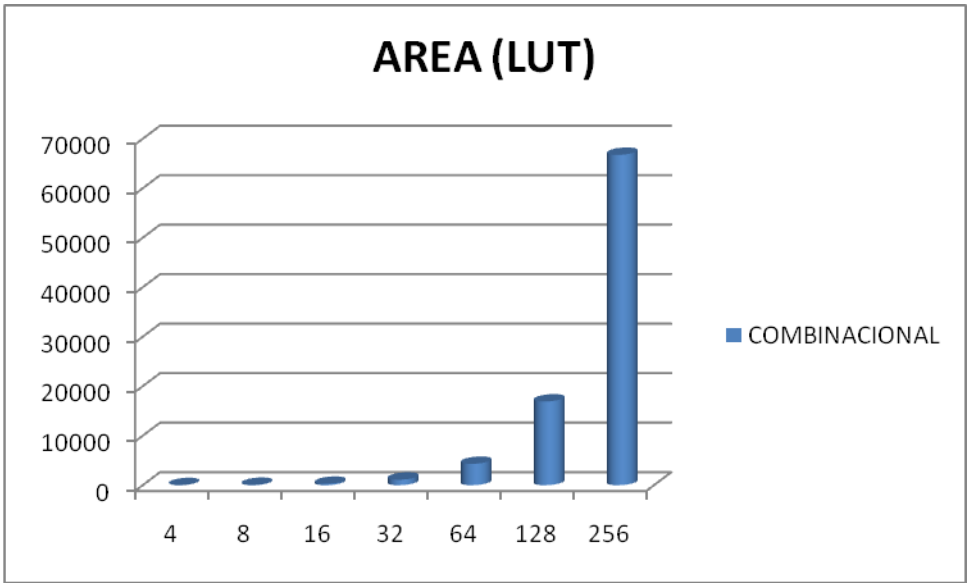


Figura 42 - AREA COMBINACIONAL

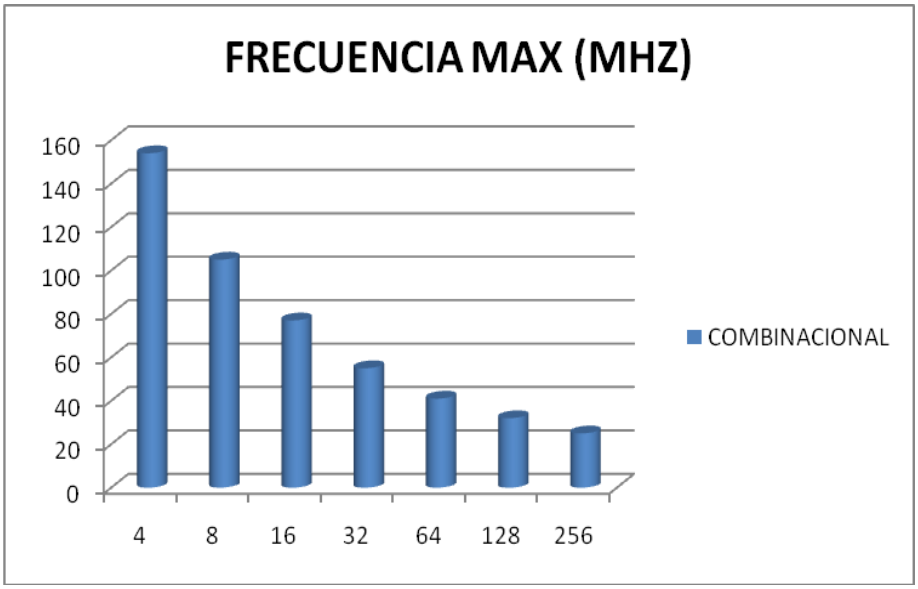


Figura 43 - FRECUENCIA COMBINACIONAL

IMPLEMENTACIÓN DE ALGORITMOS

[MULTIPLICADOR CLÁSICO]

ALGORITMO – IMPLEMENTACIÓN VHDL – SIMULACIÓN – SINTESIS - RESULTADOS

3.2 MULTIPLICADOR CLASICO

3.2.2 ALGORITMO

El siguiente circuito que se ha implementado es el del multiplicador clásico.

A continuación se explica la multiplicación para dos números de N bits.

a) Tomamos como A,B y P como el primer operando, segundo operando y producto total respectivamente.
b) Se comparará los bits de B, en caso de que el bits que se esté comparando sea 0 guardaremos una fila de 0, en caso de que sea 1 se guardará el operando A en el registro.
c) A medida de que se vayan comparando los diferentes bits del operando B, lo que guardemos tras la comparación lo iremos almacenando en un registro de desplazamiento y se irá sumando con un registro de acumulación.
d) Cuando se hallan comparado todos los bits del operando B se habrá terminado con el algoritmo y en el registro acumulador tendremos el resultado total del producto.

En la siguiente figura se muestra de una forma general como se haría el algoritmo:

$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & b_0a_3 & b_0a_2 & b_0a_1 & b_0a_0 \\
 & & & b_1a_3 & b_1a_2 & b_1a_1 & b_1a_0 & \\
 & & b_2a_3 & b_2a_2 & b_2a_1 & b_2a_0 & & \\
 & b_3a_3 & b_3a_2 & b_3a_1 & b_3a_0 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

Figura 44 - ALGORITMO CLASICO

Ejemplo: Tomemos 2 numero de 4 bits, los cuales A=3=0011 y B=4=0100 P=A*B

				0	0	1	1	3
			X	0	1	0	0	4
				0	1	0	0	
			0	1	0	0		
		0	0	0	0			
	0	0	0	0				
0	0	0	1	1	0	0	12	

Como se observa el producto total es 1100 que es el número 12 en decimal.
En la multiplicación para el resultado final necesitaremos el doble de espacio que lo que ocupe los operandos, es decir que para operandos de N bits el resultado P=A*B lo necesitaremos guardar en un registro de 2*N.

3.2.3 IMPLEMENTACION EN VHDL

Para la implementación se ha utilizado en la entidad se ha utilizado dos puertos de entrada para los numero que queremos multiplicar, y otros dos puertos que serán la señal de reloj y el reset. Para las salidas tenemos el puerto de salida y el puerto FIN que indicará que ha terminado la multiplicación. Además se empleará un puerto Inicio el cual dará comienzo el algoritmo.

A continuación se muestra el bloque que dará lugar a la implementación, para números A y B de 16 bits.

Codigo de la entidad =>

```
        GENERIC(N:INTEGER:=16);
PORT(
   A,B: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
   S: OUT STD_LOGIC_VECTOR((2*N-1) DOWNTO 0);
   FIN: OUT STD_LOGIC;
   CLK,RESET: IN STD_LOGIC;
   INICIO: STD_LOGIC   );
```

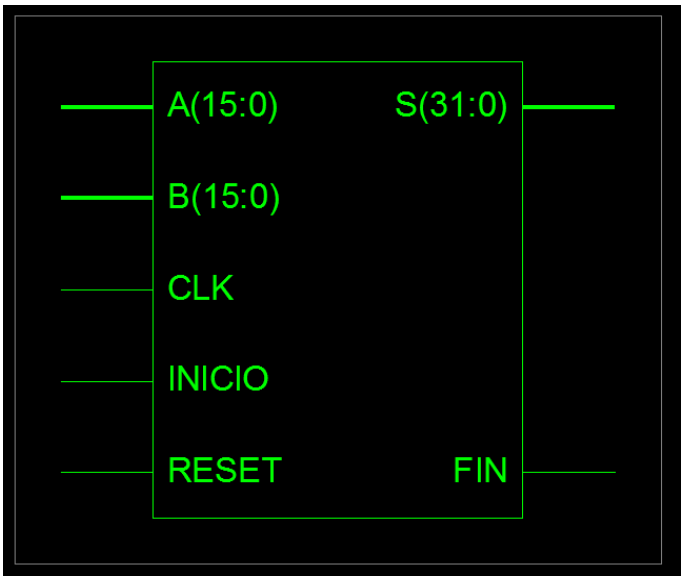


Figura 45 - ENTIDAD CLASICO

Nota: Este tipo de entidad será la utilizada en todos los algoritmos implementados.

Para la arquitectura se ha utilizado una maquina de estados, un estado de reposo y otro en el cual se hará la multiplicación. Dentro de este estado se ha utilizado un contador para ir contando el número de bits que se va comparando del operando y en función de su valor se carga en el registro de desplazamiento con ceros, o con el otro operando.

Código =>

```
            ELSIF ACTUAL=MUL THEN
        IF B(CONTADOR)='1' THEN

          IF CONTADOR=0  THEN   --SI ESTAMOS EN LA PRIMERA POSICION COPIAMOS EL DATO A
            G:=CONV_STD_LOGIC_VECTOR(0,N)&A;
            DESPLAZAMIENTO2:=CONV_STD_LOGIC_VECTOR(0,N)&A;


            ELSE        --SE VA DESPLAZANDO EL REGISTRO DE DESPLAZAMIENTO 1 POSICION
            DESPLAZAMIENTO2:=DESPLAZAMIENTO(2*N-2 DOWNTO 0) & '0'  ;
            G:=G+DESPLAZAMIENTO2;

          END IF;
```

```
                ELSE      -- EN EL CASO DE QUE EL BIT DE B SEA 0 NO SUMA Y LO UNICO QUE HACE ES
DESPLAZAR 1 POSICION
            IF CONTADOR=0 THEN
              DESPLAZAMIENTO2:=CONV_STD_LOGIC_VECTOR(0,N)&A;
              G:=G;


             ELSE DESPLAZAMIENTO2:=DESPLAZAMIENTO(2*N-2 DOWNTO 0) & '0'  ;
              G:=G;


            END IF;
          END IF;
        END IF;
```

En el registro desplazamiento se guardará el resultado de comparar el bit del operando B, en función del bits que estemos comparando se desplazará dentro del registro hacia la izquierda para luego sumar y guardar el resultado final de todas las comparaciones en el registro Final. En el registro de desplazamiento se irá guardando el operando A desplazado una vez a la izquierda, de modo que cada vez que se quiere comparar el bit de B , al copiar el operando ya se habrá realizado el desplazamiento.

El diagrama de flujo es el siguiente:

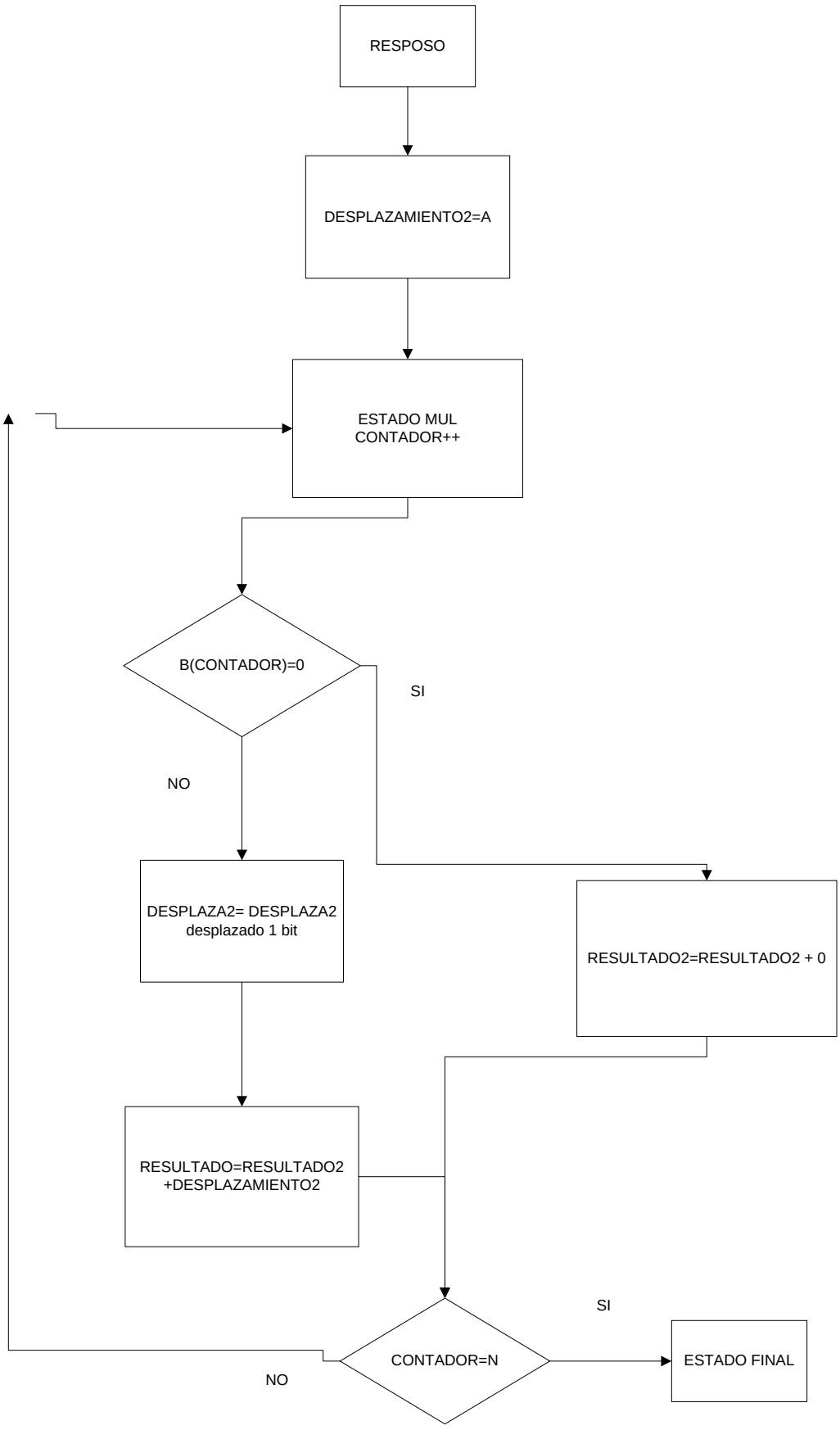


Figura 46 - D.ESTADOS CLASICO

3.2.4 SIMULACIÓN

A continuación se muestra la simulación realizada con el programa ModelSIM para dos números cualquiera de N bits.

Se ha tomado el ejemplo para A= 400 y B=555, los cuales serán de 16 bits por lo tanto el resultado final se guarda en el registro Final que será de 32 bits.

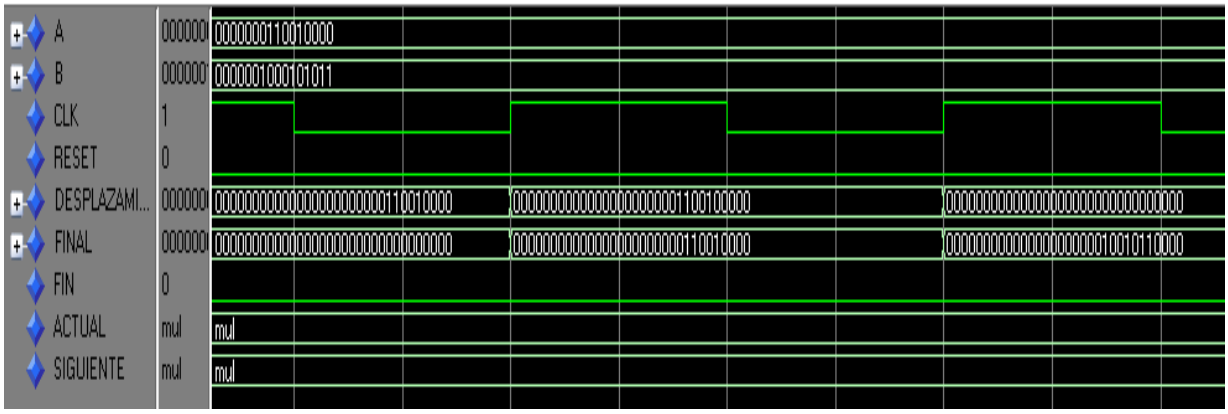


Figura 47 - SIMULACION CLASICO 1

En esta parte de la simulación se observa como en el registro desplazamiento se va guardando el valor en función de si el bits de B es 1 o 0, en cada ciclo de reloj se desplaza una posición hacia la izquierda. Además en el registro FINAL se va guardando el resultado de ir sumando los distintos valores que nos da en el registro Desplazamiento.

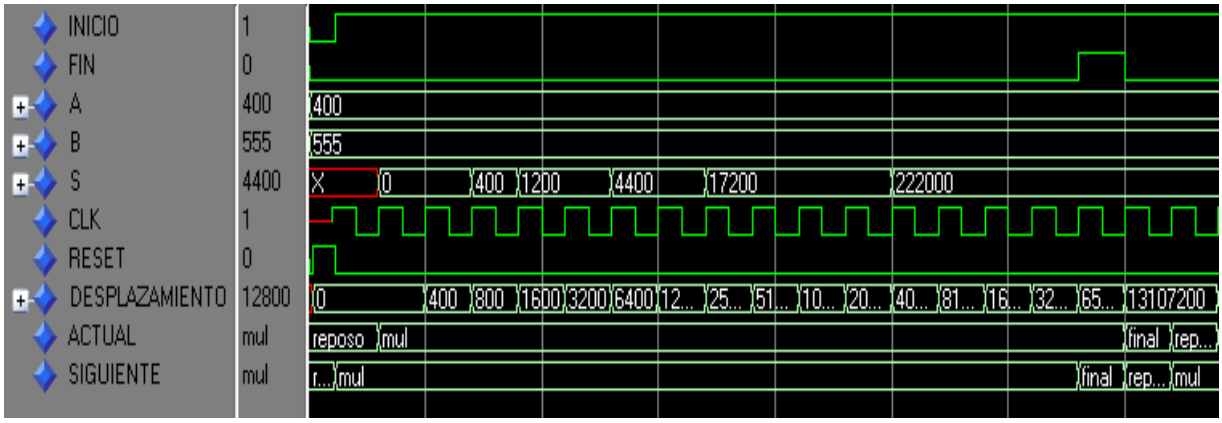


Figura 48 - SIMULACION CLASICO 2

En esta figura se muestra la simulación completa para la multiplicación. Cuando se calcula el resultado final el bit Fin se pone a 1 indicando que ha acabado la multiplicación. Como se puede observar ha tardado 16 ciclos de reloj, pues se necesita comparar cada bit del operando B, por tanto se puede afirmar que para este algoritmo de multiplicación clásica para operandos de N bits el diseño implementado tardará N ciclos de reloj.

3.2.5 SINTESIS

Para la síntesis del circuito se ha utilizado el programa ISE. A través de este programa podremos calcular el área utilizada de modo que podamos compararlo con los otros tipos de multiplicación. Para la síntesis se ha tomado 4,8,16,32,64,128,256,512 y 1024 bits. También se ha calculado la frecuencia máxima a la cual puede ir el reloj de modo que asegure un funcionamiento óptimo del circuito.

N=4

Frecuencia Máxima: 180MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	23	768	2%
Number of Slice Flip Flops	26	1536	1%
Number of 4 input LUTs	39	1536	2%
Number of bonded IOBs	20	124	16%
Number of GCLKs	1	8	12%

N=8

F.MAX=150 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	45	768	5%
Number of Slice Flip Flops	51	1536	3%
Number of 4 input LUTs	78	1536	5%
Number of bonded IOBs	36	124	29%
Number of GCLKs	1	8	12%

N=16

Frecuencia Máxima =145MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	88	768	11%
Number of Slice Flip Flops	100	1536	6%
Number of 4 input LUTs	133	1536	8%
Number of bonded IOBs	68	124	54%
Number of GCLKs	1	8	12%

N=32

Frecuencia Máxima =122MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	178	768	23%
Number of Slice Flip Flops	198	1536	12%
Number of 4 input LUTs	261	1536	16%
Number of bonded IOBs	132	124	106%
Number of GCLKs	1	8	12%

N=64

Frecuencia Máxima =85MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	361	768	47%
Number of Slice Flip Flops	390	1536	25%
Number of 4 input LUTs	499	1536	32%
Number of bonded IOBs	260	124	209%
Number of GCLKs	1	8	12%

N=128

Frecuencia Máxima =55MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	726	768	94%
Number of Slice Flip Flops	775	1536	50%
Number of 4 input LUTs	982	1536	63%
Number of bonded IOBs	516	124	416%
Number of GCLKs	1	8	12%

N=256

Frecuencia Máxima =30MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1453	768	189%
Number of Slice Flip Flops	1545	1536	100%
Number of 4 input LUTs	1952	1536	127%
Number of bonded IOBs	1028	124	829%
Number of GCLKs	1	8	12%

N=512

Frecuencia Máxima = 16.271MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	3418	768	445%
Number of Slice Flip Flops	3086	1536	200%
Number of 4 input LUTs	4914	1536	319%
Number of bonded IOBs	2052	124	1654%
Number of GCLKs	1	8	12%

N=1024

Frecuencia Máxima = 8.454MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	6827	768	888%
Number of Slice Flip Flops	6164	1536	401%
Number of 4 input LUTs	8773	1536	571%
Number of bonded IOBs	4100	124	3306%
Number of GCLKs	1	8	12%

3.2.6 RESULTADOS ALGORITMO CLÁSICO

En las siguientes tablas se muestra los resultados de este algoritmo en cuanto al área que ocupa (número de LUT), frecuencia máxima y número de ciclos que tarda el algoritmo.

Nº BITS	ÁREA (LUT)	FMAX (MHZ)	CICLOS DE RELOJ
4	39	180	4
8	78	150	8
16	133	145	16
32	261	122	32
64	499	85	64
128	982	55	128
256	1952	30	256
512	4914	16	512
1024	8773	8	1024

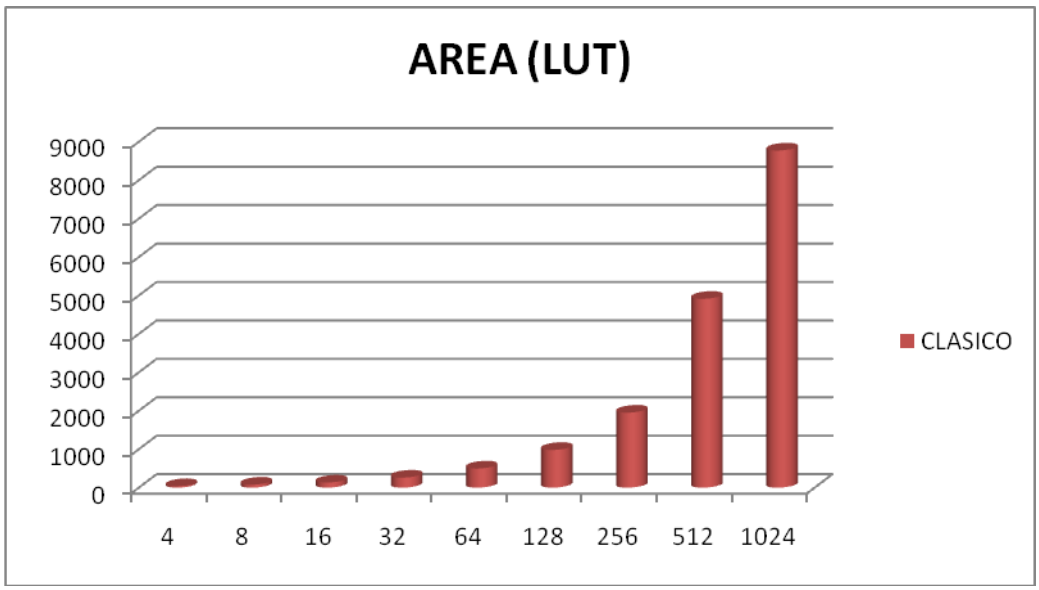


Figura 49 - AREA CLASICO

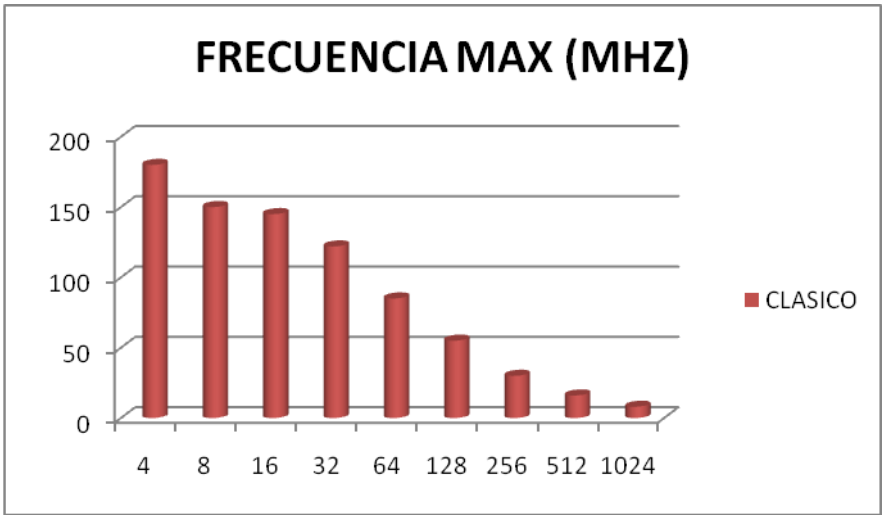


Figura 50 - FRECUENCIA CLASICO

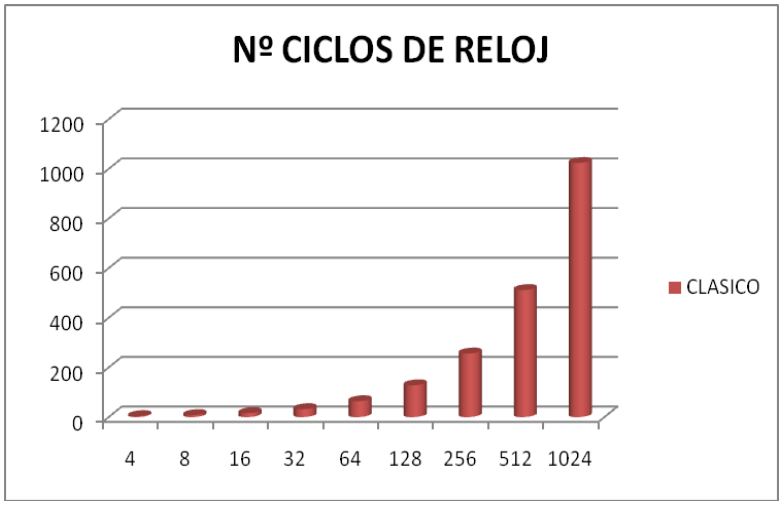


Figura 51 - CICLOS CLASICO

IMPLEMENTACIÓN DE ALGORITMOS

[MULTIPLICADOR KARATSUBA-OFMAN]

ALGORITMO – IMPLEMENTACIÓN VHDL – SIMULACIÓN – SINTESIS - RESULTADOS

3.3 KARATSUBA-OFMAN

3.3.2 ALGORITMO

A través de este algoritmo se implementará en código vhdl el circuito resultante. El algoritmo a partir del cual se implementara el código es el siguiente:

$$\begin{aligned} P &= XY \\ &= (X_H 2^n + X_L)(Y_H 2^n + Y_L) \\ &= 2^{2n}(X_H Y_H) + 2^n(X_H Y_L + X_L Y_H) + X_L Y_L \end{aligned}$$

Siendo X e Y dos numero enteros de N bits. En el algoritmo N que será el número de bits será igual a 2n.

Lo primero que se hará será realizar un ejemplo de cómo funciona el algoritmo.
Se tomará X=5= 0101 Y= 4=0100

a) $X_H = 01$; $Y_H = 01$; $X_L = 01$; $Y_L = 00$
b) $X_H * Y_H = 01 * 01 = 01 = A$
c) $X_H * Y_L = 01 * 00 = 00 = C$
d) $X_L * Y_H = 01 * 01 = 01 = B$
e) $X_L * Y_L = 01 * 00 = 00 = D$
f) $2^4 = 16 = 10000 = E$
g) $2^2 = 4 = 0100 = F$
h) $E * A = 16 = 10000$
i) $F * (B + C) = 4 = 0100$
j) $P = 10101010 = 20$

OPERACIÓN	RESULTADO DECIMAL	RESULTADO BINARIO	VARIABLE
$X_H * Y_H$	1	0 0 0 1	A
$X_H * Y_L$	1	0 0 0 1	B
$X_L * Y_H$	0	0 0 0 0	C
$X_L * Y_L$	0	0 0 0 0	D
2^4	16	1 0 0 0 0	E
2^2	4	0 1 0 0	F
E * A	16	1 0 0 0 0	EA
F * (B + C)	4	0 1 0 0	FBC
EA + FBC + D	20	1 0 1 0 0	TOTAL

Como resumen del algoritmo podemos concluir que este algoritmo se basa en el multiplicador clásico pero realiza pequeñas multiplicaciones en paralelo reduciendo así el tiempo de cálculo pero aumentando el área ocupada. Se puede obtener como conclusión que este algoritmo es una mezcla de circuito combinacional y circuito secuencial ya expuesto anteriormente.

3.3.3 IMPLEMENTACIÓN

Para poder implementar el algoritmo se ha divido este en varias partes:

a) La primera parte en la cual obtendremos las partes altas y bajas de los registros X e Y
b) La segunda parte en la cual realizaremos las multiplicaciones de lo que está dentro de los paréntesis y además la del último término (Término D de la tabla).
c) En el cuarto paso realizamos la suma de la variable B y C.
d) En la tercera parte realizaremos las multiplicaciones de 2^x. Esta multiplicación se realizará desplazando X veces los bits hacia los bits más significativos.
e) Por último sumamos todos los términos que nos queden de modo que el resultado sea la multiplicación total.

Para la multiplicación que hay que realizar en el segundo paso se ha utilizado la implementación realizada en el apartado anterior ya expuesto (Multiplicación clásica).

Como se requieren cuatro multiplicaciones se harán las cuatro en paralelo, de modo que se tarde lo menos posible en hacer la multiplicación, por el contrario se utilizará más área. En este caso esas multiplicaciones se realizan sobre partes altas y bajas y no sobre

el total de todos los bits, de esta manera se consigue que la multiplicación dure N/2 ciclos de reloj ya que multiplica N/2 bits. En el siguiente código se muestra como se ha implementado la parte de la multiplicación.

```
--MULTIPLICACION DE LA PRIMERA PARTE XHYH

            IF CONTADOR2<N/2 AND B_HIGH(CONTADOR2)='1'  THEN

                  IF CONTADOR2=0 THEN
                  FINAL_E:=CONV_STD_LOGIC_VECTOR(0,N/2) & A_HIGH;
                  DESPLAZAMIENTO:=CONV_STD_LOGIC_VECTOR(0,N/2) & A_HIGH;
                  FINAL<=desplazamiento;


                   ELSE DESPLAZAMIENTO:=DESPLAZAMIENTO(N-2 DOWNTO 0) & '0';
                        FINAL_E:=FINAL_E+DESPLAZAMIENTO;
                        FINAL<=FINAL_E;

                    END IF;

                 ELSE
                 IF CONTADOR2=0 THEN
                    DESPLAZAMIENTO:=CONV_STD_LOGIC_VECTOR(0,N/2)&A_HIGH;
                    FINAL_E:=FINAL_E;
                    FINAL<=FINAL_E;
                   --  G<=DESPLAZAMIENTO3;
                    ELSE
                      DESPLAZAMIENTO:=DESPLAZAMIENTO(N-2 DOWNTO 0) & '0';
                      FINAL_E:=FINAL_E;
                      FINAL<=FINAL_E;
                     --  G<=DESPLAZAMIENTO3;
                  END IF;
              END IF;


            -- MULTIPLICACION DE LA SEGUNDA PARTE XHYL
            IF CONTADOR2<N/2 AND B_LOW(CONTADOR2)='1' THEN

                  IF CONTADOR2=0 THEN
                   FINAL_E2:=CONV_STD_LOGIC_VECTOR(0,N/2) & A_HIGH;
                   DESPLAZAMIENTO2:=CONV_STD_LOGIC_VECTOR(0,N/2) & A_HIGH;
                   FINAL2<=DESPLAZAMIENTO2;
                   g<=desplazamiento2;

                    ELSE DESPLAZAMIENTO2:=DESPLAZAMIENTO2(N-2 DOWNTO 0) & '0';
                        FINAL_E2:=FINAL_E2+DESPLAZAMIENTO2;
                        FINAL2<=FINAL_E2;
                        g<=desplazamiento2;
                     END IF;

                 ELSE
                 IF CONTADOR2=0 THEN
                    DESPLAZAMIENTO2:=CONV_STD_LOGIC_VECTOR(0,N/2)&A_HIGH;
                    FINAL_E2:=FINAL_E2;
                    FINAL2<=FINAL_E2;
                    G<=DESPLAZAMIENTO2;
                    ELSE
                      DESPLAZAMIENTO2:=DESPLAZAMIENTO2(N-2 DOWNTO 0) & '0';
                      FINAL_E2:=FINAL_E2;
                      FINAL2<=FINAL_E2;
                      G<=DESPLAZAMIENTO2;
                  END IF;
              END IF;


            -- MULTIPLICACION DE LA TERCERA PARTE XLYH
            IF CONTADOR2<N/2 AND B_HIGH(CONTADOR2)='1' THEN
```

```
IF CONTADOR2=0 THEN
                    FINAL_E3:=CONV_STD_LOGIC_VECTOR(0,N/2) & A_LOW;
                    DESPLAZAMIENTO3:=CONV_STD_LOGIC_VECTOR(0,N/2) & A_LOW;
                    FINAL3<=DESPLAZAMIENTO3;
                   -- G<=DESPLAZAMIENTO3;

                     ELSE DESPLAZAMIENTO3:=DESPLAZAMIENTO3(N-2 DOWNTO 0) & '0';
                          FINAL_E3:=FINAL_E3+DESPLAZAMIENTO3;
                          FINAL3<=FINAL_E3;
                       --    G<=DESPLAZAMIENTO3;
                       END IF;

                   ELSE
                     IF CONTADOR2=0 THEN
                      DESPLAZAMIENTO3:=CONV_STD_LOGIC_VECTOR(0,N/2)&A_LOW;
                      FINAL_E3:=FINAL_E3;
                      FINAL3<=FINAL_E3;
                     --  G<=DESPLAZAMIENTO3;
                      ELSE
                        DESPLAZAMIENTO3:=DESPLAZAMIENTO3(N-2 DOWNTO 0) & '0';
                        FINAL_E3:=FINAL_E3;
                        FINAL3<=FINAL_E3;
                       --  G<=DESPLAZAMIENTO3;
                      END IF;
                  END IF;


              --MULTIPLICACION DE LA CUARTA PARTE XLYL
                   IF CONTADOR2<N/2 AND B_LOW(CONTADOR2)='1' THEN

                    IF CONTADOR2=0 THEN
                     FINAL_E4:=CONV_STD_LOGIC_VECTOR(0,N/2) & A_LOW;
                     DESPLAZAMIENTO4:=CONV_STD_LOGIC_VECTOR(0,N/2) & A_LOW;
                     FINAL4<=DESPLAZAMIENTO4;

                      ELSE DESPLAZAMIENTO4:=DESPLAZAMIENTO4(N-2 DOWNTO 0) & '0';
                           FINAL_E4:=FINAL_E4+DESPLAZAMIENTO3;
                           FINAL4<=FINAL_E4;
                       END IF;

                   ELSE
                   IF CONTADOR2=0 THEN
                      DESPLAZAMIENTO4:=CONV_STD_LOGIC_VECTOR(0,N/2)&A_LOW;
                      FINAL_E4:=FINAL_E4;
                      FINAL4<=FINAL_E4;
                      --G<=DESPLAZAMIENTO3;
                     ELSE
                       DESPLAZAMIENTO4:=DESPLAZAMIENTO4(N-2 DOWNTO 0) & '0';
                       FINAL_E4:=FINAL_E4;
                       FINAL4<=FINAL_E4;
                      -- G<=DESPLAZAMIENTO3;
                     END IF;
                  END IF;
```

En los registro Desplazamiento (X) se irá guardando el resultado de comparar el valor del bits de B, si es 0 se carga el registro con ceros, en caso contrario se guarda el valor de A desplazado hacia los bits más significativos en función del bits de B que se esté comparando. En los registros Final (X) se irá guardando las sumas de los valores que se de en Desplazamiento (X).

Para la multiplicación de 2^{x} simplemente se cogerá los desplazará X posiciones hacia los bits más significativos del operando que se quiera multiplicar. En el siguiente código se muestra la implementación en VHDL.

```
RESULTADO1<= FINAL(N-1 DOWNTO 0) & C(N-1 DOWNTO 0);
            RESULTADO2 <= C(N-1 DOWNTO N/2) & FINAL5(N-1 DOWNTO 0) & C(N-1 DOWNTO N/2);
            RESULTADO3<=C(N-1 DOWNTO 0) & FINAL4(N-1 DOWNTO 0);
```

El registro C es un registro de N bits de ceros.

Se ha utilizado una maquina de estados con 8 estados, además se necesitará varios registros de 2n bits además de el registro para el resultado total que será de 2n bits. Es decir que para operando de N bits necesitaremos un registro de 2 * N bits para almacenar el resultado.

El diagrama de flujo se muestra en la siguiente figura.

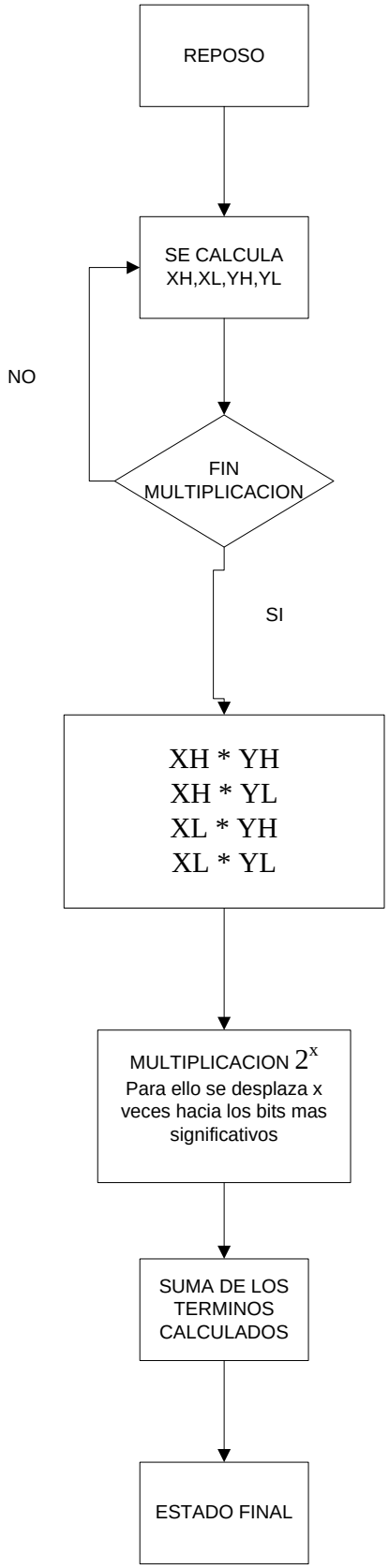


Figura 52 - D.ESTADOS KARATSUBA

3.3.4 SIMULACIÓN

A continuación se va a realizar la simulación de dos productos a través del programa ModelSim.

El primer ejemplo que simularemos será P = XY = 27 * 42. Siendo X e Y números de 8 bits.

En la siguiente parte de la simulación se muestra como se hace la multiplicación para el resultado de hacer X_L * Y_L = 1011 * 1010 . En la simulación se ha representado solo los registros de Final 4 y Desplazamiento 4, para los otros registros el funcionamiento es igual aunque cambian los valores. Se observa que en el registro desplazamiento 4 se va guardando el resultado de comparar el bits de B, en función del bits que se esté comparando se va desplazando hacia la izquierda como ocurre en la Multiplicación Clásica, el registro Final4 es el registro Acumulador, en el cual se irá sumando todos los valores de desplazamiento4. Como ya se ha explicado anteriormente las demás multiplicaciones se harán de la misma manera de forma paralela.

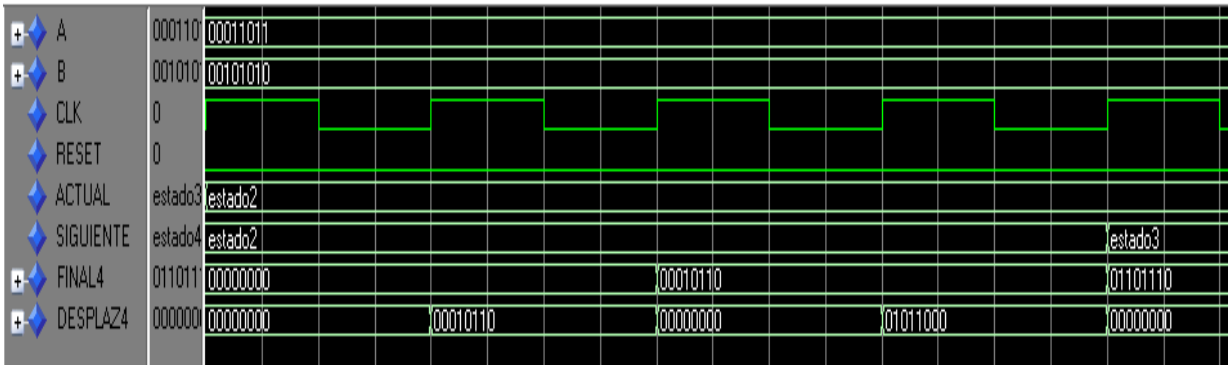


Figura 53 - SIMULACION KARATSUBA 1

En la siguiente simulación se muestra la simulación completa de la multiplicación.

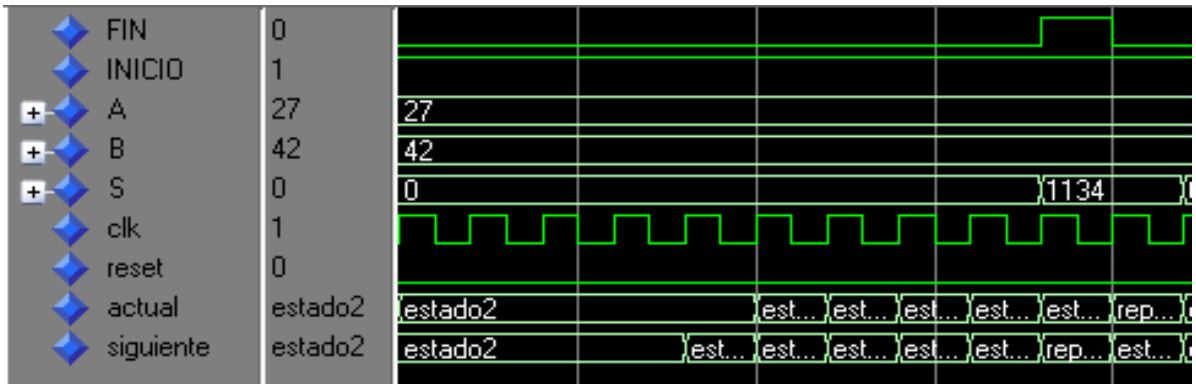


Figura 54 - SIMULACION KARATSUBA 2

En la siguiente simulación se calcula el producto de X= 255 e Y =120 , para estos operandos se utilizarán 16 bits, por tanto para el resultado se necesitará un registro de 32 bits.

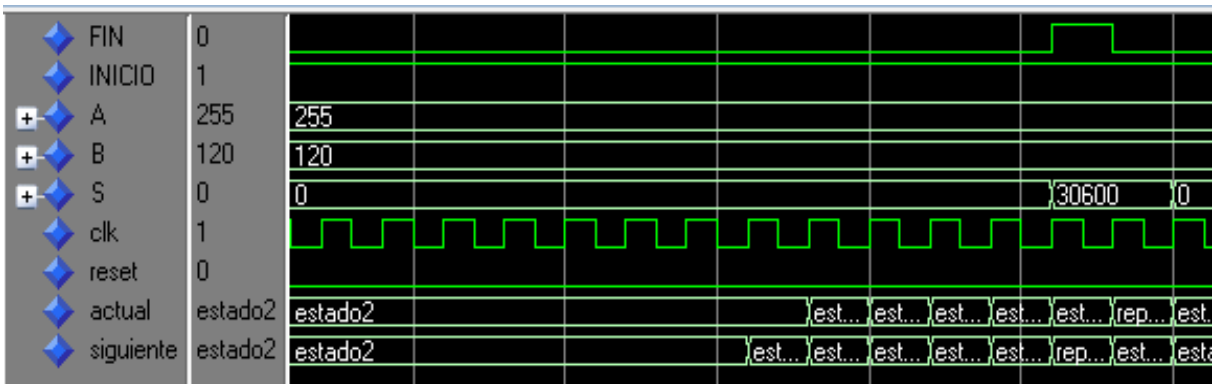


Figura 55 - SIMULACION KARATSUBA 3

Como se observa el resultado es de 30600. Para este caso de 16 bits la multiplicación ha tardado 8 ciclos de reloj, por tanto este algoritmo tarda N/2 ciclos de reloj para operandos de N bits.

3.3.5 SINTESIS

El resultado de la síntesis ha sido el siguiente, para las mismas condiciones que en los algoritmos anteriores.

N=4
Frecuencia Máxima =177MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	103	768	13%
Number of Slice Flip Flops	91	1536	5%
Number of 4 input LUTs	194	1536	12%
Number of bonded IOBs	20	124	16%
Number of GCLKs	1	8	12%

N=8
Frecuencia Máxima =131MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	190	768	24%
Number of Slice Flip Flops	176	1536	11%
Number of 4 input LUTs	358	1536	23%
Number of bonded IOBs	36	124	29%
Number of GCLKs	1	8	12%

N=16
Frecuencia Máxima =119MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	352	768	45%
Number of Slice Flip Flops	344	1536	22%
Number of 4 input LUTs	662	1536	43%
Number of bonded IOBs	68	124	54%
Number of GCLKs	1	8	12%

N=32
Frecuencia Máxima =105MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	712	768	92%
Number of Slice Flip Flops	685	1536	44%
Number of 4 input LUTs	1341	1536	87%
Number of bonded IOBs	132	124	106%
Number of GCLKs	1	8	12%

N=64
Frecuencia Máxima = 93MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1360	768	177%
Number of Slice Flip Flops	1357	1536	88%
Number of 4 input LUTs	2554	1536	166%
Number of bonded IOBs	260	124	209%
Number of GCLKs	1	8	12%

N=128
Frecuencia Máxima =65 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	2618	768	340%
Number of Slice Flip Flops	2700	1536	175%
Number of 4 input LUTs	4778	1536	311%
Number of bonded IOBs	516	124	416%
Number of GCLKs	1	8	12%

N=256
Frecuencia Máxima =38MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	5344	768	695%
Number of Slice Flip Flops	5393	1536	351%
Number of 4 input LUTs	10044	1536	653%
Number of bonded IOBs	1028	124	829%
Number of GCLKs	1	8	12%

N= 512
Frecuencia Máxima = 21.183MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	13152	768	1712%
Number of Slice Flip Flops	10808	1536	703%
Number of 4 input LUTs	24206	1536	1575%
Number of bonded IOBs	2052	124	1654%
Number of GCLKs	1	8	12%

N= 1024
Frecuencia Máxima = 11.132MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	26158	768	3405%
Number of Slice Flip Flops	21612	1536	1407%
Number of 4 input LUTs	48792	1536	3176%
Number of bonded IOBs	4100	124	3306%
Number of GCLKs	1	8	12%

3.3.6 RESULTADOS KARATSUBA

A continuación se muestra el resultado del área que ocupa (lut), frecuencia máxima y número de ciclos de reloj para distintos números de bits.

Nº BITS	ÁREA (LUT)	FMAX (MHZ)	CICLOS DE RELOJ
4	194	177	2
8	358	131	4
16	662	119	8
32	1341	105	16
64	2554	93	32
128	4778	65	64
256	8230	43	128
512	24206	21	256
1024	48792	11	512

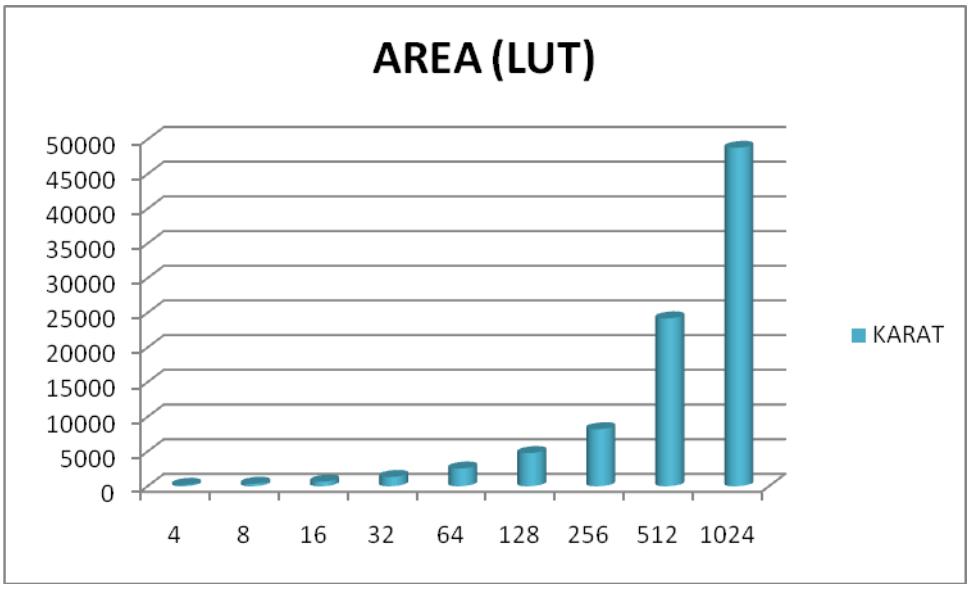


Figura 56 - AREA KARATSUBA

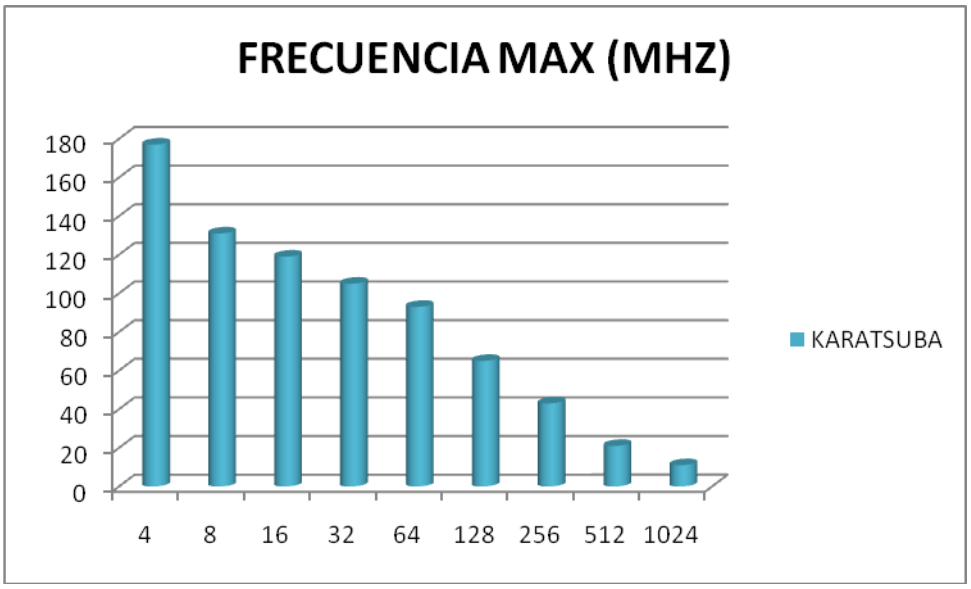


Figura 57 - FRECUENCIA KARATSUBA

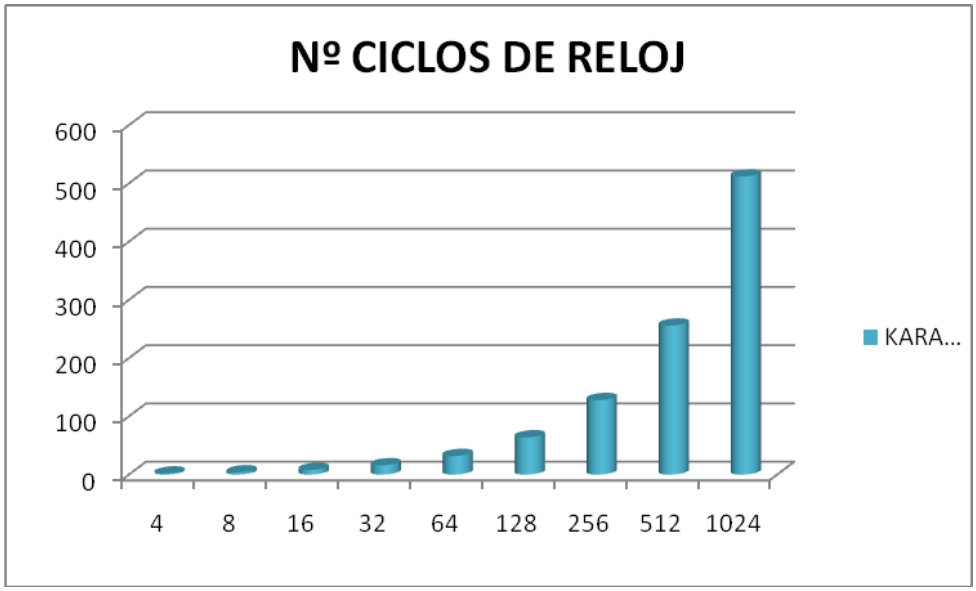


Figura 58 - CICLOS KARATSUBA

IMPLEMENTACIÓN DE ALGORITMOS

[REDUCCIÓN BARRET]

BARRET 1 Y BARRET2 => ALGORITMO – IMPLEMENTACIÓN VHDL – SIMULACIÓN – SINTESIS – RESULTADOS

3.4 ALGORITMOS DE REDUCCIÓN

ALGORITMOS

Una vez que se ha realizado la multiplicación se pasará a realizar el modulo de la multiplicación. Al realizar el módulo sobre el producto se obtiene el resto de la división entre el producto y el módulo.

Ejemplo: Para un producto= 122 , y Modulo=M=37 ; PmodM=>

122	37	
111		3 cociente
11 resto		

En este caso el resto sería 11.

Para realizar esta operación se va a utilizar el método de Barret, se van a realizar dos algoritmos los cuales denominaremos Barret1 y Barret2.

El algoritmo utilizado para barret1 será el siguiente:

```
Algorithm NaiveReduction(P, M)
   Int R := P;
   Do R := R - M;
   While R > 0;
   If R ≠ 0 Then R := R + M;
   Return R;
End NaiveReduction
```

P será el producto de los números A,B; M es el módulo por el cual queremos realizar la operación. Simplemente se basa en sucesivas restas hasta que el resto sea menor que 0.

El algoritmo utilizado para barret2 será el siguiente:

```
Algorithm RestoringReduction(P, M)
   Int R0 := P;
   Int N := LeftShift(M, n);
   For i = 1 To n Do
      Ri := Ri-1 - N;
      If R < 0 Then Ri := Ri-1;
      N := RightShift(N);
   Return Ri;
End RestoringReduction
```

La idea principal de este algoritmo se basa en realizar restas más grandes para llegar antes a que el resto sea menor que 0.

En estos dos casos el producto será del doble de tamaño que M, debido a que el producto es el resultado de multiplicar dos operandos de N bits y dará un resultado de 2*N bits.

3.4.2 BARRET1

3.4.2.2 IMPLEMENTACIÓN EN VHDL BARRET1

Para la implementación se necesita un registro para almacenar el producto, otro para almacenar el módulo y un tercer registro para obtener el resultado de N bits, siendo N el número de bits de los operando A y B.

Se ha implementado el algoritmo de NaiveReduction aunque incluyendo una modificación. Debido a que estamos considerando números positivos no se puede comparar el resto R con 0 ya que nunca será negativo, lo que se ha hecho es comparar R con M de modo que si el resto es menor que M finaliza el algoritmo y obtenemos el resto. Esta modificación tiene la desventaja de que realiza una comparación más en cada proceso, una solución a esto podría ser con la representación de números negativos, de modo que uno de los bits lo utilizamos para ver si el número es negativo de esta forma se podría utilizar la comparación con 0 y nos ahorraríamos una comparación, por el contrario en desventaja con la modificación que se ha hecho esta utilizaría un bit más

(para poder ver el signo). En este caso no se ha representado números negativos con lo que se utiliza una comparación más.

En la siguiente parte del código se observa como se ha obtenido el resto =>

```
IF ACTUAL=REPOSO THEN
                        R<=A;
                        M<=B;

                    ELSIF ACTUAL=ESTADO1 THEN   --A R SE VA RESTANDO M HASTA QUE R ES MENOR QUE R
                     R<=RESTO-(CONV_STD_LOGIC_VECTOR(0,N)&M);
                     M<=B;

                    ELSIF ACTUAL=ESTADO2 OR ACTUAL=ESTADO_FIN THEN
                     R<=R;
                     M<=B;
```

Esta implementación aunque es sencilla de calcular es bastante ineficiente ya que necesita como máximo 2^n ciclos de reloj siendo N el número de bits de los operandos.
El diagrama de flujo es el siguiente:

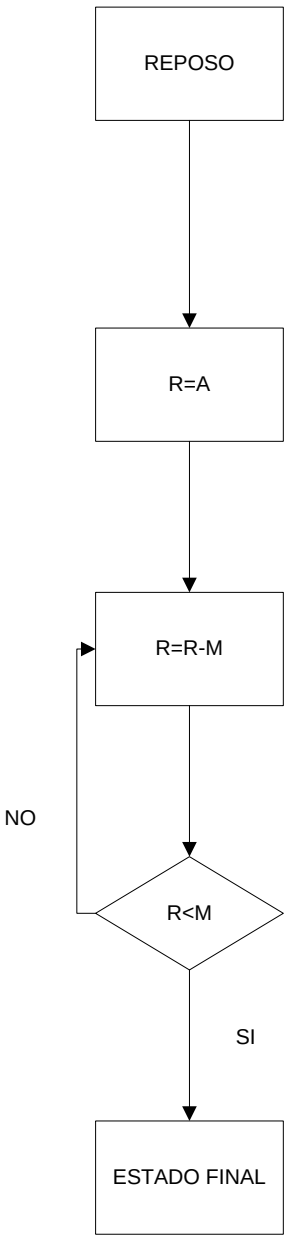


Figura 59 - D. ESTADOS BARRET1

3.4.2.3 SIMULACIÓN

En la siguiente simulación se muestra el resultado de hacer el módulo de la siguiente operación => P mod M; siendo P=122 , M=37

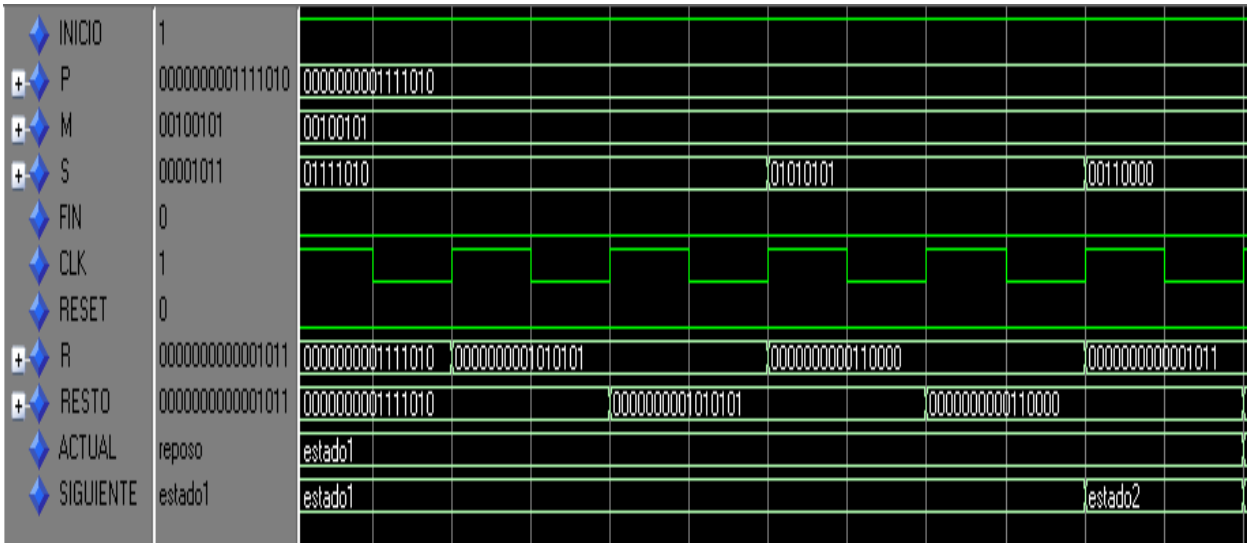


Figura 60 - SIMULACION BARRET1 1

Se observa como en R se acumula la operación de restar Resto – M, finalmente se almacenará en Resto el resultado final.

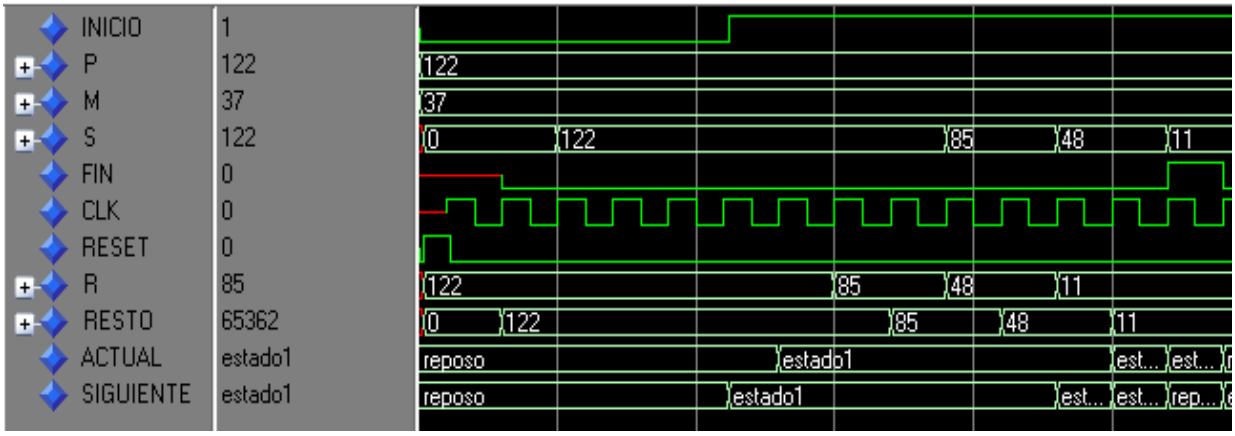


Figura 61 - SIMULACION BARRET1 2

3.4.2.4 SÍNTESIS

Se realiza la síntesis obteniéndose los siguientes resultados:

N=4
Frecuencia Máxima = 196MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	28	768	3%
Number of Slice Flip Flops	27	1536	1%
Number of 4 input LUTs	50	1536	3%
Number of bonded IOBs	20	124	16%
Number of GCLKs	1	8	12%

N=8
Frecuencia Máxima = 180MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	52	768	6%
Number of Slice Flip Flops	51	1536	3%
Number of 4 input LUTs	95	1536	6%
Number of bonded IOBs	36	124	29%
Number of GCLKs	1	8	12%

N=16
Frecuencia Máxima = 155MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	102	768	13%
Number of Slice Flip Flops	99	1536	6%
Number of 4 input LUTs	185	1536	12%
Number of bonded IOBs	68	124	54%
Number of GCLKs	1	8	12%

N=32
Frecuencia Máxima = 121MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	201	768	26%
Number of Slice Flip Flops	195	1536	12%
Number of 4 input LUTs	365	1536	23%
Number of bonded IOBs	132	124	106%
Number of GCLKs	1	8	12%

N=64
Frecuencia Máxima = 85MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	401	768	52%
Number of Slice Flip Flops	387	1536	25%
Number of 4 input LUTs	725	1536	47%
Number of bonded IOBs	260	124	209%
Number of GCLKs	1	8	12%

N=128
Frecuencia Máxima = 53MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	799	768	104%
Number of Slice Flip Flops	771	1536	50%
Number of 4 input LUTs	1445	1536	94%
Number of bonded IOBs	516	124	416%
Number of GCLKs	1	8	12%

N=256
Frecuencia Máxima = 30MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1596	768	207%
Number of Slice Flip Flops	1539	1536	100%
Number of 4 input LUTs	2887	1536	187%
Number of bonded IOBs	1028	124	829%
Number of GCLKs	1	8	12%

N=512
Frecuencia Máxima = 16.265MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	3190	768	415%
Number of Slice Flip Flops	3075	1536	200%
Number of 4 input LUTs	5770	1536	375%
Number of bonded IOBs	2052	124	1654%
Number of GCLKs	1	8	12%

N=1024
Frecuencia Máxima = 8.452MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	6378	768	830%
Number of Slice Flip Flops	6147	1536	400%
Number of 4 input LUTs	11534	1536	750%
Number of bonded IOBs	4100	124	3306%
Number of GCLKs	1	8	12%

3.4.2.5 RESULTADOS BARRET1

En la siguiente tabla se muestra el resultado de la implementación de este algoritmo. Se ha calculado el área ocupada, frecuencia máxima del reloj, y números de ciclos.

Nº BITS	ÁREA (LUT)	FMAX (MHZ)	CICLOS DE RELOJ
4	50	196	16
8	95	180	256
16	185	155	65536
32	365	121	alto
64	725	85	alto
128	1445	53	alto
256	2887	30	alto
512	5770	17	alto
1024	11534	9	alto

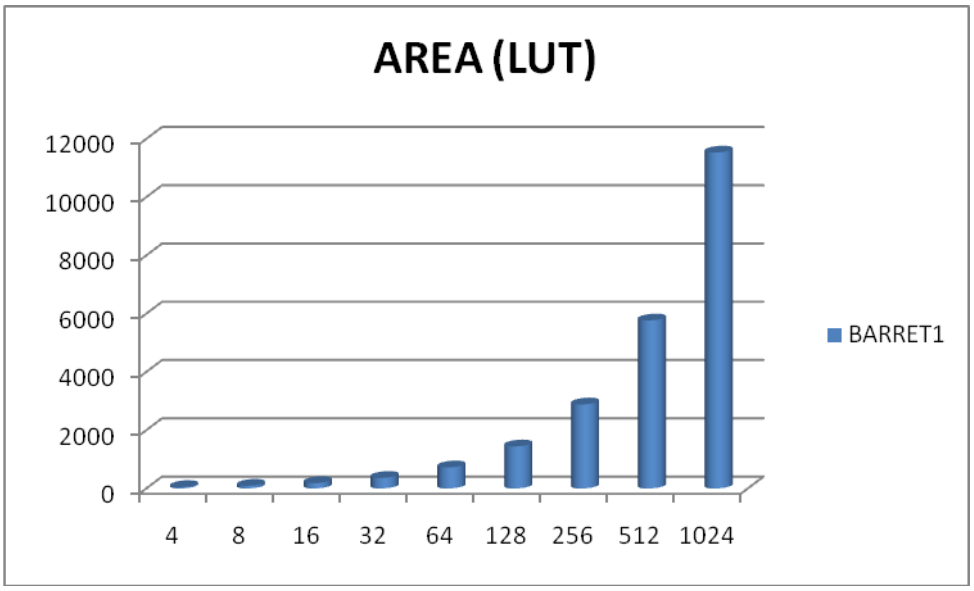


Figura 62 - AREA BARRET1

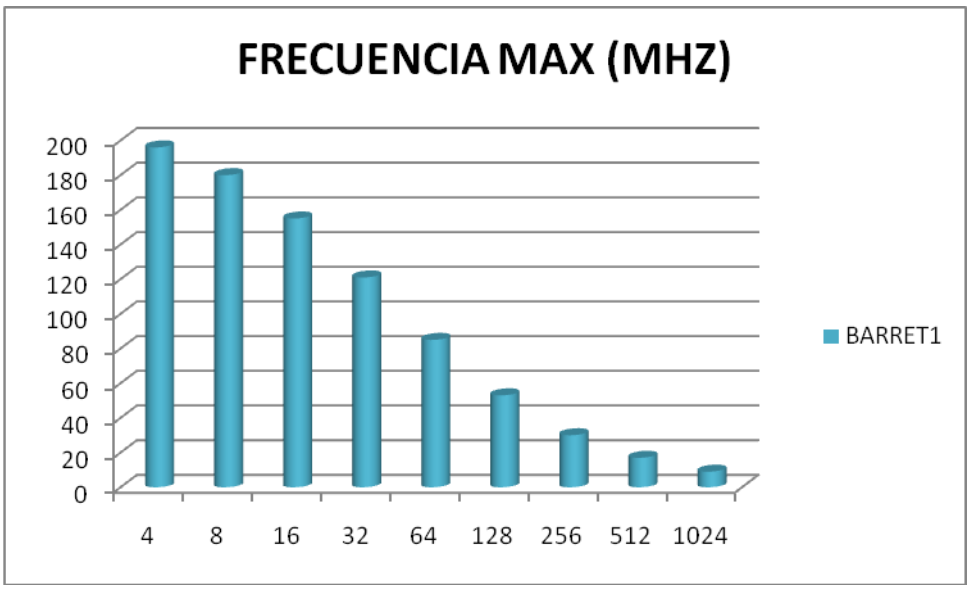


Figura 63 - FRECUENCIA BARRET1

3.4.3. BARRET 2

3.4.3.2 IMPLEMENTACIÓN EN VHDL

Esta implementación se basa en convertir el módulo en número mayor de modo que la restas que se realizan hace que se llegue antes al resto que queremos obtener.

La implementación del algoritmo se muestra en la siguiente parte del código del programa.

Código=>

```
if actual=estado1 then
        -- SI R0 ES MAYOR QUE K SERA POSITIVO EL RESTO
          IF R0>K THEN
            R1<=R0-k;
            R0<=R1;

        -- EN CASO CONTRARIO SERA NEGATIVO POR LO QUE K SE DIVIDE POR 2 PARA IR APROXIMANDO
          ELSE
            K<='0' & K(2*n-1 downto 1);
            R1<=R0;
          END IF;
      end if;

       if actual=estado2 or actual=reposo then
          R0<=A;
          K<= B & CONV_STD_LOGIC_VECTOR(0,N);
       end if;
```

Se observa que antes de restar k se ha desplazado hacia la izquierda, primero se compara si R0>k en caso afirmativo se realizará (R0 – k), en caso contrario se desplaza un bit hacia bits menos significativos (dividir por 2) de modo que se va aproximando el valor del resultado final.

Para esta implementación a diferencia del barret 1 tenemos un contador que va desde 0 hasta N bits. Por tanto con este algoritmo para obtener el resto se tardará N ciclos de reloj siendo N el número de bits de los operandos.

El diagrama de flujo es el siguiente:

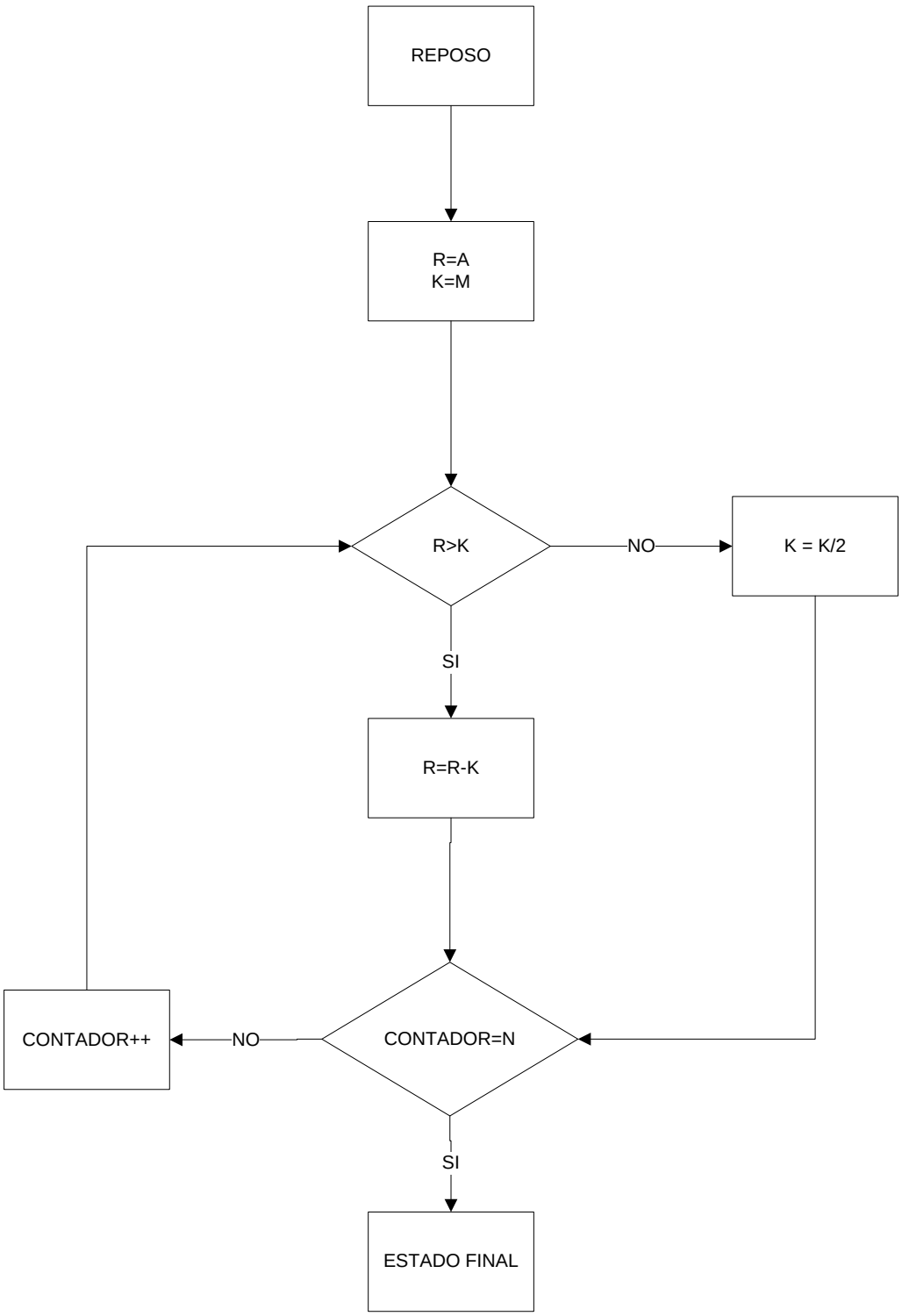


Figura 64 - D.ESTADOS BARRET2

3.4.3.3 SIMULACIÓN

Los datos simulados son:

Producto: A = 122 Modulo: B = 37 Resultado = 11

*Primera parte de la simulación

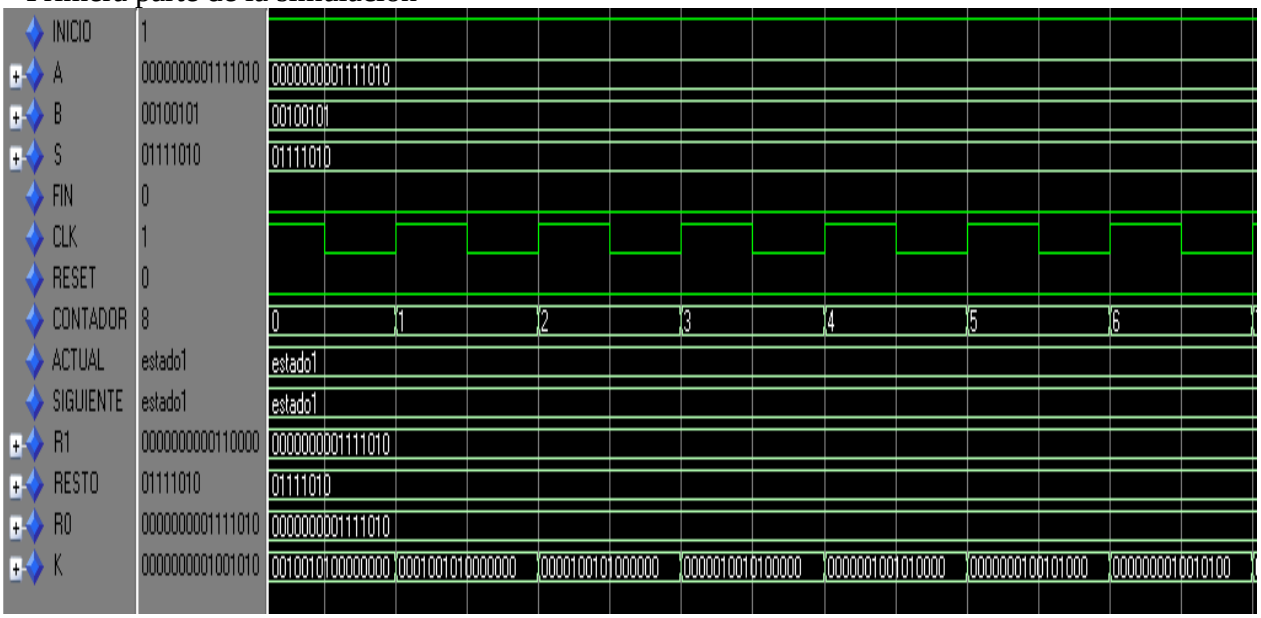


Figura 65 - SIMULACION BARRET2 1

*Segunda parte de la simulación.

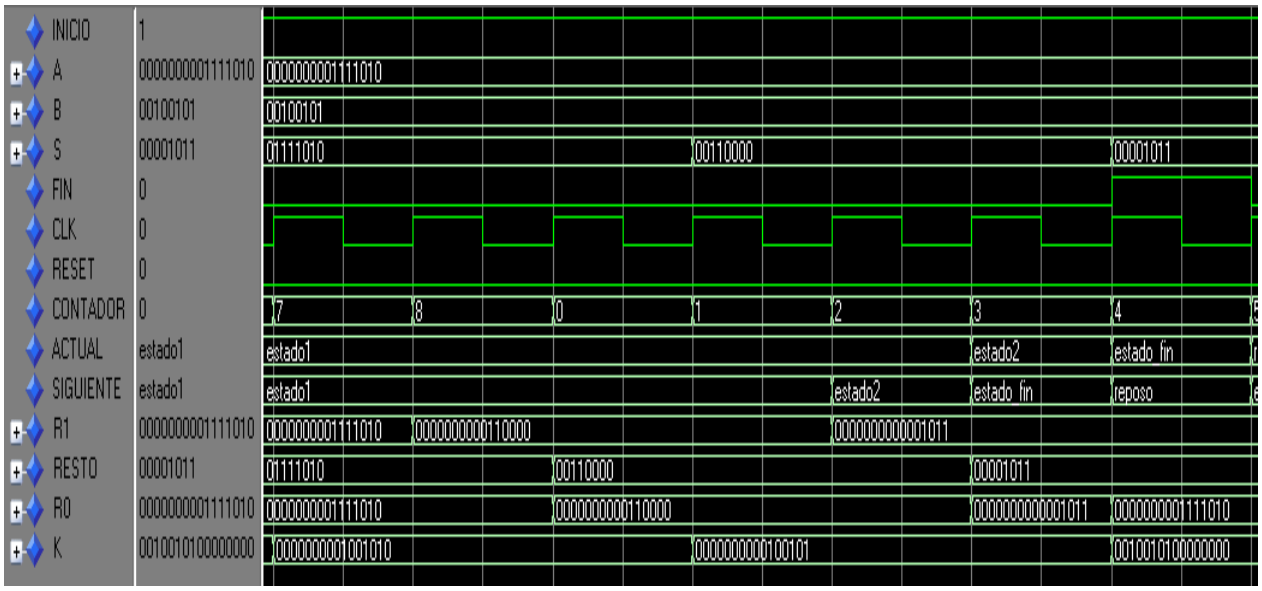


Figura 66 - SIMULACION BARRET2 2

Se observa como k ha sido desplazado n/2 hacia los bits más significativos, en cada ciclo de reloj se realiza la resta de Resto-k, en cada ciclo de reloj se comprueba como k es desplazado un bit hacia la derecha para ir restándolo a Resto. En el estado 2 se realiza de nuevo el algoritmo con el nuevo valor de k obtenido al terminar estado1, el valor de inicio del resto en el estado2 será el que se obtiene de resto al acabar el estado 1.

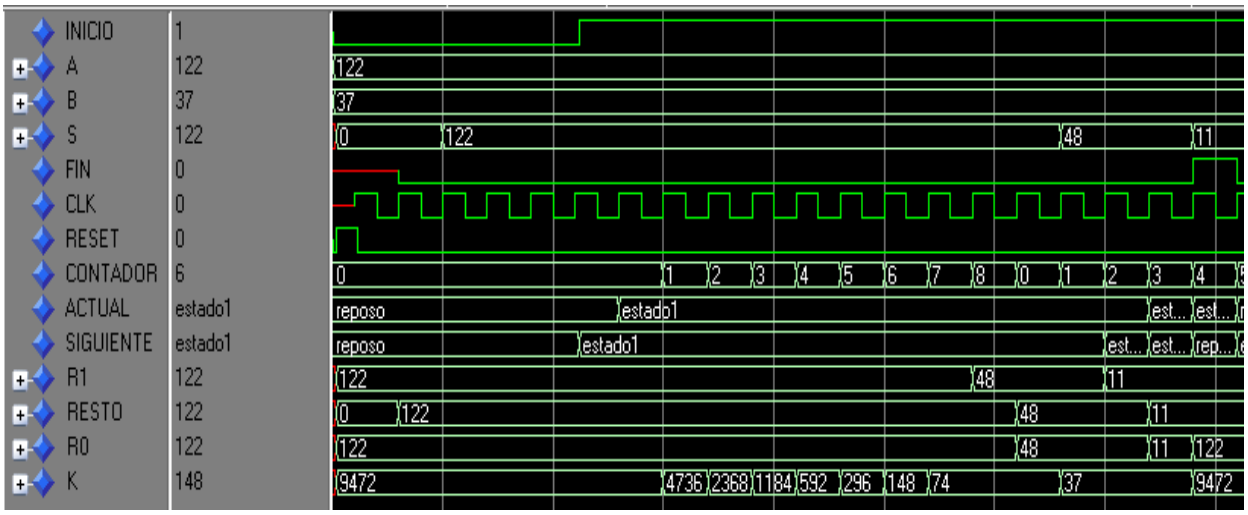


Figura 67 - SIMULACION BARRET2 3

Se comprueba como en el registro R1 se calcula el resultado de la operación del módulo, cuando el contador llega a 7 el resultado es el deseado el cual se guarda en R1, ha tardado 8 ciclos el calculo del algoritmo de ahí que este tipo de reducción (barret 2) tarde N ciclos de reloj siendo N el número de bits de los operandos.

3.4.3.4 SÍNTESIS

A continuación se muestra la síntesis de la implementación realizada a través del ISE.

N=4
Frecuencia Máxima = 168MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	58	768	7%
Number of Slice Flip Flops	39	1536	2%
Number of 4 input LUTs	109	1536	7%
Number of bonded IOBs	20	124	16%
Number of GCLKs	1	8	12%

N=8
Frecuencia Máxima = 132MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	117	768	15%
Number of Slice Flip Flops	75	1536	4%
Number of 4 input LUTs	221	1536	14%
Number of bonded IOBs	36	124	29%
Number of GCLKs	1	8	12%

N=16
Frecuencia Máxima = 112MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	234	768	30%
Number of Slice Flip Flops	147	1536	9%
Number of 4 input LUTs	440	1536	28%
Number of bonded IOBs	68	124	54%
Number of GCLKs	1	8	12%

N=32
Frecuencia Máxima = 88MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	463	768	60%
Number of Slice Flip Flops	291	1536	18%
Number of 4 input LUTs	870	1536	56%
Number of bonded IOBs	132	124	106%
Number of GCLKs	1	8	12%

N=64
Frecuencia Máxima = 62MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	968	768	126%
Number of Slice Flip Flops	581	1536	37%
Number of 4 input LUTs	1812	1536	117%
Number of bonded IOBs	260	124	209%
Number of GCLKs	1	8	12%

N=128
Frecuencia Máxima = 41MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1941	768	252%
Number of Slice Flip Flops	1159	1536	75%
Number of 4 input LUTs	3634	1536	236%
Number of bonded IOBs	516	124	416%
Number of GCLKs	1	8	12%

N=256
Frecuencia Máxima = 25MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	3083	768	401%
Number of Slice Flip Flops	2309	1536	150%
Number of 4 input LUTs	5877	1536	382%
Number of bonded IOBs	1028	124	829%
Number of GCLKs	1	8	12%

N=512
Frecuencia Máxima = 14.681MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	7750	768	1009%
Number of Slice Flip Flops	4629	1536	301%
Number of 4 input LUTs	14772	1536	961%
Number of bonded IOBs	2052	124	1654%
Number of GCLKs	1	8	12%

N=1024
Frecuencia Máxima = 7.871MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	14389	768	1873%
Number of Slice Flip Flops	9251	1536	602%
Number of 4 input LUTs	27320	1536	1778%
Number of bonded IOBs	4100	124	3306%
Number of GCLKs	1	8	12%

3.4.3.5 RESULTADOS BARRET2

En las siguientes tablas se muestra el área ocupada, frecuencia máxima y número de ciclos que tarda el algoritmo.

Nº BITS	ÁREA (LUT)	FMAX (MHZ)	CICLOS DE RELOJ
4	109	168	4
8	221	132	8
16	440	112	16
32	870	88	32
64	1812	62	64
128	3634	41	128
256	5877	25	256
512	14772	14	512
1024	27320	8	1024

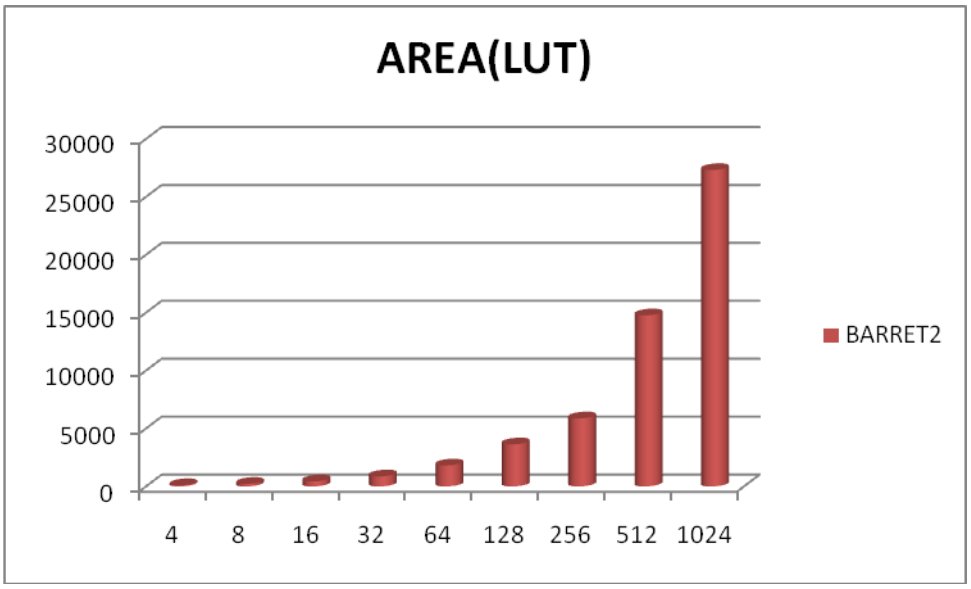


Figura 68 - AREA BARRET2

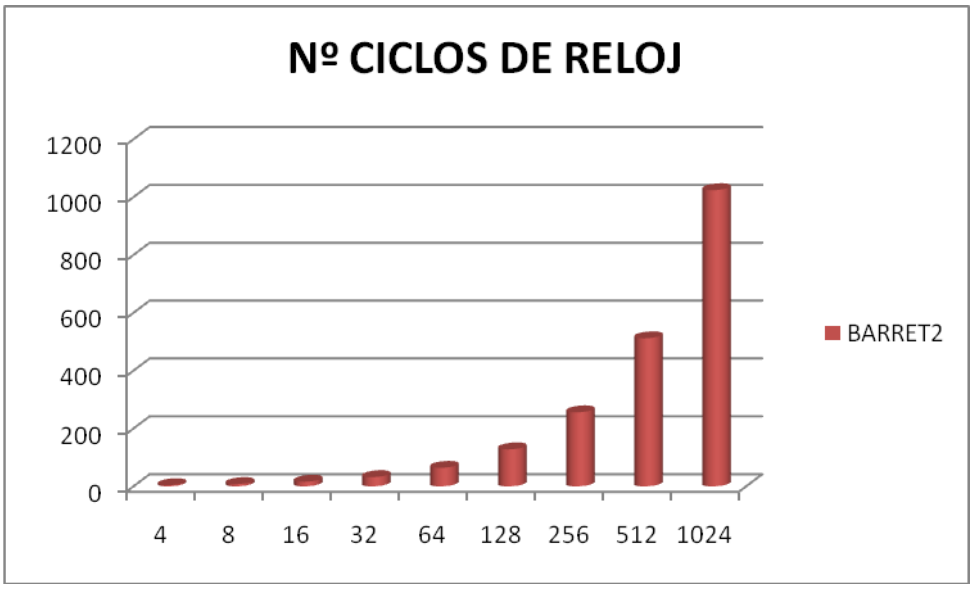


Figura 69 - CICLO BARRET2

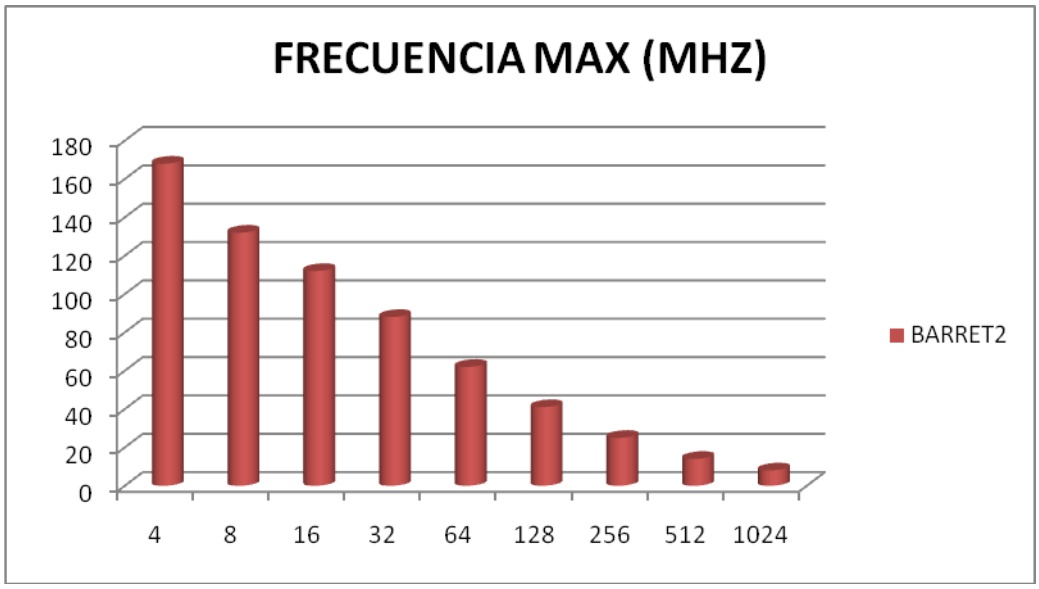


Figura 70 - FRECUENCIA BARRET2

IMPLEMENTACIÓN DE ALGORITMOS

[MONTGOMERY]

ALGORITMO – IMPLEMENTACIÓN VHDL – SIMULACIÓN – SINTESIS - RESULTADOS

3.5 MONTGOMERY

3.5.2 *ALGORITMO*

Este algoritmo obtiene directamente el modulo de la multiplicación, a diferencia de los anteriores métodos no se obtiene el resultado de la multiplicación, sino que se obtiene unos valores intermedios los cuales darán resultado a modulo del producto de los dos operandos. Para el cálculo de A * B mod M, el algoritmo será el siguiente tomando (A,B,M)

```
for i = 0 to N - 1
        d' = d' + b a[i]
        d' = d' + n d'[0]
        d' >> 1
if d' < n then return (d')
        else return (d' - n)
```

Dos aspectos a destacar, uno el dato de n es una constante que habrá que previamente calcular, esta constante será igual a 4^N mod M. Una vez calculado el resto habrá que realizar el algoritmo otra vez con los valores nuevos (d,n,M). La expresión d>>1 indica que se realizará un desplazamiento hacia bit menos significativos de 1 bit, es decir se divide por 2.

Existe dos condiciones para poder realizar este algoritmo, la primera es que lo operandos deben ser más pequeños que el modulo, y la segunda es que el modulo debe ser número impar.

Ejemplo:

N = 4 a = 5 b = 7 n = 11

El valor de salida es d'= 7 , como 4^N mod n = 3 , se usa el algoritmo de nuevo usando como a y b a estos dos últimos resultados y se obtiene el valor real de d = 2.

i	0	1	2	3
d'= + b a(i)	7	9	17	14
d'= + n d'(0)	18	20	28	14
d' >> 1	9	10	14	7

3.5.3 IMPLEMENTACION EN VHDL

En la siguiente parte del código se muestra la implementación de la parte del algoritmo de Montgomery:

Codigo=>

```
ELSIF ACTUAL=ESTADO1 OR ACTUAL=ESTADO3 THEN



                IF CC(CONTADOR)='1' THEN R:=R+DD;
                ELSE C:=CONV_STD_LOGIC_VECTOR(0,N);
                END IF;

                IF R(0)='0' THEN R:='0' & R(2*N-1 DOWNTO 1);
                Else
                R:=R+M;
                R:='0' & R(2*N-1 DOWNTO 1);
                END IF;

 ELSIF ACTUAL=ESTADO2 THEN


                 CC<=R1(n-1 downto 0);
                 DD<=E;
                 R1<=CONV_STD_LOGIC_VECTOR(0,2*N);
                 R:=CONV_STD_LOGIC_VECTOR(0,2*N);
```

En el estado 1 se realizará el algoritmo una vez a través de un contador desde i = 0 hasta i = n-1, en el estado 2 se realizará la comparación, en la cual obtendremos los valores nuevos del algoritmo (DD,CC,M), y por ultimo en el estado 3 se realizará de nuevo el algoritmo pero con los nuevos valores obtenidos. Por tanto para realizar el algoritmo necesitamos N ciclos de reloj en el estado 1, mas 1 ciclo de reloj en el estado 2, mas N ciclos de reloj en el estado 3, por lo que se puede deducir que este algoritmo tardará 2N+1 ciclos de reloj siendo N el numero de bits de los operandos. Para la comparación con los demás métodos tomaremos que tarda 2N ciclos de reloj debido a que ese ciclo de reloj de más se puede considerar despreciable para número de bits elevados como es el caso en el que se quiere utilizar.

El diagrama de flujo se muestra a continuación:

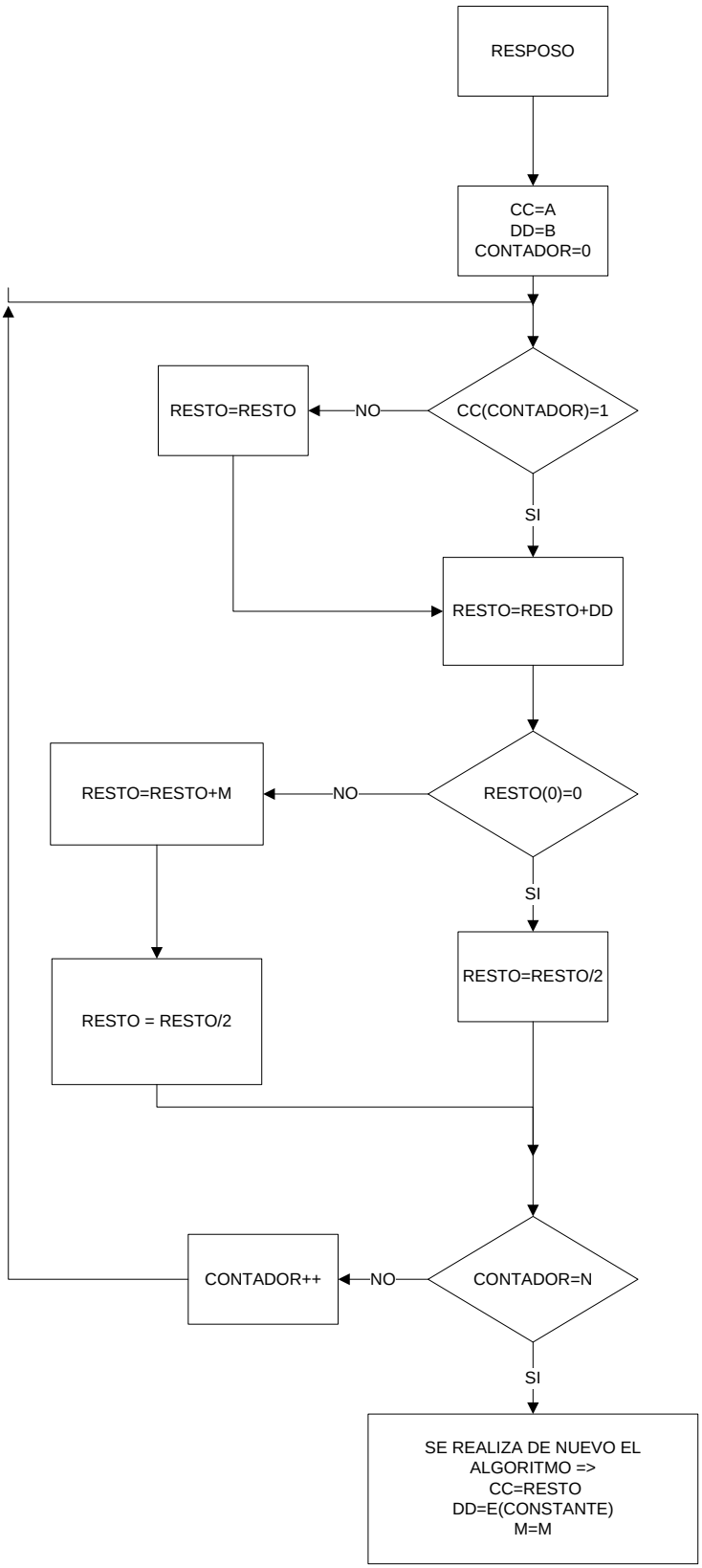


Figura 71 - D.ESTADOS MONTGOMERY

3.5.4 SIMULACIÓN

A continuación se muestra la simulación con 4 bits

N = 4 A = 5 B = 7 n = 11

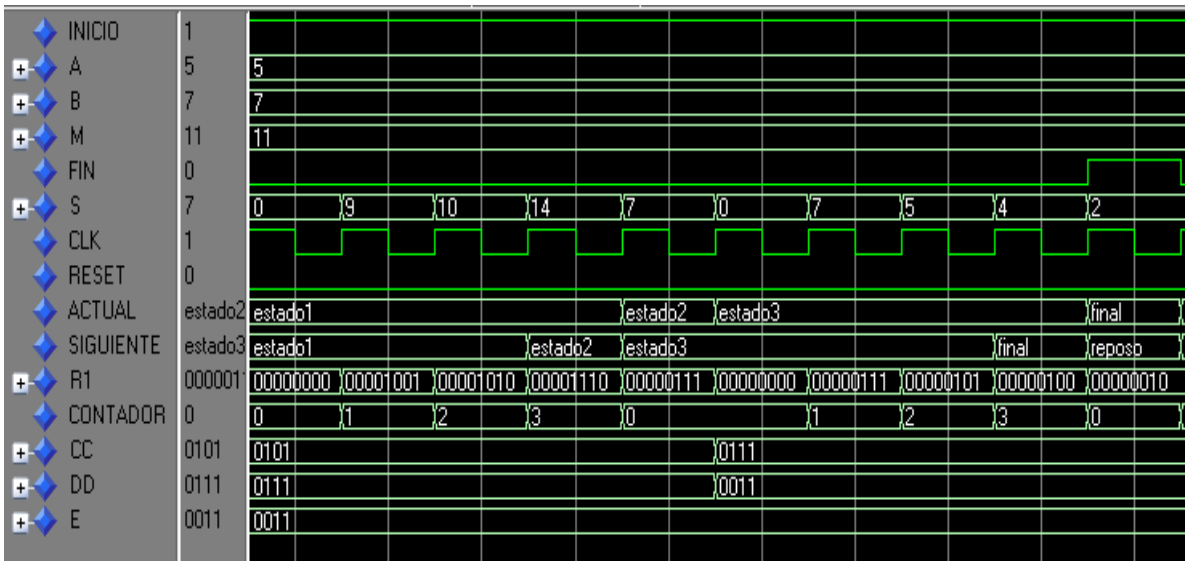


Figura 72 - SIMLACION MONTGOMERY 1

En primer lugar se observa como el contador se realiza dos veces, pues este algoritmo necesita realizarse por duplicado, la primera parte del algoritmo realiza (5,7,11), en la segunda parte que es cuando empieza estado3 realiza el mismo algoritmo con los valores de (cc,dd,E), el valor de E que es la constante se ha calculado previamente y vale 11. En el estado final se activa el puerto final para indicar que ha acabado el algoritmo y en el puerto S esta el resultado final.

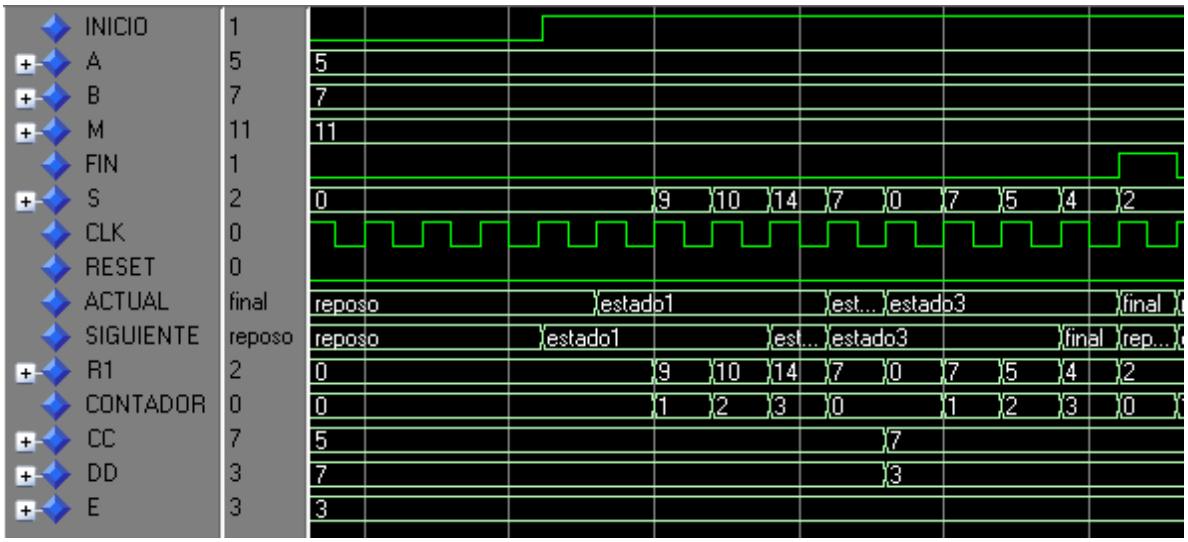


Figura 73 - SIMLACION MONTGOMERY 2

Se observa en esta parte de la simulación que los nuevos valores con los cuales se realiza el algoritmo son (5,7,3) cuyo resultado es 2.

La simulación completa queda representada en la siguiente simulación:

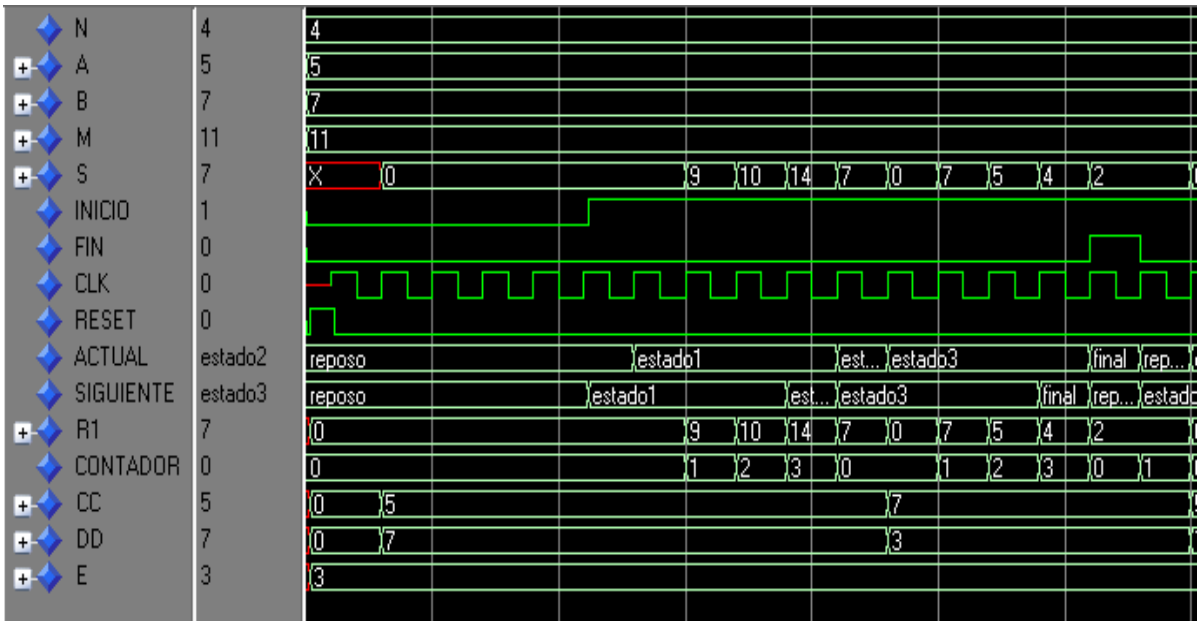


Figura 74 - SIMULACION MONTGOMERY 3

Con operandos de 16 bits la simulación será la siguiente: (450,210,101)

(1)

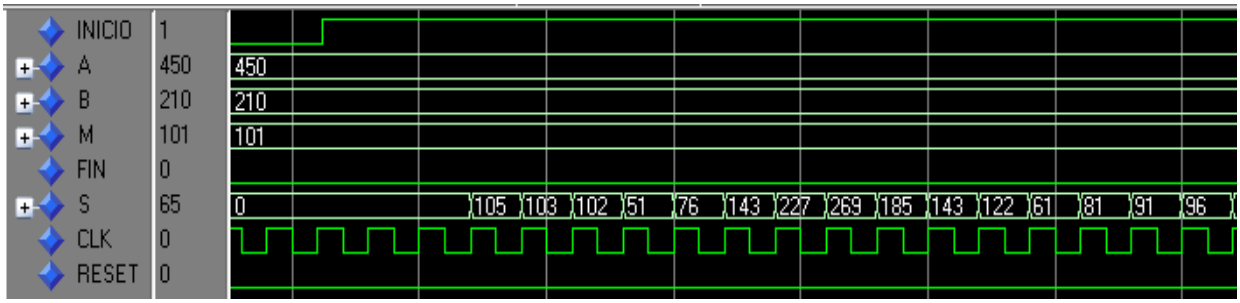


Figura 75 - SIMULACION MONTGOMERY 4

(2)

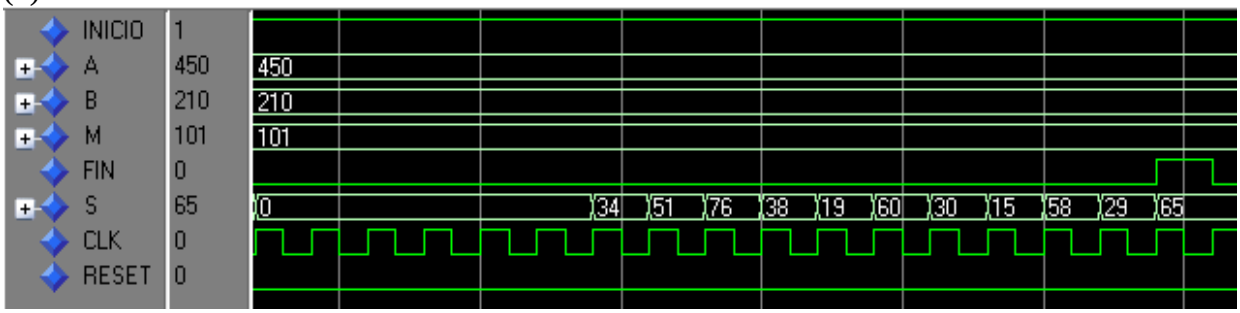


Figura 76 - SIMULACION MONTGOMERY 5

Se puede comprobar que en este caso la parte del algoritmo ha durado 32 ciclos de reloj, pues en la implementación de este algoritmo para obtener el resultado se tarda 2 *N ciclos de reloj siendo N el numero de bits de los operandos.

3.5.5 SÍNTESIS

N=4
Frecuencia Máxima = 122MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	31	768	4%
Number of Slice Flip Flops	24	1536	1%
Number of 4 input LUTs	59	1536	3%
Number of bonded IOBs	20	124	16%
Number of GCLKs	1	8	12%

N=8
Frecuencia Máxima = 113MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	60	768	7%
Number of Slice Flip Flops	45	1536	2%
Number of 4 input LUTs	115	1536	7%
Number of bonded IOBs	36	124	29%
Number of GCLKs	1	8	12%

N=16
Frecuencia Máxima = 96MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	119	768	15%
Number of Slice Flip Flops	86	1536	5%
Number of 4 input LUTs	225	1536	14%
Number of bonded IOBs	68	124	54%
Number of GCLKs	1	8	12%

N=32
Frecuencia Máxima = 88MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	247	768	32%
Number of Slice Flip Flops	170	1536	11%
Number of 4 input LUTs	442	1536	28%
Number of bonded IOBs	132	124	106%
Number of GCLKs	1	8	12%

N=64
Frecuencia Máxima = 59MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	501	768	65%
Number of Slice Flip Flops	332	1536	21%
Number of 4 input LUTs	880	1536	57%
Number of bonded IOBs	260	124	209%
Number of GCLKs	1	8	12%

N=128
Frecuencia Máxima = 42 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	975	768	126%
Number of Slice Flip Flops	658	1536	42%
Number of 4 input LUTs	1756	1536	114%
Number of bonded IOBs	516	124	416%
Number of GCLKs	1	8	12%

N=256
Frecuencia Máxima = 24MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1914	768	249%
Number of Slice Flip Flops	1305	1536	84%
Number of 4 input LUTs	3484	1536	226%
Number of bonded IOBs	1028	124	829%
Number of GCLKs	1	8	12%

N= 512

Frecuencia Máxima = 14.413MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	3815	768	496%
Number of Slice Flip Flops	2606	1536	169%
Number of 4 input LUTs	6966	1536	453%
Number of bonded IOBs	2052	124	1654%
Number of GCLKs	1	8	12%

N= 1024
Frecuencia Máxima = 7.898MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	7693	768	1001%
Number of Slice Flip Flops	5223	1536	340%
Number of 4 input LUTs	14035	1536	913%
Number of bonded IOBs	4100	124	3306%
Number of GCLKs	1	8	12%

3.5.6 RESULTADOS MONTGOMERY

En las siguientes tablas mostramos los resultados en cuanto a área ocupada (LUT), número de ciclos y frecuencia máxima del circuito.

Nº BITS	ÁREA (LUT)	FMAX (MHZ)	CICLOS DE RELOJ
4	59	122	9
8	115	113	17
16	225	96	33
32	442	88	65
64	880	59	129
128	1756	42	257
256	3484	24	513
512	6966	14	1025
1024	14035	8	2049

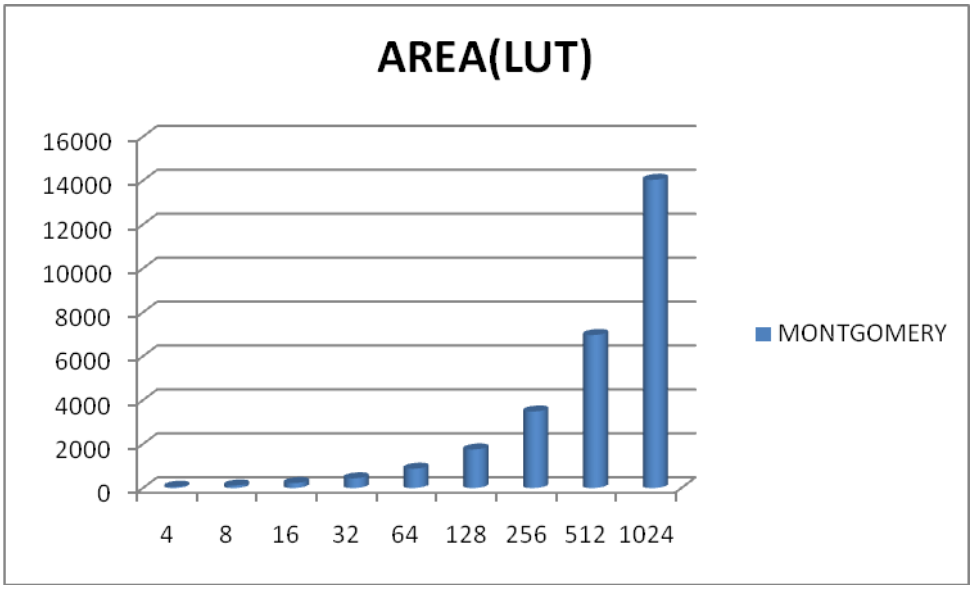


Figura 77 - AREA MONTGOMERY

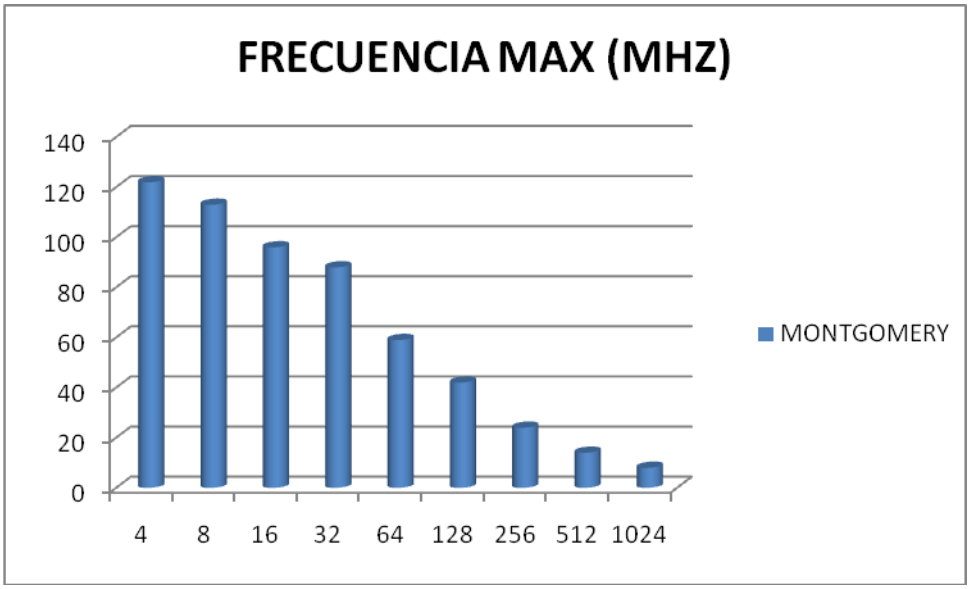


Figura 78 - FRECUENCIA MONTGOMERY

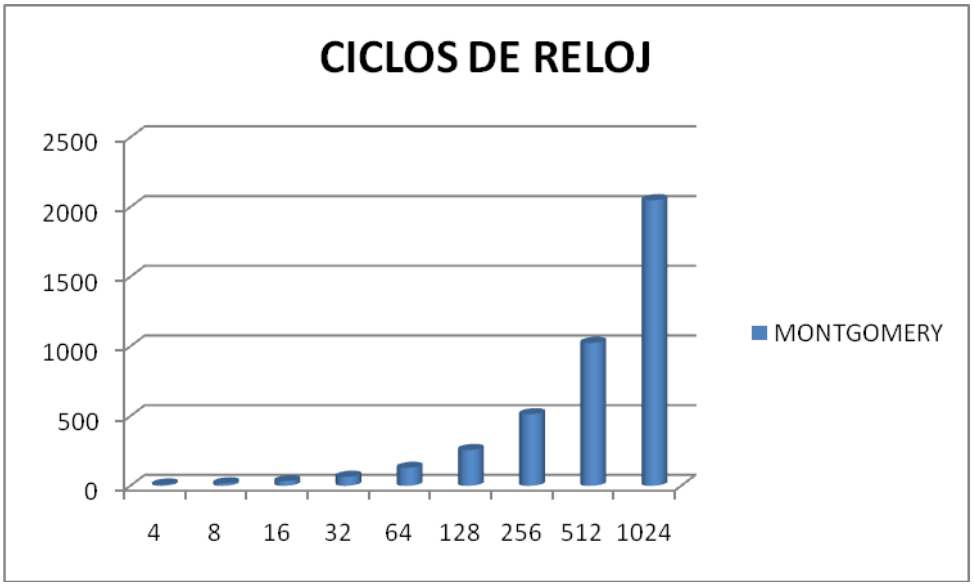


Figura 79 - CICLOS MONTGOMERY

IMPLEMENTACIÓN DE ALGORITMOS

[BUCKLEY]

ALGORITMO – IMPLEMENTACIÓN VHDL – SIMULACIÓN – SINTESIS - RESULTADOS

3.6 BUCKLEY

3.6.2 ALGORITMO

Este algoritmo al igual que el de Montgomery realiza obtiene directamente el modulo de la multiplicación de dos operandos.

El algoritmo utilizado será el siguiente:

Entrada: $A(a_{n-1},\ldots,a_0)$, $B(b_{n-1},\ldots,b_0)$ y $M(b_{n-1},\ldots,b_0)$
Salida: $R = A \cdot B \mod M$

1. $R = 0$;
2. **para** $i = 0$ **hasta** $n-1$ **hágase**
3. $\quad R = SHIFT\_LEFT(R)$;
4. $\quad$ **si** $a_{n-1-i} = 1$ **entonces**
5. $\quad\quad R = R + B$;
6. $\quad\quad R = R \mod M$;
7. **regresa** R

Una vez realizado el algoritmo tendríamos que realizar el modulo par que nos dé el resultado final, para esta parte del código podríamos aplicar barret1 o barret2, como se ha explicado anteriormente utilizaríamos barret2 debido a que tardaría menos. También este algoritmo se puede obtener el modulo realizando como mucho dos sustracciones.

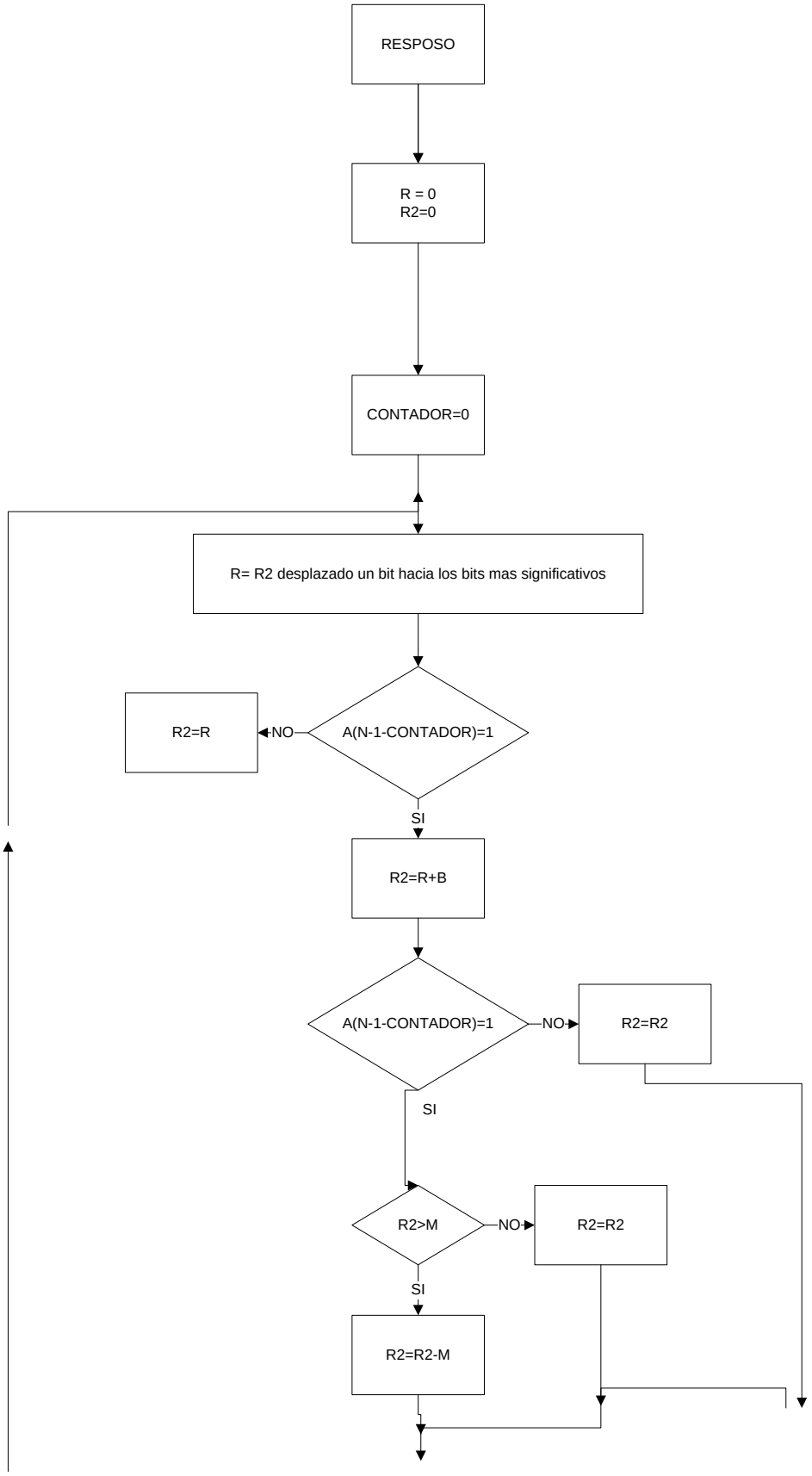
Ejemplo: A= 15 B = 10 M = 13

A(i)	R <= 1 bit	R+B	R mod M
3	0	10	10
2	20	30	17
1	34	44	31
0	62	72	59

Como A es 15 que en número binario es 1111, en la comparación de si el bit de A es 1 siempre se da, por tanto en este caso siempre se realiza la suma de R+B y R mod M.

El valor obtenido tras realizar buckley es 59, a este nuevo valor lo que se realizará será aplicar un algoritmo de reducción como el de barret 2, por tanto los nuevos valores los cuales se realizara el algoritmo en barret 2 será P=59 y M=13, dando como resultado el 7.

El diagrama de flujo es el siguiente:



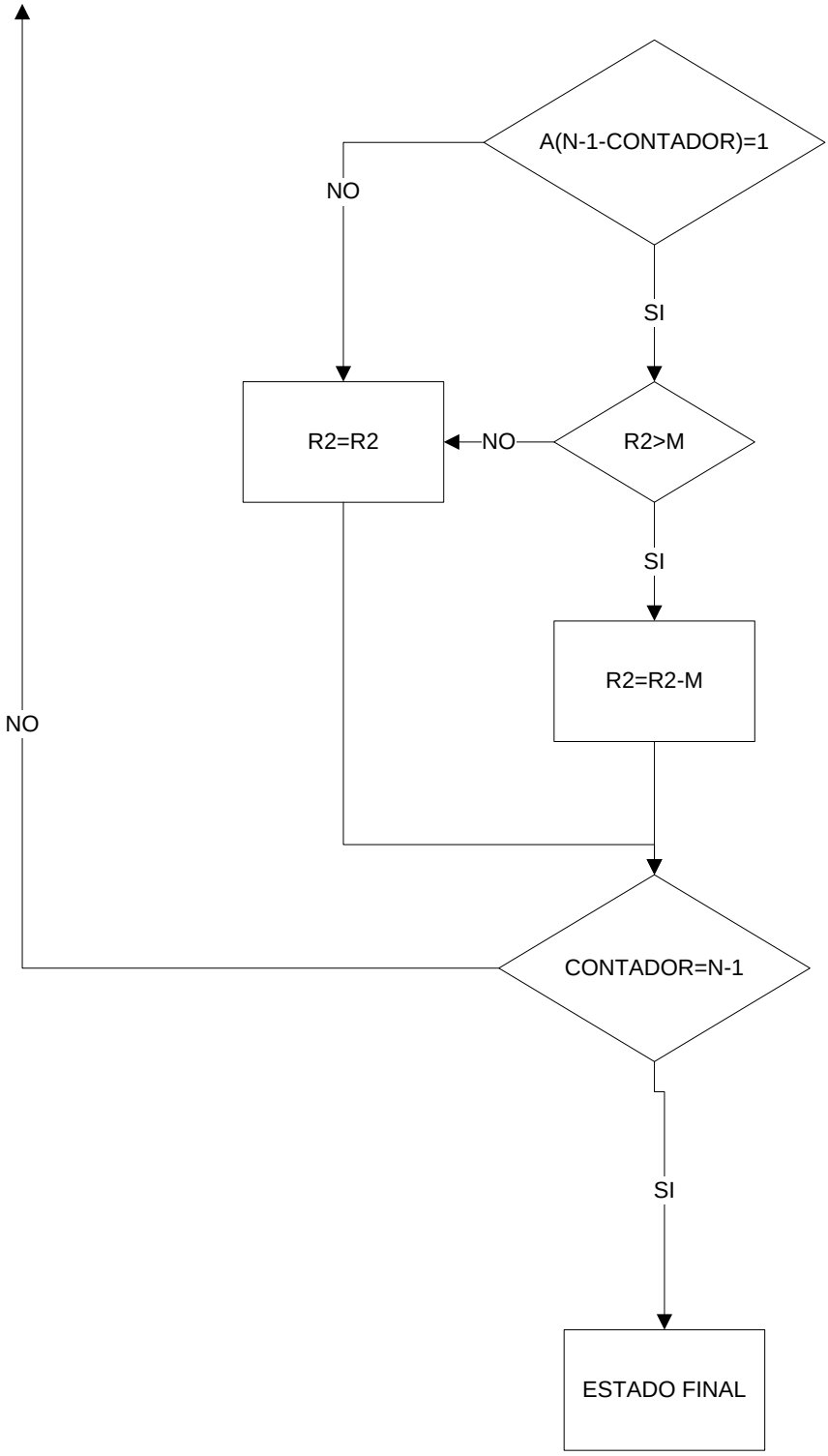


Figura 80 - D.ESTADOS BUCLEY

3.6.3 IMPLEMENTACIÓN EN VHDL

La parte del código en la cual que se obtiene la implementación es la siguiente:

```
ELSIF ACTUAL=ESTADO1 THEN
                    ENABLE3<='0';
                    R:=R2(2*N-2 DOWNTO 0) & '0';

                    IF A(N-1-CONTADOR)='1' THEN
                       R2<=R+B;
                       else R2<=R;
                     END IF;

                  ELSIF ACTUAL=ESTADO2 THEN
                     ENABLE3<='0';
                     CONTADOR<=CONTADOR+1;

                      IF A(N-1-CONTADOR)='1' THEN
                        IF R2>M THEN
                        R2<=R2-M;
                        END IF;
                      END IF;

                  ELSIF ACTUAL=ESTADO3 THEN
                      CONTADOR<=0;
                      IF R2>M THEN
                         R2<=R2-M;
                         ENABLE3<='0' ;
                         ELSE ENABLE3<='1';
                      END IF;
```

En cada contador = i se necesita tres estados, como para realizar el algoritmo necesitamos que el contador vaya desde i = 0 hasta n-1 podemos concluir que este algoritmo necesita 2N ciclos de reloj para obtener el resultado al finalizar el estado 1 y 2. Una vez que tenemos este resultado podemos obtener el modulo simplemente realizando la resta de R2 –M, este parte del algoritmo se puede realizar de esta manera que sería como realizar de la forma de barret1 u aplicando barret 2. Si aplicásemos barret2 habría que añadir N ciclos de reloj. Por tanto el algoritmo en total tardaría los 2N (estado1,estado2) mas N (estado3), en total tardaría 3N en el caso de obtener el módulo a través del barret2. Este algoritmo a diferencia del de Montgomery no tiene condiciones iniciales, además no necesita una constante previamente calculada como en el caso del Montgomery.

3.6.4 SIMULACION

Realizamos la simulación para 4 bits de A=15, B=10 y M=13. El resultado debe ser 7

Como en este caso A es 15=1111 en la comparación del bit siempre es 1 por lo que siempre en el estado 2 se realiza el módulo, además se observa como una vez calculado el resto antes de calcular el nuevo valor con i+1 se desplaza un bit hacia los bits mas significativos.

(1)

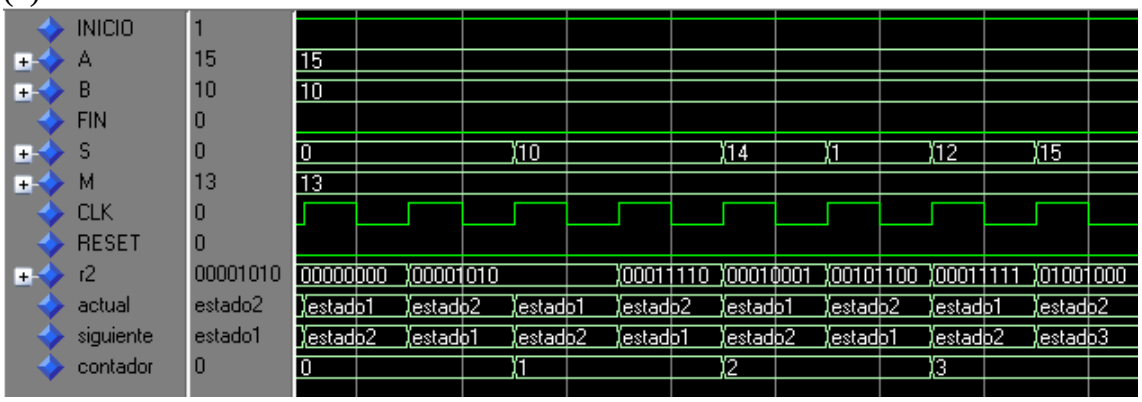


Figura 81 - SIMULACION BUCKLEY 1

(2)

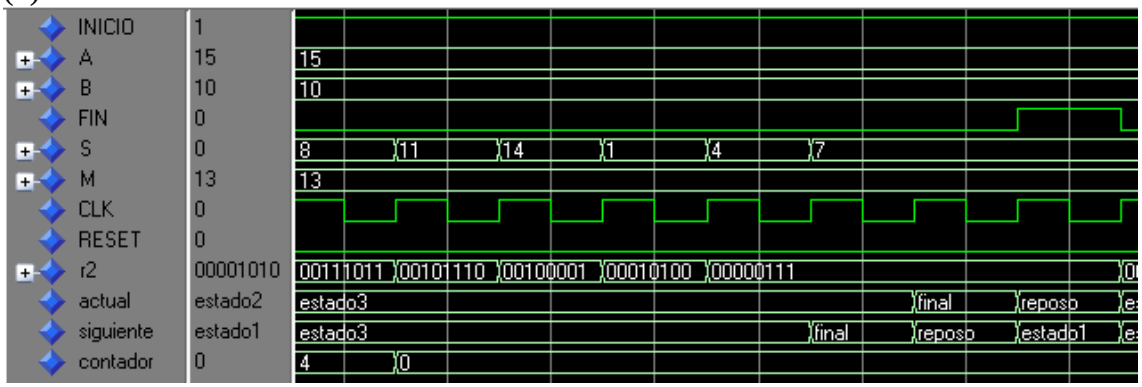


Figura 82 - SIMULACION BUCKLEY 2

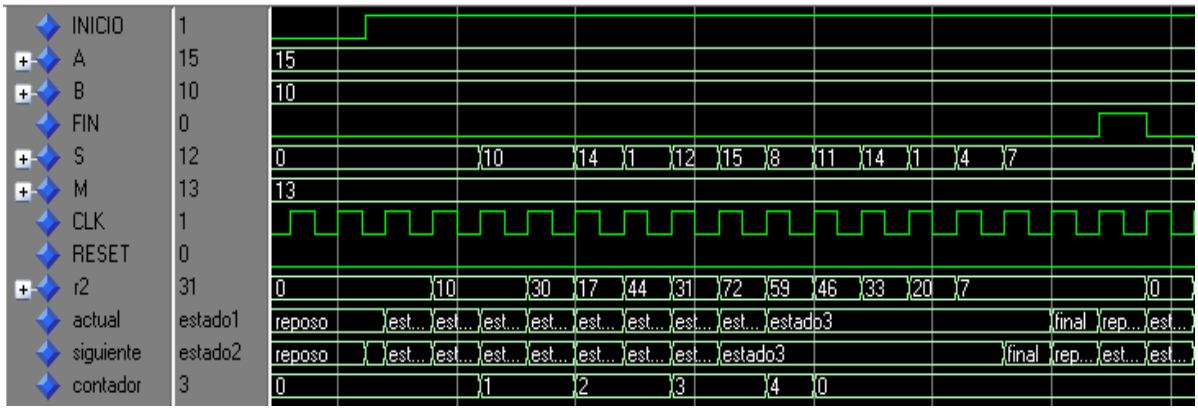


Figura 83 - SIMULACION BUCKLEY 3

En la simulación se observa cómo entre el estado 1 y 2 ha tardado 8 ciclos de reloj para calcular el resultado al cual habrá que aplicarle la reducción. En el estado 3 se calcula el resto el cual se puede comprobar cómo una vez que estamos en el estado 3 se tarda 4 ciclos de reloj en calcular el resultado que será 7. Por tanto el tiempo que ha tardado en calcular el resultado a través de este algoritmo ha sido de 12 ciclos. En esta simulación hay más ciclos de reloj para calcular el resultado debido a que inicio no comienza en el primer ciclo y a que una vez terminado el algoritmo se para al estado final para activar el bit de fin.

3.6.5 SINTESIS

N=4
Frecuencia Máxima = 150MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	36	768	4%
Number of Slice Flip Flops	24	1536	1%
Number of 4 input LUTs	69	1536	4%
Number of bonded IOBs	20	124	16%
Number of GCLKs	1	8	12%

N=8
Frecuencia Máxima = 126MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	72	768	9%
Number of Slice Flip Flops	35	1536	2%
Number of 4 input LUTs	140	1536	9%
Number of bonded IOBs	36	124	29%
Number of GCLKs	1	8	12%

N=16
Frecuencia Máxima = 122MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	101	768	13%
Number of Slice Flip Flops	61	1536	3%
Number of 4 input LUTs	193	1536	12%
Number of bonded IOBs	68	124	54%
Number of GCLKs	1	8	12%

N=32
Frecuencia Máxima = 105MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	193	768	25%
Number of Slice Flip Flops	116	1536	7%
Number of 4 input LUTs	370	1536	24%
Number of bonded IOBs	132	124	106%
Number of GCLKs	1	8	12%

N=64
Frecuencia Máxima = 76MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	372	768	48%
Number of Slice Flip Flops	223	1536	14%
Number of 4 input LUTs	714	1536	46%
Number of bonded IOBs	260	124	209%
Number of GCLKs	1	8	12%

N=128
Frecuencia Máxima =49MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	731	768	95%
Number of Slice Flip Flops	417	1536	27%
Number of 4 input LUTs	1402	1536	91%
Number of bonded IOBs	516	124	416%
Number of GCLKs	1	8	12%

N=256
Frecuencia Máxima = 29MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1454	768	189%
Number of Slice Flip Flops	798	1536	51%
Number of 4 input LUTs	2788	1536	181%
Number of bonded IOBs	1028	124	829%
Number of GCLKs	1	8	12%

N=512
Frecuencia Máxima = 15.873MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	2891	768	376%
Number of Slice Flip Flops	1569	1536	102%
Number of 4 input LUTs	5548	1536	361%
Number of bonded IOBs	2052	124	1654%
Number of GCLKs	1	8	12%

N=1024
Frecuencia Máxima = 8.297MHz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	5768	768	751%
Number of Slice Flip Flops	3115	1536	202%
Number of 4 input LUTs	11072	1536	720%
Number of bonded IOBs	4100	124	3306%
Number of GCLKs	1	8	12%

3.6.6 RESULTADOS

En las siguientes tablas mostramos los resultados en cuanto a área ocupada (LUT), número de ciclos y frecuencia máxima del circuito.

Nº BITS	ÁREA (LUT)	FMAX (MHZ)	CICLOS DE RELOJ
4	69	150	12
8	140	126	24
16	193	122	48
32	370	103	96
64	714	76	192
128	1402	49	384
256	2788	29	768
512	5548	16	1536
1024	11072	9	3072

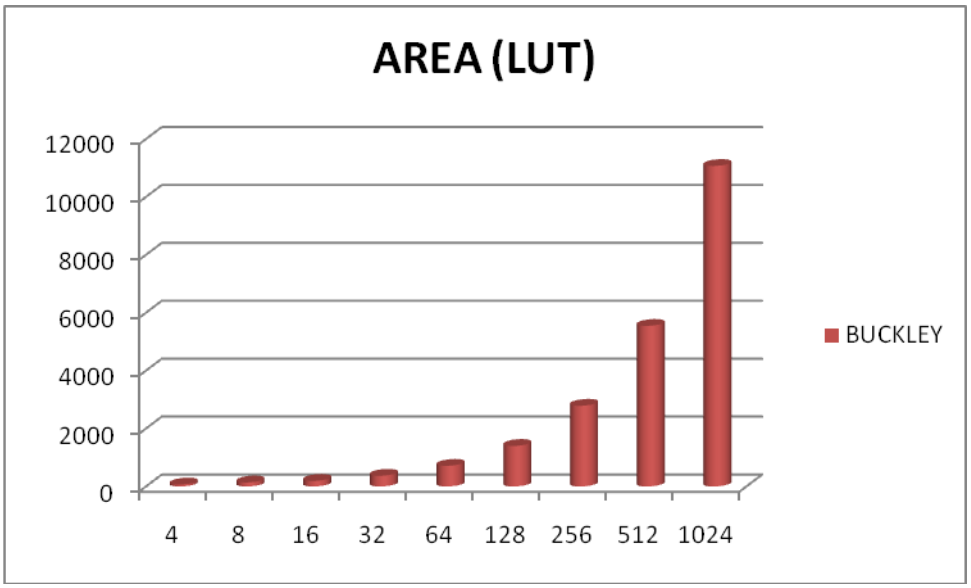


Figura 84 - AREA BUCKLEY

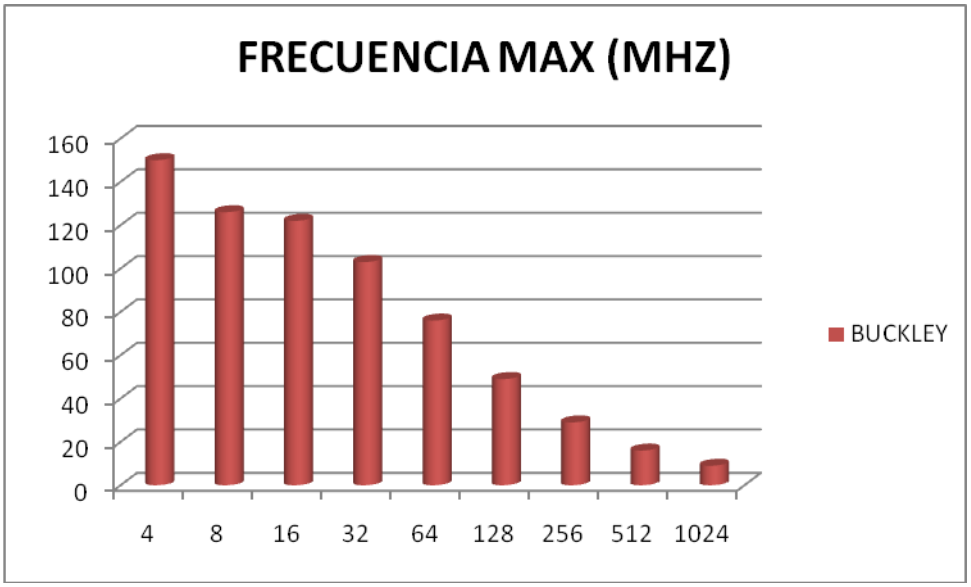


Figura 85 - FRECUENCIA BUCKLEY

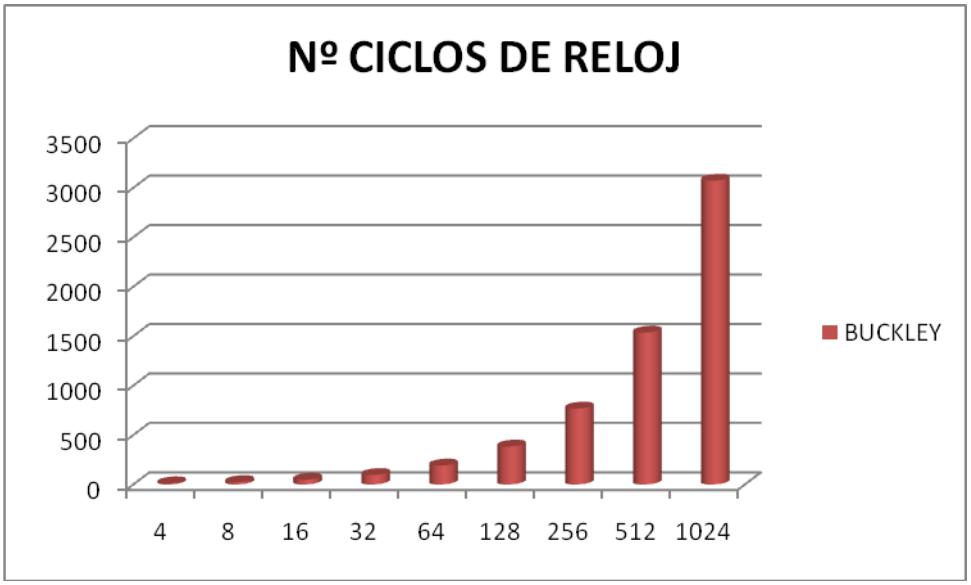


Figura 86 - CICLOS BUCKLEY

IMPLEMENTACIÓN DE ALGORITMOS

[CONCLUSIÓN MULTIPLICACIÓN MODULAR]

CLASICO/KARATSUBA – OFMAN/COMBINACIONAL.
N º DE CICLOS – FRECUENCIA MÁXIMA – ÁREA OCUPADA

MONTGOMERY/ BUCKLEY
ÁREA OCUPADA – Nº DE CICLOS – FRECUENCIA MÁXIMA – AREA X Nº DE CICLOS

COMBINACIONAL/CLASICO + BARRET2/KARATSUBA + BARRET2 /MONTGOMERY/BUCKLEY
ÁREA OCUPADA – TIEMPO –AREA X Nº DE CICLOS DE RELOJ

3.7 CONCLUSIÓN DE RESULTADOS

3.7.2 CONCLUSIÓN COMBINACIONAL_CLASICO_KARATSUBA

En primer lugar compararemos los tres algoritmos en el tiempo que tardan en realizar la multiplicación. En la siguiente gráfica se compara los tres métodos:

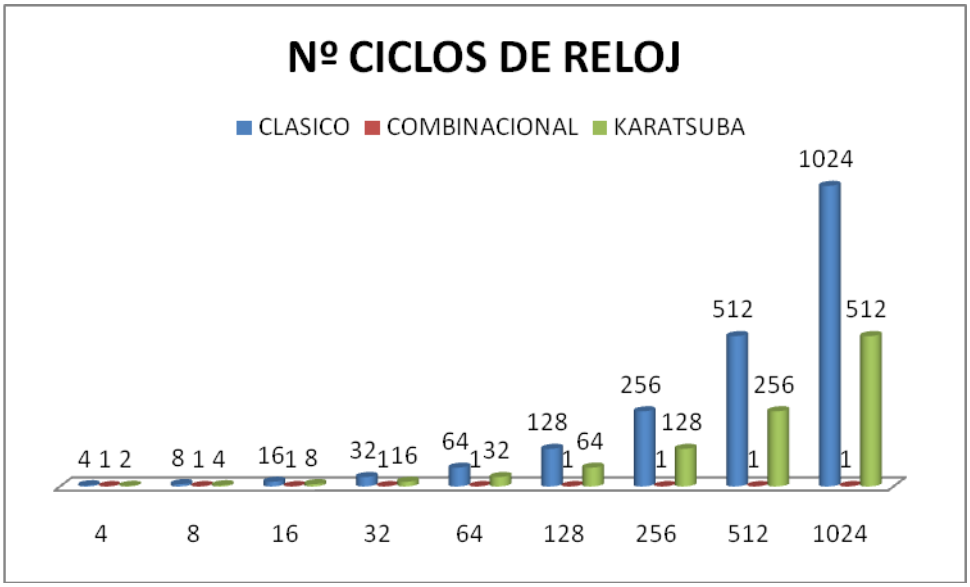


Figura 87 - CICLOS COMB_CLASICO_KARAT

Se observa que el multiplicador Combinacional tarda un ciclo de reloj ya que es totalmente combinacional, entre el Clásico y el de Karatsuba es más rápido este último, por tanto si lo que nos interesa fuese la rapidez elegiríamos la implementación del multiplicador combinacional seguido del de Karatsuba y por último eligiríamos el Clásico. A continuación mostramos a la frecuencia máxima que puede ir el reloj para las tres implementaciones.

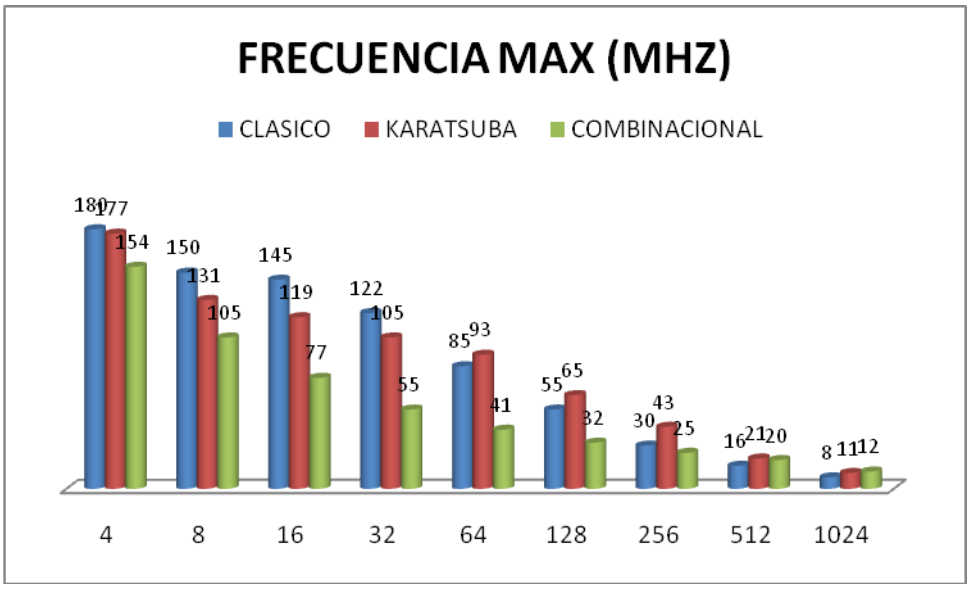
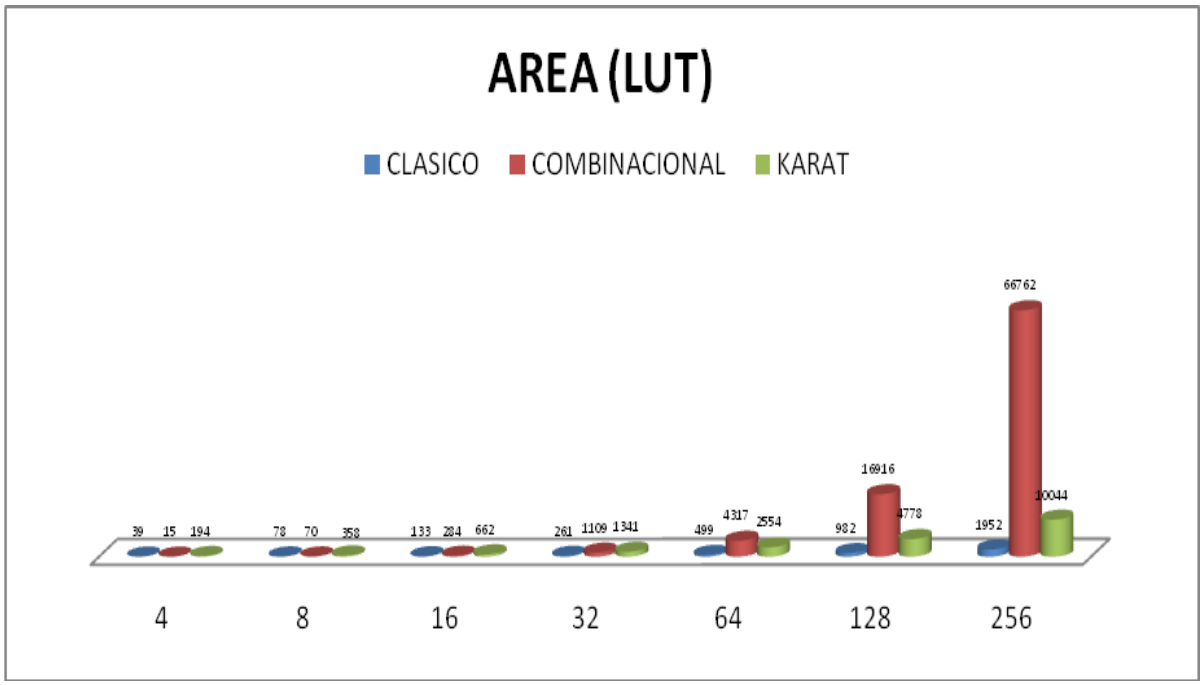


Figura 88 - FRECUENCIA COMB_CLA_KARAT

Para bajo números de bits la frecuencia máxima que podemos poner en el reloj es mayor que en el karatsuba, para número de bits a partir de 64 bits la frecuencia del reloj es mayor la de karatsuba, debido a que en la criptografía se utiliza un número elevado de bits podemos concluir que si lo que necesitamos es velocidad en el proceso utilizaríamos la implementación de karatsuba en vez de la del multiplicador clásico. En el que menos velocidad podemos introducir es en el combinacional debido a que es el que mayor parte de circuito combiancional posee.

Por otro lado compararemos área que ocupa cada implementación, medida en número de LUT.



La primera conclusión observando la gráfica es que para número de bits altos el multiplicador combinacional ocupa mucho espacio en comparación con los otros dos. Debido a que en la criptografía se utiliza número de bits (512,1024…) esta implementación la podemos descartar debido a la excesiva área que ocupa. En cuanto al karatsuba y clásico podemos decir que ocupa más el karatsuba que el clásico a medida que el número de bits es mayor esa diferencia de área también es mayor.

Finalmente podemos afirmar que para la criptografía entre las tres implementaciones expuestas la del multiplicador combinacional se descarta debido al área utilizada. Por otro lado entre los otros dos algoritmos dependerá de si lo que se necesita es velocidad en el cálculo o se necesita área utilizaremos una u otra. En el caso de que necesitemos velocidad utilizaremos Karatsuba y en el caso de que necesitemos área utilizaremos la implementación del multiplicador clásico

3.7.3 CONCLUSIÓN DE RESULTADOS BARRET

A continuación compararemos las dos implementaciones hechas para realizar el módulo. La primera comparación que haremos será la de área ocupada que se muestra en la siguiente gráfica:

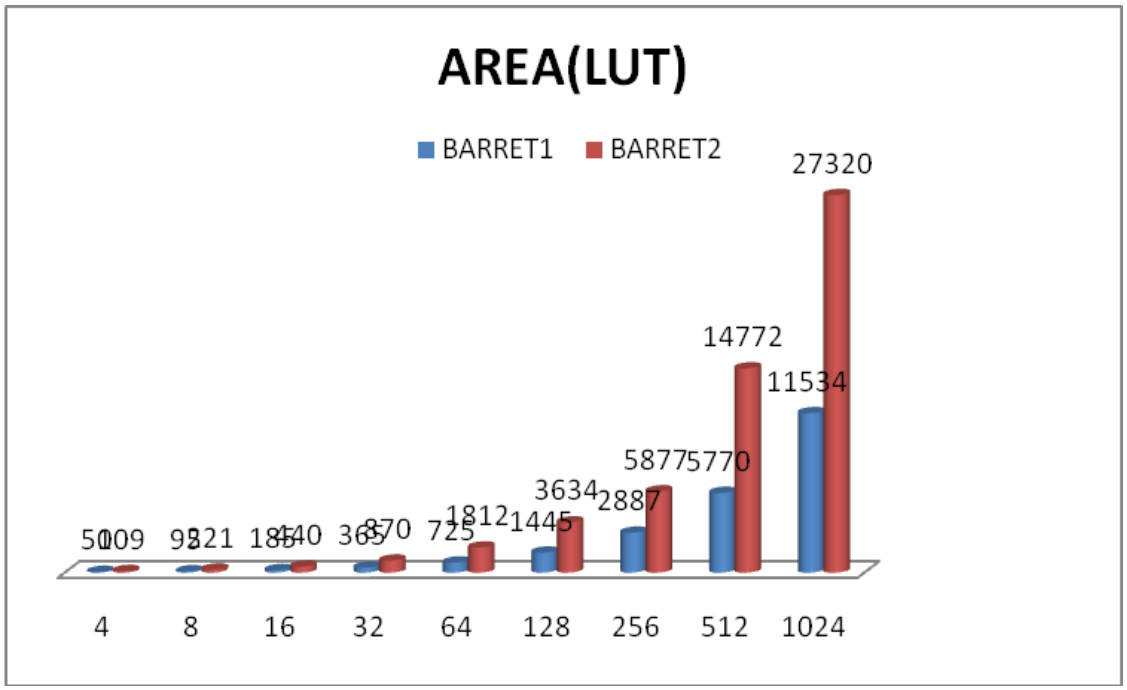


Figura 89 - AREA BARRET1_BARRET2

Se observar que el barret2 ocupa mayor área que el barret 1, por lo que necesitaremos un circuitos con mayor espacio si se quiere realizar la implementación barret 2.

Por otro lado compararemos la velocidad de las dos implementaciones, por un lado observando la frecuencia máxima a la cual puede ir el reloj, y por otro lado el número de ciclos que tarda cada implementación.

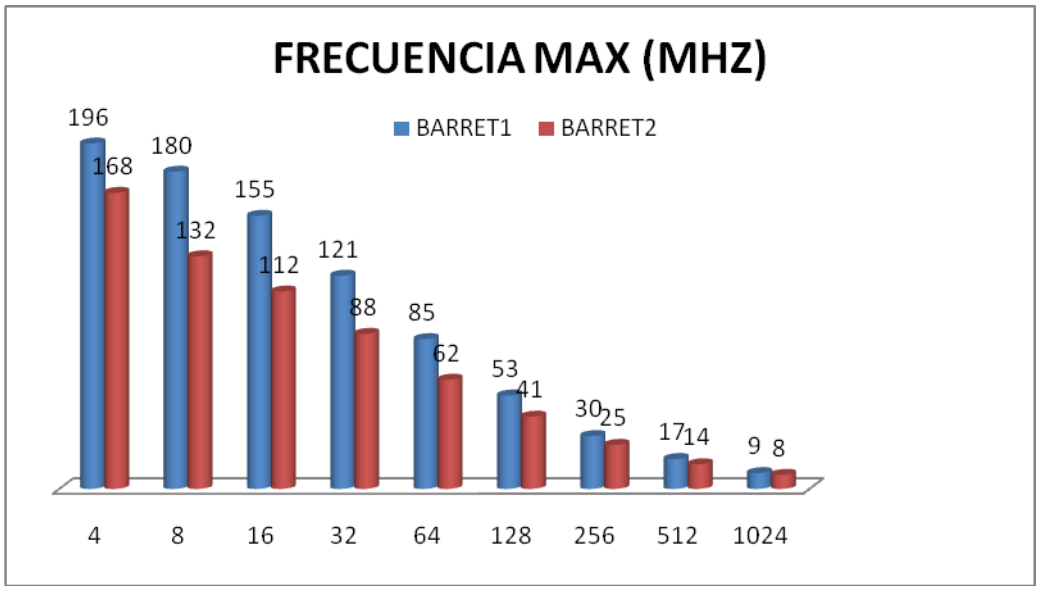


Figura 90 - FRECUENCIA BARRET1_BARRET2

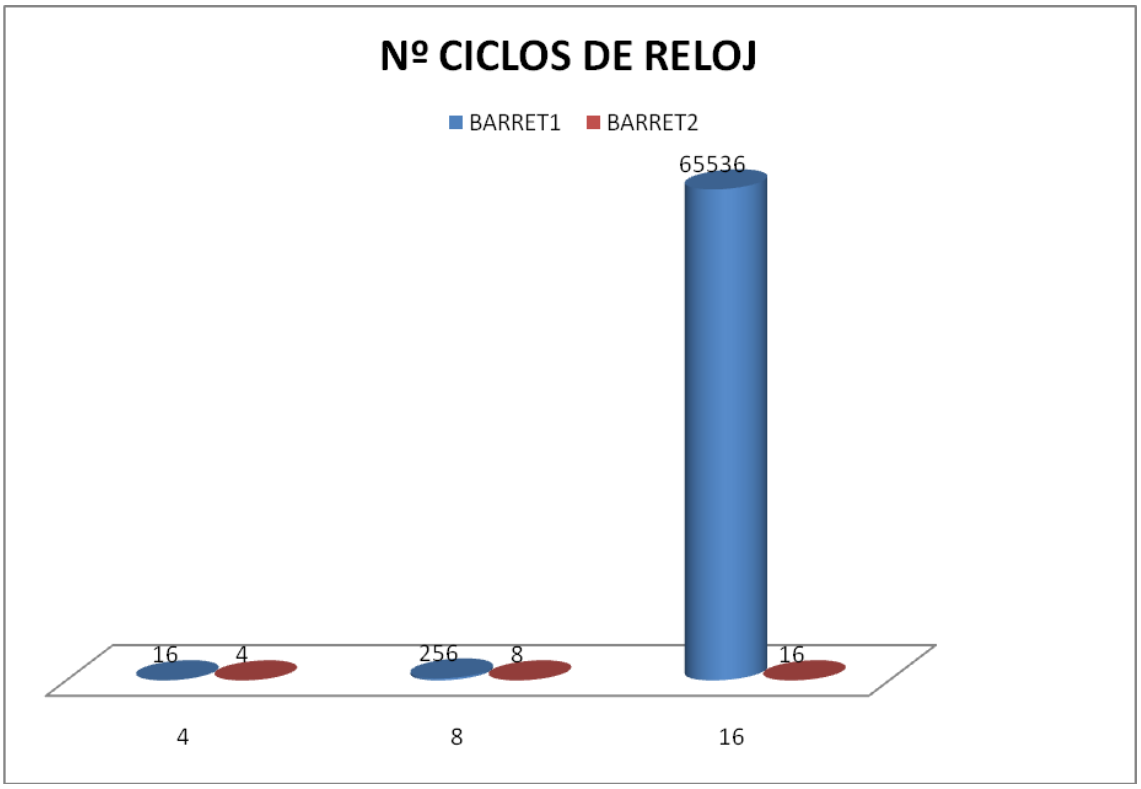


Figura 91 - CICLOS BARRET1_BARRET2

Fijandonos en la grafica de frecuencia maxima se puede decir que podemos poner una velocidad mayor del reloj en el barret1 que en el barret2, pero por otro lado obsevando en la grafica de numero de ciclos que a medidad que aumentamos el número de bits la diferencia de numero de ciclos de reloj es mucho mayor que el barret2, como en la criptografia se utilizan número de bits grandes, se puede concluir que no nos sirve el barret 1 debido al tiempo que tarda y tendremos que utilizar la implementación de barret2 a pesar de que ocupa mayor área.

3.7.4 CONCLUSIÓN RESULTADOS BUCKLEY-MONTGOMERY

En primer lugar compararemos los resultados del área calculada.

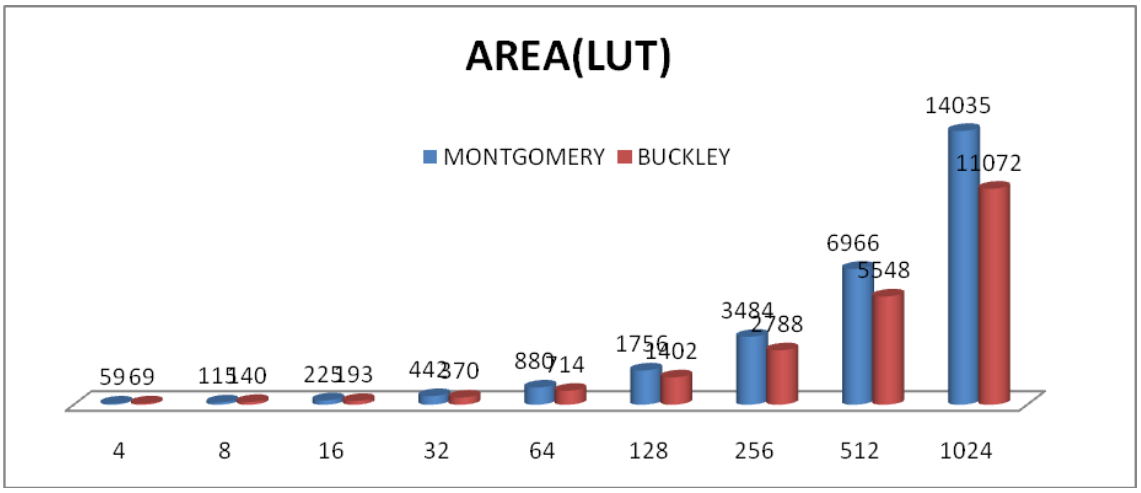


Figura 92 - AREA MONTGOMERY_BUCKLEY

Se observa que el algoritmo de montgomery ocupa mas que el de buckley, por tanto para un circuito en el que necesitemos menor área utilizaremos el algoritmo de buckley.

A continuación comparamos los resultado en cuanto a la frecuencia máxima utilizada y el numero de ciclos de reloj.

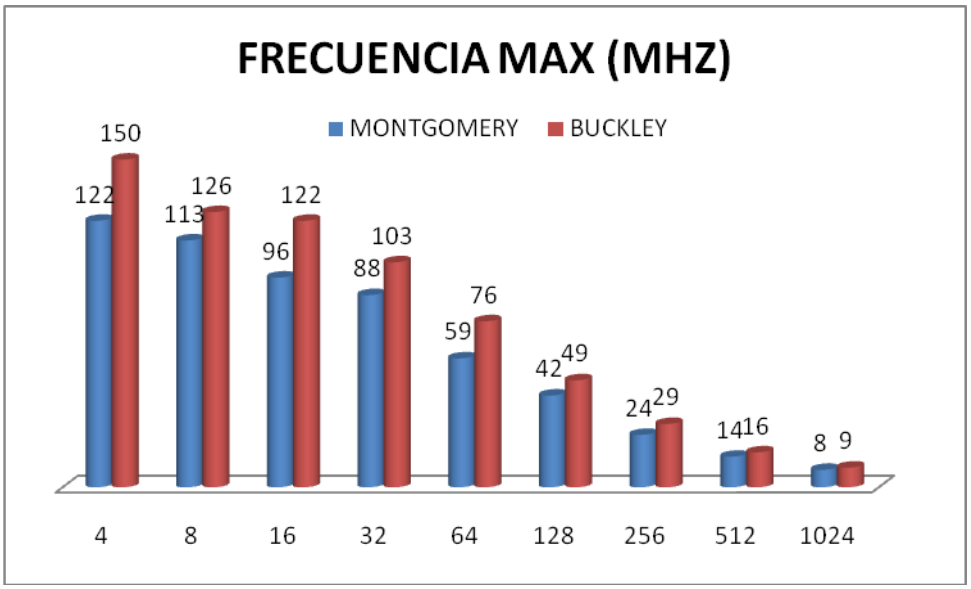


Figura 93 - FRECUENCIA MONTGOMERY-BUCKLEY

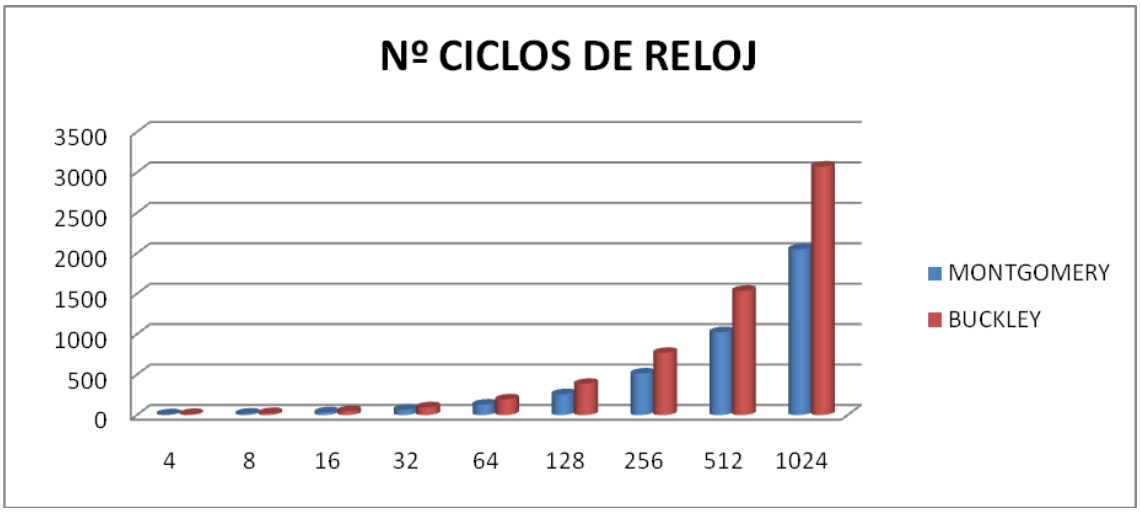


Figura 94 - CICLOS MONTGOMERY_BUCKLEY

En cuanto a la frecuencia maxima los dos algoritmos son similares siendo un poco superior el algoritmo de buckley. Por otro lado en ciclos de reloj a medida que el número de bits es mayor la diferencia es apreciable siendo mucho mas lento el de buckley que el de montgomery, por lo que para un cálculo en el cual necesitemos rapidez utilizaríamo el algoritmo de Montgomery. Para poder comparar por completo las dos implementaciones vamos a mostrar un valor que será area x nº ciclos para poder comparar estos dos parámetros a la vez.

MONTGOMERY	BUCKLEY
531	828
1955	3360
7425	9264
28730	35520
113520	137088
451292	538368
1787292	2141184
7140150	8521728
28757715	34013184

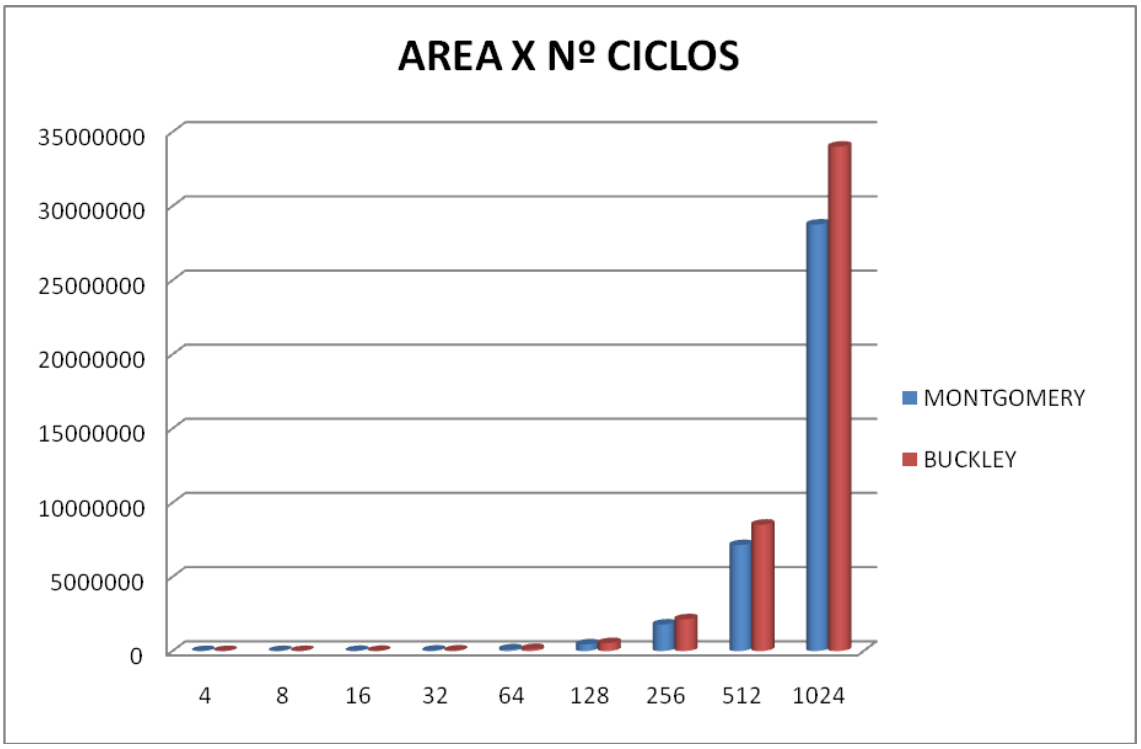


Figura 95 - AREA X CICLOS MONT_BUCKLEY

Se observa que aunque para numero de bits pequeños la diferencia no es muy grande, a medida que el numero de bits va siendo mayor se aprecia que el factor calculado es mucho menor en el de Montgomery que en el buckley, como para la criptografia se usa numeros de bits grandes (512,1024…) podemos concluir que la implementación de Montgomery es la mas adecuada para el caso el cual se está estudiando.

3.7.5 CONCLUSIÓN MULTIPLICACIÓN MODULAR

En las siguientes tablas se muestra el resultado de las cinco implementaciones para la multiplicación modular, para el caso de las tres primeras implementaciones en la reducción se le aplicará barret2. Se compará tanto el tiempo que tarda cada implementación como el área ocupada por cada circuito.

Area ocupada (LUT):

CLASICO	KARATSUBA	COMBINACIONAL	MONTGOMEY	BUCLEY
148	303	124	59	69
299	579	291	115	140
573	1102	724	225	193
1131	2211	1979	442	370
2311	4366	6129	880	714
4616	8412	20550	1756	1402
7829	14107	72639	3484	2788
19686	38978	282892	6966	5548
36093	76112	1016320	14035	11072

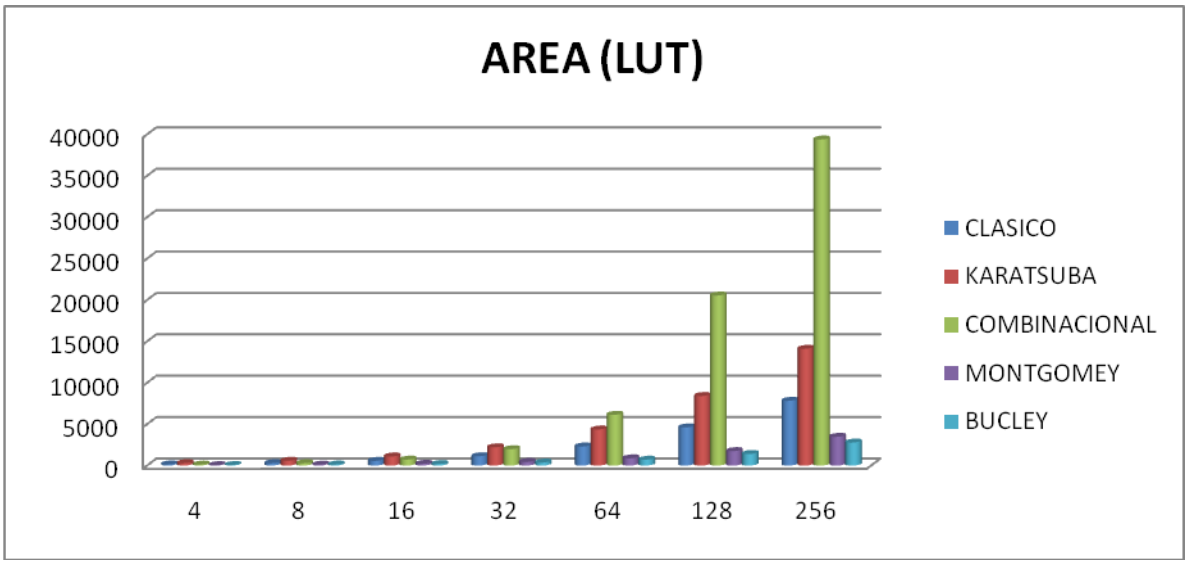


Figura 96 - COMPARACIÓN AREA TODOS

Se puede observar como el que más ocupa con una gran diferencia sobre los demas será el combinacional por ese motivo es el que descartamos para una posible implementación en la encriptación para los que se requiere numero de bits elevados. Debido que en el caso que se plantea no se dispone de un espacio grande.

En la siguiente tabla habiendo eliminado ya la implementación Combinacional, se representa el tiempo que tarda cada una de las cuatro implementaciones.

Tiempo (microsegundos):

CLASICO	KARATSUBA	MONTGOMEY	BUCLEY
0,04603175	0,035108959	0,073770492	0,08
0,11393939	0,091140412	0,150442478	0,19047619
0,25320197	0,210084034	0,34375	0,393442623
0,62593145	0,516017316	0,738636364	0,932038835
1,78519924	1,376344086	2,186440678	2,526315789
5,44922395	4,106566604	6,119047619	7,836734694
18,7733333	13,21674419	21,375	26,48275862
68,5714286	48,76190476	73,21428571	96
256	174,5454545	256,125	341,3333333

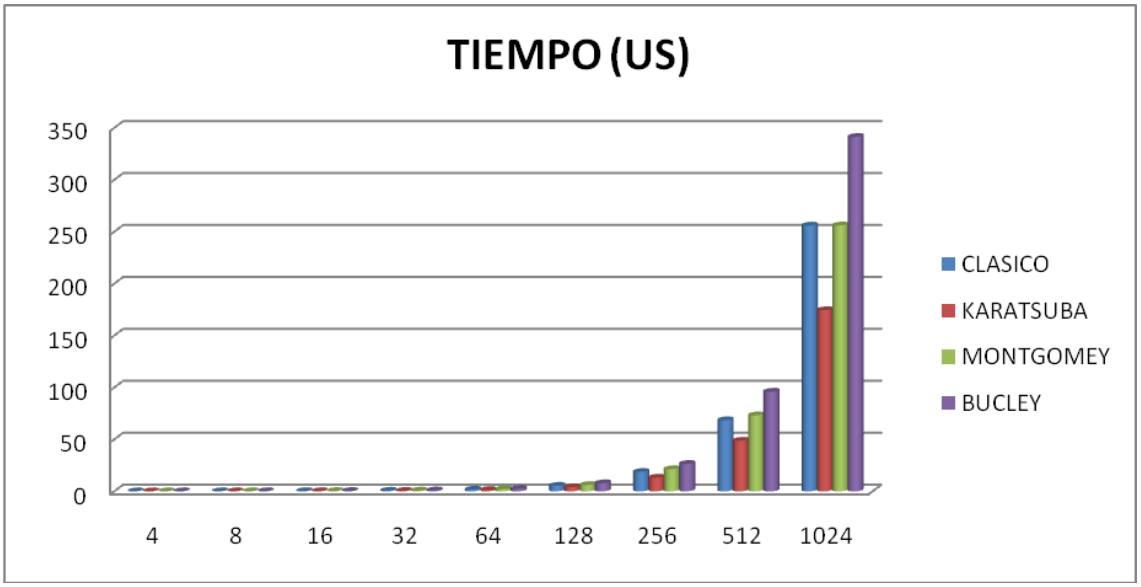


Figura 97 - COMPARACION TIEMPO TODOS

Basandonos en los datos tomados se tomará la decisión de elegir entre los dos algoritmos de Multiplicacion_modular directa elegir el algoritmo de Montgomery, y para los otros algortimos de multpiplicacion mas barret 2 se realizará a continuación un estudio más a fondo comparándolo con la implementación de Montgomery.

COMPARACIÓN KARATSUBA-CLASICO-MONTGOMERY

En la siguiente grafica representamos el tiempo que tarda cada una de las implementaciones, el tiempo se medirá en microsegundos y dependerá de su frecuencia máxima por el numero de ciclos necesarios.

CLASICO	KARATSUBA	MONTGOMERY
0,04603175	0,035108959	0,073770492
0,11393939	0,091140412	0,150442478
0,25320197	0,210084034	0,34375
0,62593145	0,516017316	0,738636364
1,78519924	1,376344086	2,186440678
5,44922395	4,106566604	6,119047619
18,7733333	13,21674419	21,375
68,5714286	48,76190476	73,21428571
256	174,5454545	256,125

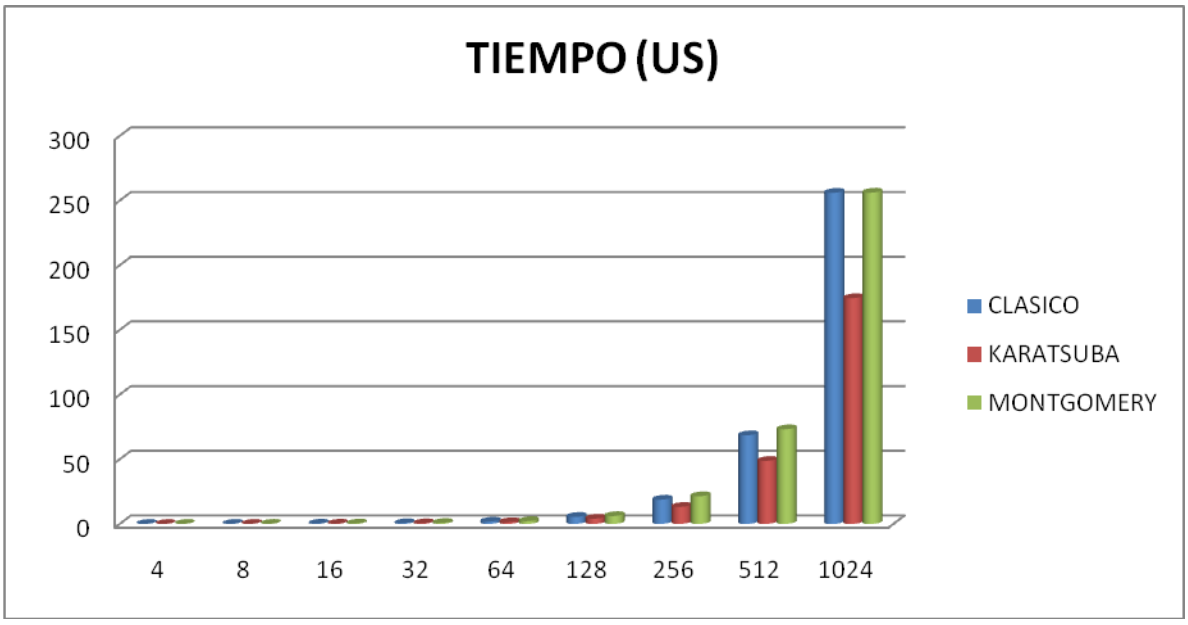


Figura 98 - TIEMPO CLASICO_KARAT_MONTGOMERY

Se comprueba como el tiempo en el circuito combinacional y el Montgomery es similar debido a que las frecuencias máximas no son apreciables y el número de ciclos es prácticamente el mismo. Por otro lado la implementación de karatsuba es la que menos tarda ya en número de ciclos tardaría 1,5N mientras que las otras dos implementaciones necesitan 2N ciclos de reloj.

A continuación vamos comparar estos metodos en un factor de Area x Tiempo. En primer lugar para los de multiplicación mas reducción hemos elegido el multiplicador clásico y el karatsuba habiendo descartado el combinacional, a estos le aplicaremos el algoritmo de reducción de barret2 descartando barret1, por tanto para obtener el numero de puertas y numero de ciclos de reloj se sumarán a los algoritmos de multiplicador clasico y karatsuba el algoritmo barret2. Para comparar los tres métodos lo haremos a través del factor Area x Tiempo, el cual observaremos cual es mas adecuado según los numeros de bits, se representará hasta 1024 bits, para criptografía lo usual suele ser de 512 o 1024 bits (números de bits altos).

CLASICO	KARATUSBA	MONTGOMERY
592	824	531
2392	3200	1955
9168	12336	7425
36192	49296	28730
147904	197696	113520
590848	770944	451292
2004224	2557952	1787292
10079232	13760000	7140150
36959232	52957184	28757715

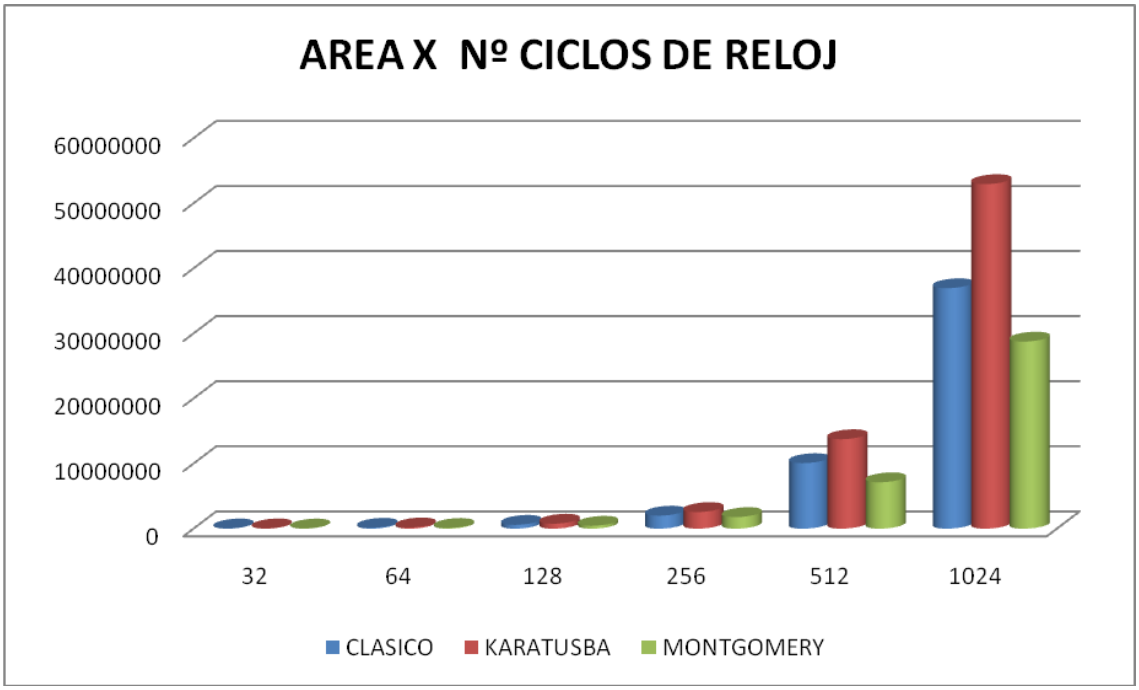


Figura 99 - AREA X CICLOS CLASICO_KARAT_MONTG

Se puede confirmar que para el algoritmo de multiplicación modular la implementación más favorable es el algoritmo de Montgomery tanto para un número bajo de bits como para un número alto de bits como es el caso de la criptografía.

Para los otros dos algoritmos se observa que para un numero de bits bajo son similares por lo tanto en ese caso elegiríamos el karatsuba ya que tarda menos, pero para un numero de bits elevado el mas adecuado seria el clasico como se refleja en la grafica.

En condiciones de que para cada implementación pudiésemos poner la frecuencia máxima de cada una, si lo que se quiere es una compensación entre área y tiempo la implementación más adecuada para número de bits elevados sería la implementación basada en el algoritmo de Montgomery.

GENERADOR DE NÚMEROS ALEATORIOS

[BLUM BLUM AND SHUB]

ALGORITMO

4. IMPLEMENTACIÓN GENERADOR DE NUMEROS ALEATORIOS

Una vez se ha hecho el estudio de los diferentes algoritmos estudiados se realiza la implementación del generador de números aleatorios, el cual servirá en un futuro para encriptar datos. El algoritmo que usaremos para generar los numero aleatorios sera el Blum Blum and Shub. Este algoritmo fue propuesto por Lenore Blum, Manuel Blum y Michael Shub en 1986.

El algoritmo es el siguiente:

$X_{n+1} = X_n^{\,2} \bmod M$

Donde $M=pq$ es el producto de dos números primos muy grandes p y q. En cada paso del algoritmo, se obtiene un resultado para X_n, el resultado es por lo general o bien el bit de paridad de X_n ó uno ó más de los bits menos significativos de X_n. Para calcular el X_0 se utilizará un valor denominado semilla el cual será el X_{-1} de la sucesión, para posteriormente calcular todos los sucesivos valores de la sucesión.

Los dos números primos, p y q, deben ser ambos congruentes a 3 (mod 4) (esto asegura que cada residuo cuadrático posee una raíz cuadrada que también es un residuo cuadrático) y mcd ($\varphi(p-1)$, $\varphi(q-1)$) debe ser pequeña (esto hace que la longitud del ciclo sea extensa).

Una característica interesante del generador BBS, es la posibilidad de calcular todo valor X_i en forma directa, el algoritmo para calcular ese valor será el siguiente:

$$x_i = \left(x_0^{2^i \bmod (p-1)(q-1)}\right) \bmod M.$$

Para poder realizar esta operación se necesitará la exponenciación modular que se basa en la multiplicación modular, más adelante se realizará una implementación en vhdl usando la exponenciación modular para poder realizar el algoritmo.

Para implementar este algoritmo se ha usado las implementaciones de los siguientes algoritmos de multiplicación modular: clásico+barret2, karatsuba+barret2, Montgomery. Una vez hechas estas implementaciones se realizará un estudio entre las tres para el área ocupada y el tiempo que tarda.

Se ha considerado dos niveles de seguridad:

a) Nivel bajo los cuales se utiliza 2^{32} por tanto se utilizará un modulo de 160 bits.
b) Un nivel de seguridad alto para 2^{64} para el cual se necesitará un modulo de 512 bits.

A continuación se muestra la implementaciones para el generador de numeros primos usando el algoritmo Blum Blum and Shub.

GENERADOR DE NÚMEROS ALEATORIOS

[BLUM BLUM AND SHUB => MONTGOMERY]

IMPLEMENTACIÓN – SIMULACIÓN – SÍNTESIS – RESULADOS

4.2 BLUM BLUM AND SHUB =>MONTGOMERY

4.2.2 IMPLEMENTACIÓN EN VHDL

A continuacion se muestra la entidad implementada en vhdl:

```
PORT(

        DATO_INICIO: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        M: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        S: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        INICIO: IN STD_LOGIC;
        FIN: OUT STD_LOGIC;
        CLK,RESET: IN STD_LOGIC;
        FIN_TERMINOi: OUT STD_LOGIC
        );
```

Figura 100 - ENTIDAD BLUM BLUM & SHUB MONTGOMERY

El bit Fin_Sucesion muestra el fin de cada termino de la sucesión una vez que se ha obtenido el numero, en cambio el bit Fin indica el final de la sucesión. Activando el bit Inicio empeiza a generar numero aleatorios.

En la siguiente parte del codigo se muestra la parte de la implementación:

Codigo=>

```
        --PARTE SINCRONA
        ……
    IF ACTUAL=REPOSO THEN
                        CONTADOR<=0;

                        NUMERO_SUCESION<=3;
```

```
                        ELSIF ACTUAL=ESTADO1 THEN
                           R1<=DATO_INICIO;
                           R2<=DATO_INICIO;


                        ELSIF ACTUAL=ESTADO2_2 THEN
                            R1<=R_FINAL;
                            R2<=R_FINAL;
                            CONTADOR<=CONTADOR+1;
                            S<=R_FINAL;
                        ELSIF ACTUAL=ESTADO3 THEN
                           S<=R_FINAL;
                        ELSIF ACTUAL=E_FINAL THEN

--PARTE MAQUINA DE ESTADOS
WHEN ESTADO2=>
                    FIN_TERMINOi<='0';
                    FIN<='0';
                    INICIO_M<='1';
                    IF FIN_M='1' THEN

                       IF CONTADOR=NUMERO_SUCESION THEN
                          SIGUIENTE<=ESTADO3;
                          ELSE SIGUIENTE<=ESTADO2_2;
                       END IF;
                    ELSE
                      SIGUIENTE<=ACTUAL;
                   END IF;


                 WHEN ESTADO2_2=>
                    FIN_TERMINOi<='1';
                    FIN<='0';
                    INICIO_M<='0';
                    SIGUIENTE<=ESTADO2;
```

En numero total de terminos que queremos calcular, sera el numero total de numeros aleatorios que queremos calcular, este numero de terminos esta condicionado por el valor del contador que sera una constante la cual podemos modificar.

En el codigo se observa que en el estado 2 se realizara montgomery y pasará al estado 2_2, estos dos estados se realizará tantas veces como numeros aleatorios queramos calcular. Cuando llegamos al ultimo termino de la sucesión se pasará directamente al estado 3 donde terminará de generar numeros aleatorios.

El diagrama de flujo se muestra a continuación:

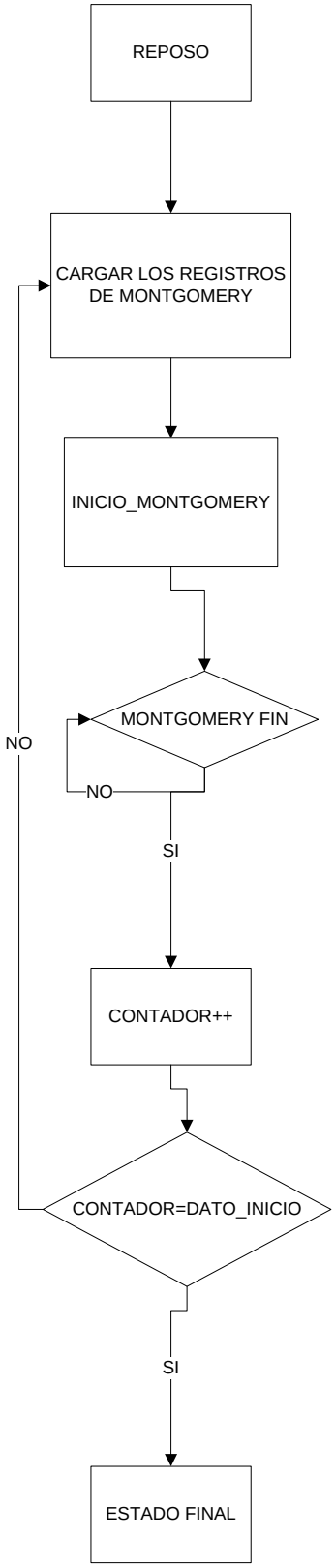


Figura 101 - D.ESTADOS BB&SHUB MONTGOMERY

4.2.3 SIMULACION

En la siguiente simulación se mostrará detalladamente los pasos que se realizará en el circuito. Hemos tomado como numeros primos p = 377 n = 233 => m = 87841 , y como termino raiz será X_{-1}= 555

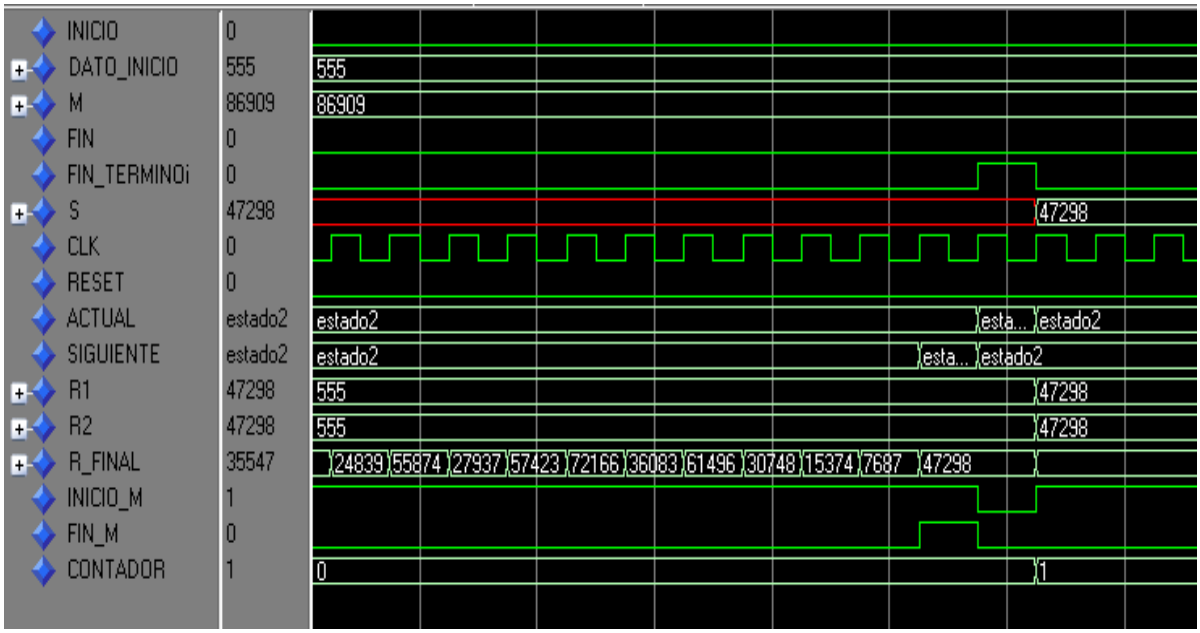


Figura 102 - SIMULACION BBS MONTGOMERY 1

Se puede observar como cada vez que se quiere realizar Montgomery se activa el bit Inicio_M, y el circuito de Montgomery cada vez que obtiene el resultado activa el bit Fin_M. El contador muestra el termino de la sucesión que queremos calcular.

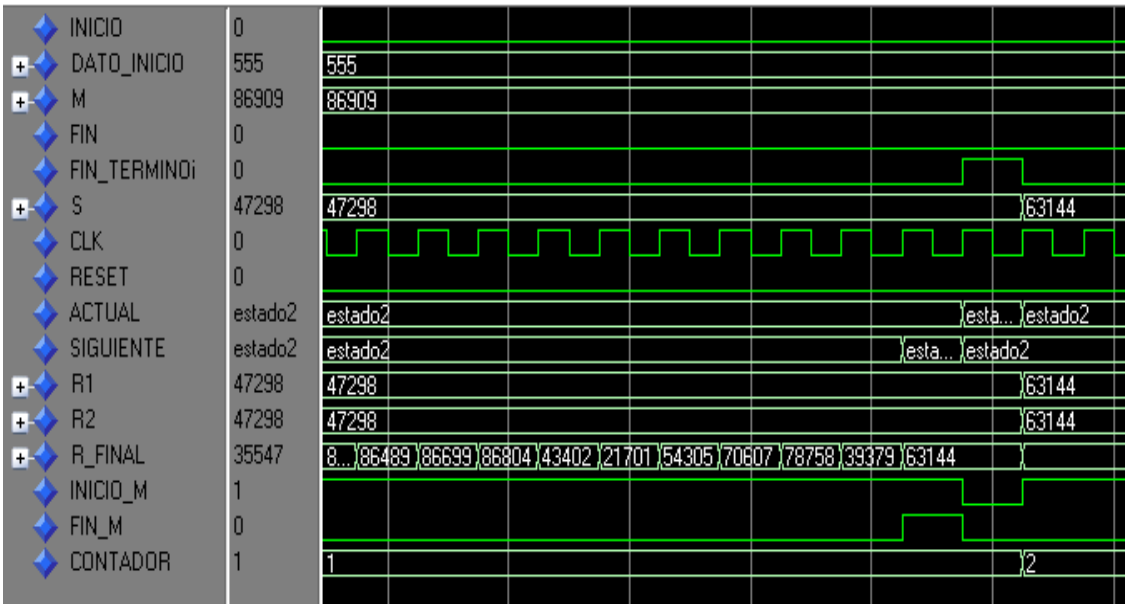


Figura 103 - SIMULACION BB&S MONTGOMERY 2

Se ira incrementando el contador hasta que este sea igual al numero de sucesión que indicará que ya ha calculado los numeros X de la sucesión, y por tanto pasará al estado 3.

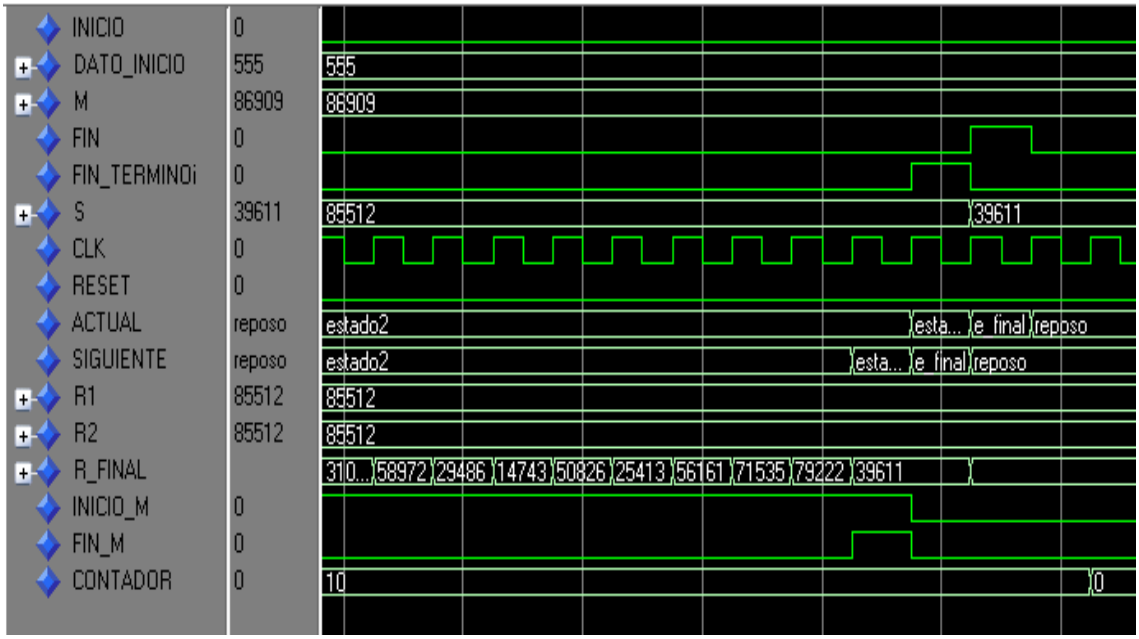


Figura 104 - SIMULACION BB&S MONTGOMERY 3

Mostramos la simulación entera, con estos valores será con los cuales compararemos las otras 2 implementaciones.

Simulacion 1 para 10 terminos. N = 233 , P = 377 , M = 87841 , X_{-1} = 555

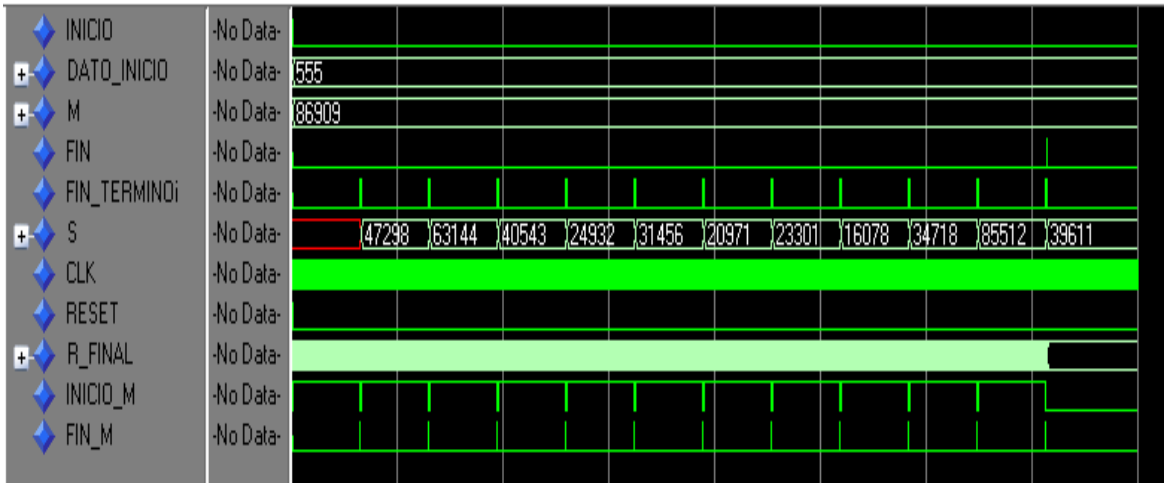


Figura 105 - SIMULACION BB&S MONTGOMERY 4

Comprobamos estos resultados obtenidos con la siguiente tabla para ver si son correctos. En la ultima columna se representa la partidad de cada termino en el caso de que una vez terminado con el algoritmo de Blum BLum and Shub se quiere obtener un resultado a partir de la paridad de estos terminos. Tiempo tarda = 7129 us

X(i)	n	p	M=n*p	PRODUCTO	X_{-1}=555	PARIDAD	
0	233	373	86909	308025	47298	0	
1	233	373	86909	2237100804	63144	0	
2	233	373	86909	3987164736	40543	1	
3	233	373	86909	1643734849	24932	0	
4	233	373	86909	621604624	31456	0	
5	233	373	86909	989479936	20971	1	
6	233	373	86909	439782841	23301	1	
7	233	373	86909	542936601	16078	0	
8	233	373	86909	258502084	34718	0	
9	233	373	86909	1205339524	85512	0	
10	233	373	86909	7312302144	39611	1	
						10001100100	1124

Simulación 2 para 10 terminos.

N = 557 , P = 113 , M = 62941 , X_{-1} =357

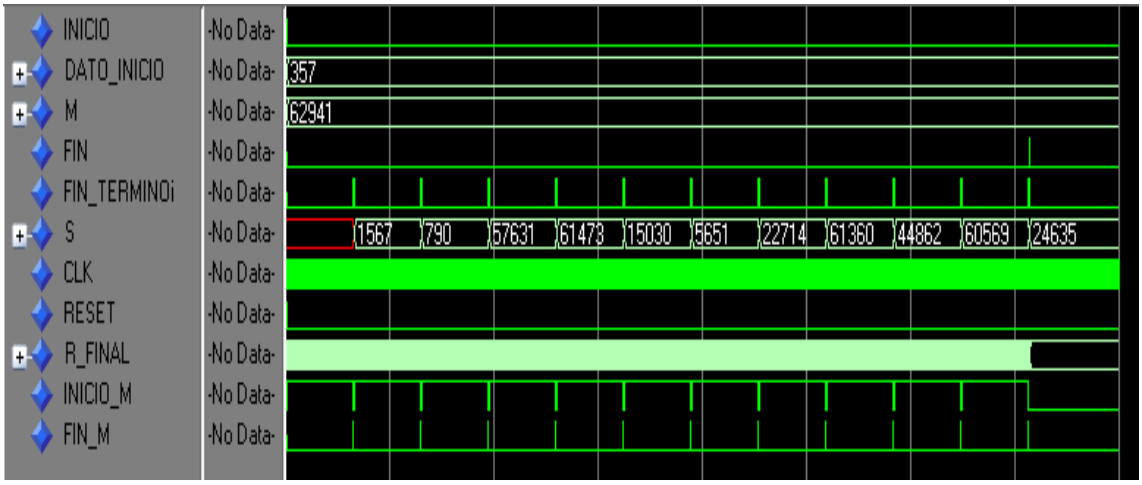


Figura 106 - SIMULACION BB&S MONTGOMERY 5

Con la siguiente tabla comprobamos que la simulación es correcta.

X(i)	n	p	M=n*p	PRODUCTO	X_{-1}=357	PARIDAD	
0	557	113	62941	127449	1567	1	
1	557	113	62941	2455489	790	0	
2	557	113	62941	624100	57631	1	
3	557	113	62941	3321332161	61473	1	
4	557	113	62941	3778929729	15030	0	
5	557	113	62941	225900900	5651	1	
6	557	113	62941	31933801	22714	0	
7	557	113	62941	515925796	61360	0	
8	557	113	62941	3765049600	44862	0	
9	557	113	62941	2012599044	60569	1	
10	557	113	62941	3668603761	24635	1	
						11000011001	1561

4.2.4 SINTESIS

Se muestra la sintesis hecha con el programa ISE, para 160 bit y 512 bit.

BLUM_MONTGOMERY
N=160
Frecuencia Máxima = 35 MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1602	768	208%
Number of Slice Flip Flops	1152	1536	75%
Number of 4 input LUTs	3052	1536	198%
Number of bonded IOBs	485	124	391%
Number of GCLKs	1	8	12%

N=512
Frecuencia Máxima = 14Mhz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	5009	768	652%
Number of Slice Flip Flops	3649	1536	237%
Number of 4 input LUTs	9553	1536	621%
Number of bonded IOBs	1541	124	1242%
Number of GCLKs	1	8	12%

4.2.5 RESULTADOS

Se muestra en las siguientes graficas el area ocupada y el tiempo que tarda

N º BITS	AREA (LUT)	TIEMPO (us)
160	3052	35630
512	9953	112000

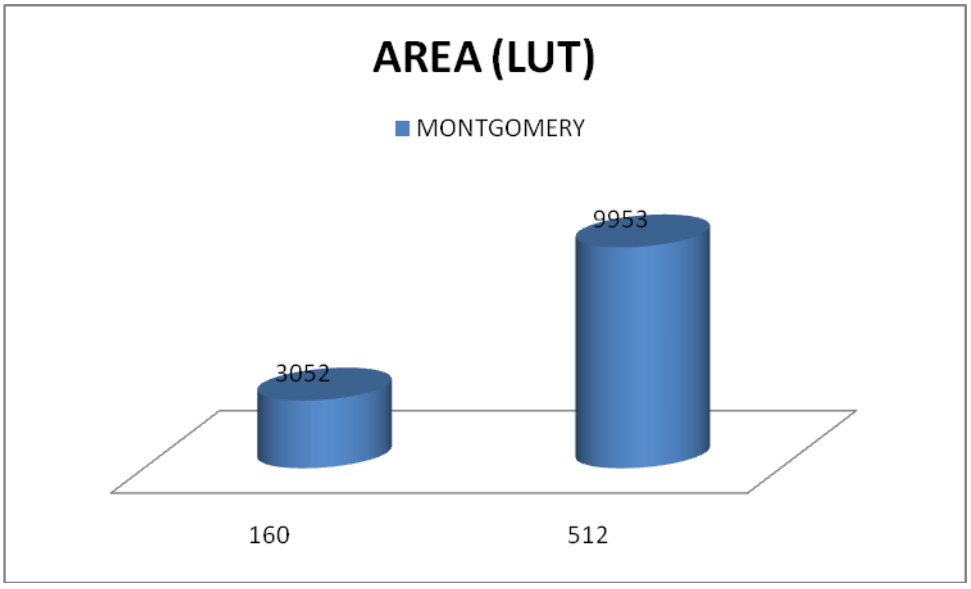


Figura 107 - AREA BB&S MONTGOMERY

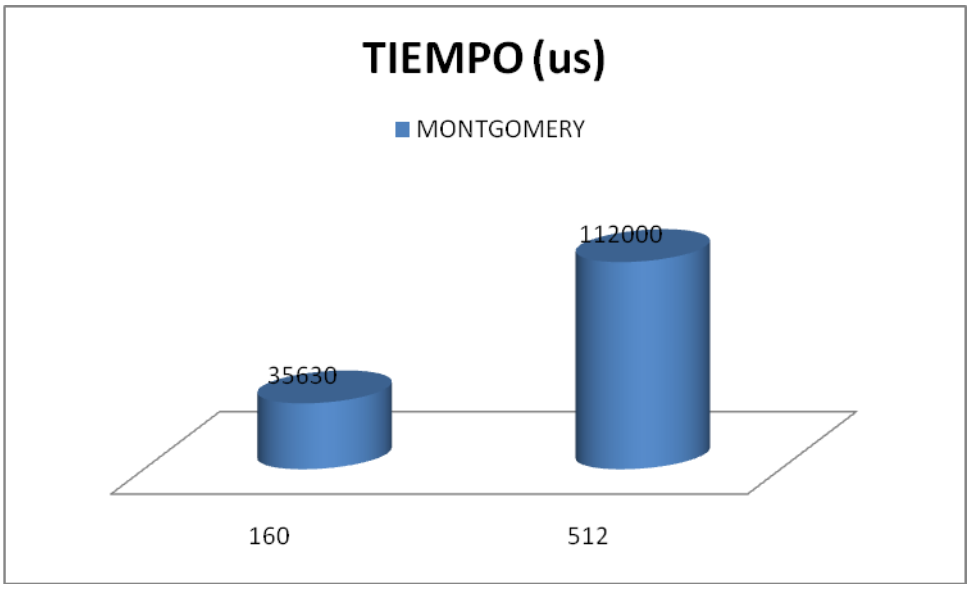


Figura 108 - TIEMPO BB&S MONTGOMERY

GENERADOR DE NÚMEROS ALEATORIOS

[BLUM BLUM AND SHUB => KARATSUBA - OFMAN]

IMPLEMENTACIÓN – SIMULACIÓN – SÍNTESIS – RESULADOS

4.3 BLUM BLUM AND SHUB (KARATSUBA)

4.3.2 IMPLEMENTACIÓN EN VHDL.

En cuanto a la entidad del circuito se ha realizado igual que en la anterior implementación.

Entidad =>

```
entity blum_karat IS
    GENERIC(N:INTEGER:=160);
    PORT(

        DATO_INICIO: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        M: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        S: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        INICIO: IN STD_LOGIC;
        FIN: OUT STD_LOGIC;
        CLK,RESET: IN STD_LOGIC;
        FIN_TERMINOi: OUT STD_LOGIC
        );
    END blum_KARAT;
```

Es la misma entidad que en caso de Montgomery

Figura 109 - ENTIDAD BB&S KARATSUBA

Codigo del algoritmo=>

- PROCESO DE MAQUINA DE ESTADOS

```
……….
WHEN ESTADO1=>
                FIN_TERMINOi<='0';
                FIN<='0';
                INICIO_K<='1';
                INICIO_B<='0';
                SIGUIENTE<=ESTADO2;

              WHEN ESTADO2=>
                FIN_TERMINOi<='0';
                FIN<='0';
                INICIO_K<='1';
                INICIO_B<='0';

                IF FIN_K='1' THEN

                   SIGUIENTE<=ESTADO2_2;
                ELSE
                  SIGUIENTE<=ACTUAL;
               END IF;


              WHEN ESTADO2_2=> --CARGAR LOS REGITROS EN BARRET
                FIN_TERMINOi<='0';
                FIN<='0';
                INICIO_K<='0';
                INICIO_B<='0';
                SIGUIENTE<=ESTADO2_3;

              WHEN ESTADO2_3=>    -- SE INICIA BARRET
                FIN_TERMINOi<='0';
                INICIO_K<='0';
                INICIO_B<='1';
                FIN<='0';

                SIGUIENTE<=ESTADO2_4;

              WHEN ESTADO2_4=>     --SE HACE BARRET
                 INICIO_K<='0';
                 INICIO_B<='0';
                 FIN<='0';
                 FIN_TERMINOi<='0';

                 IF FIN_B='1' THEN
                   SIGUIENTE<=ESTADO2_5;
                   ELSE SIGUIENTE<=ACTUAL;
               END IF;

             WHEN ESTADO2_5=>
                FIN_TERMINOi<='1';
                INICIO_K<='0';
                INICIO_b<='0';
                 FIN<='0';
```

```
IF CONTADOR=NUMERO_SUCESION+1 THEN
                    SIGUIENTE<=ESTADO3;
                    ELSE SIGUIENTE<=ESTADO2;
                  END IF;

            WHEN ESTADO3=>
               FIN_TERMINOi<='1';
               FIN<='0';
               INICIO_K<='0';
               INICIO_B<='0';
               SIGUIENTE<=E_FINAL;

            WHEN E_FINAL=>
               FIN_TERMINOi<='0';
               INICIO_K<='0';
               INICIO_B<='0';
               FIN<='1';
               SIGUIENTE<=REPOSO;
            END CASE;
         END PROCESS;


      -  PARTE SECUENCIAL

  ……..

  ELSIF ACTUAL=ESTADO1 THEN
                    R1<=DATO_INICIO;
                    R2<=DATO_INICIO;


                ELSIF ACTUAL=ESTADO2_2 THEN
                    PRODUC_BARRET<=R_FINAL1;







                ELSIF ACTUAL=ESTADO2_4 THEN
                   R1<=R_FINALB;
                   R2<=R_FINALB;
                  IF FIN_B='1' THEN CONTADOR<=CONTADOR+1;
                   S<=R_FINALB;
                   END IF;
……………….
```

Se observa como en el estado 2 se activa el bit de Inicio_K, una vez se obtenga la multiplicación a través de Karatsuba se activa el bit Fin_K, y se activa el bit Inicio_B, con el cual comenzará a realizar el modulo de la multiplicación obtenida en la multiplicación por karatsuba. Esta parte de la multiplicación y reducción se realizará tantas veces como numeros aletorios se quiera obtener. Igual que en la implementación anterior a través del contador se obtiene la cantidad de numeros aleatorios que se quiera obtener, comparandolo con el valor de Dato_Inicio.

El diagrama de flujo es el siguiente:

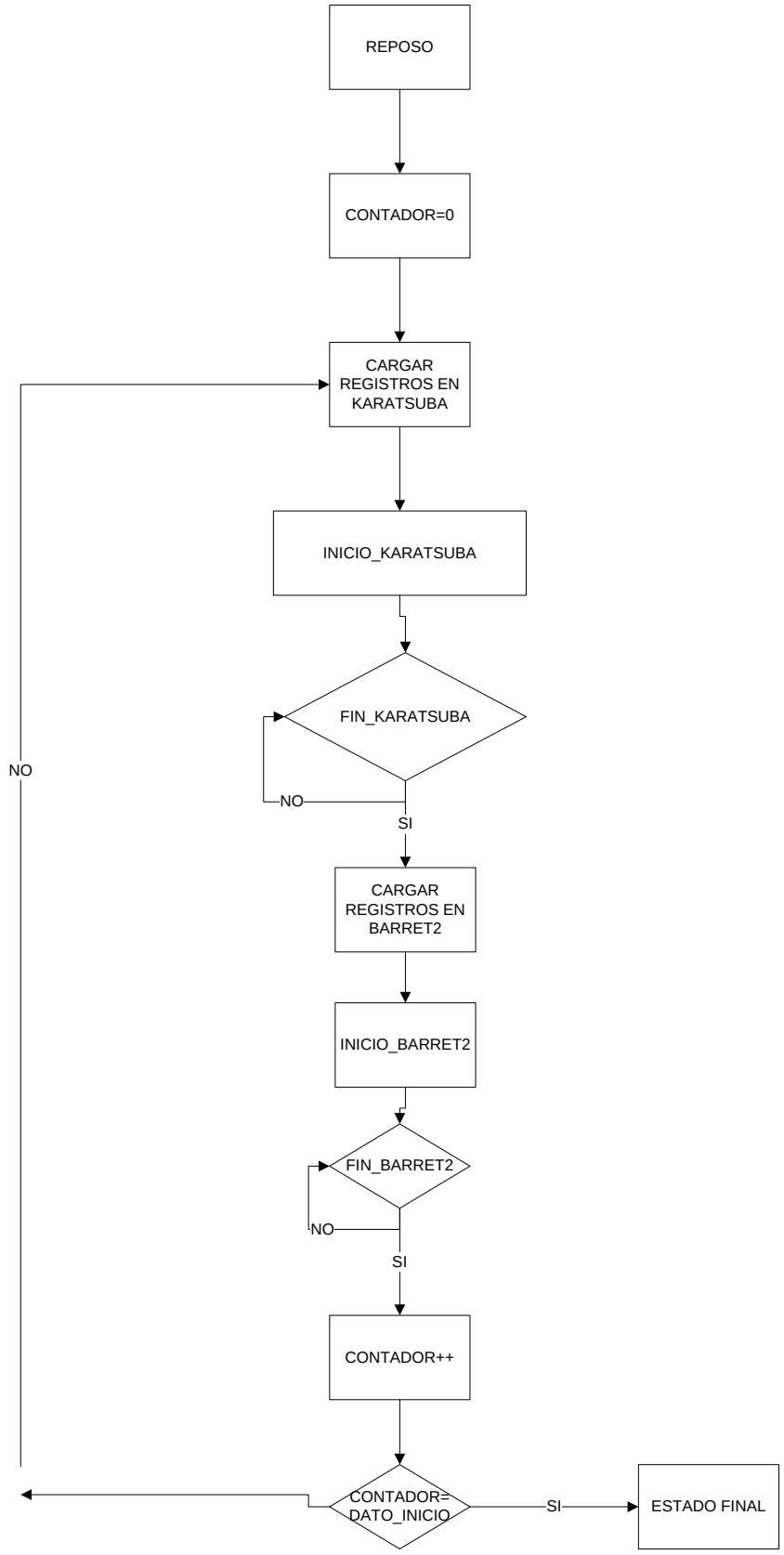


Figura 110 - D.ESTADOS BB&S KARATSUBA

4.3.3 SIMULACIÓN

En la siguiente simulación se comprobara este algoritmo para realizar X numeros aleatorios.

N = 457 , P = 157 , M = 71749 , X_{-1} = 728

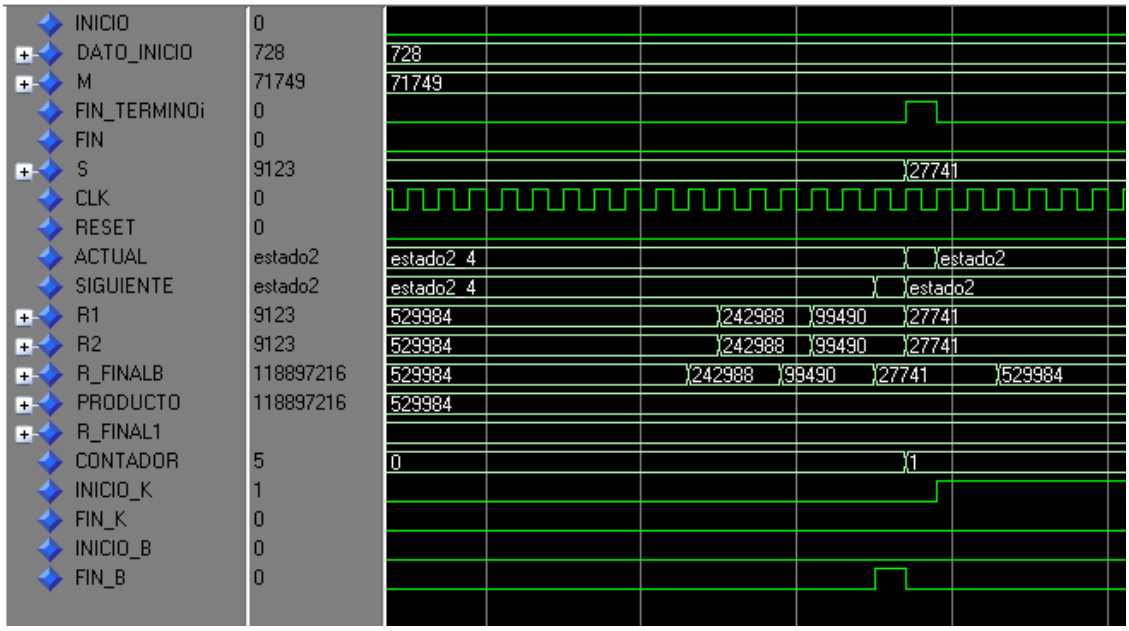


Figura 111 - SIMULACION BB&S KARATSUBA 1

Se observa como una vez que se activa el bit de Inicio_K empieza a calcular la multiplicación, cuando el bit Fin_K se activa indica que la multiplicación ha terminado como se puede obervar en el registro R_final que por ejemplo tiene el resultado de multiplicar X * Y. En ese momento se activa el bit Inicio_B en el cual dará lugar que que el bloque de Barret2 empiece a calcular el modulo.

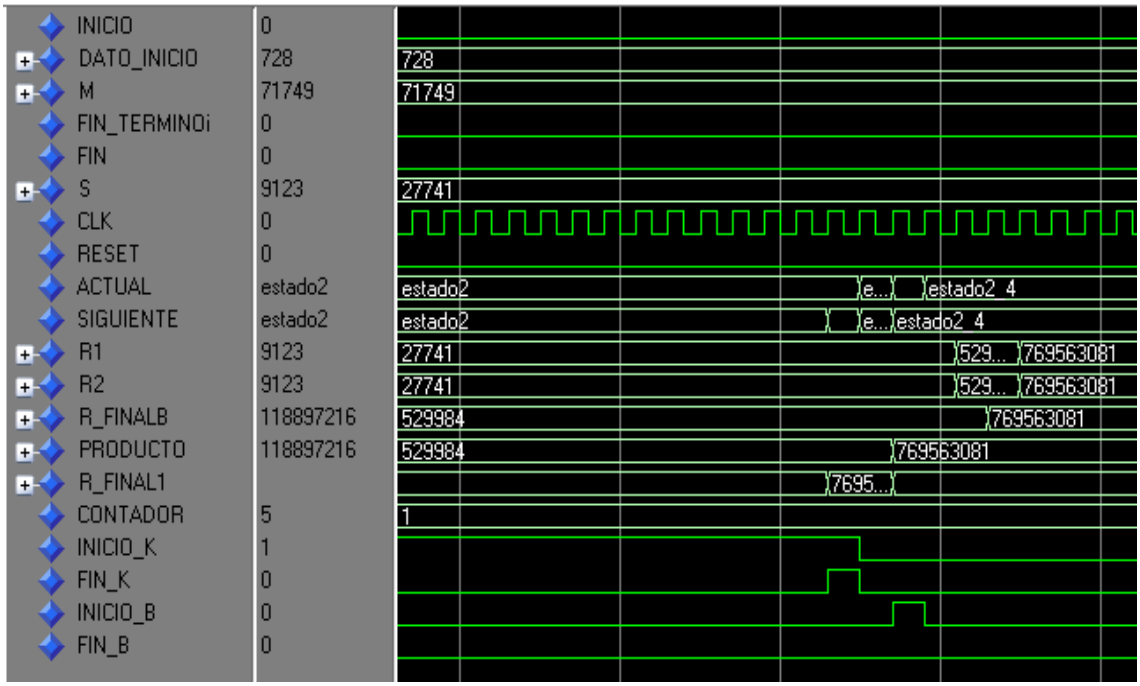


Figura 112 - SIMULACION BB&S KARATSUBA 2

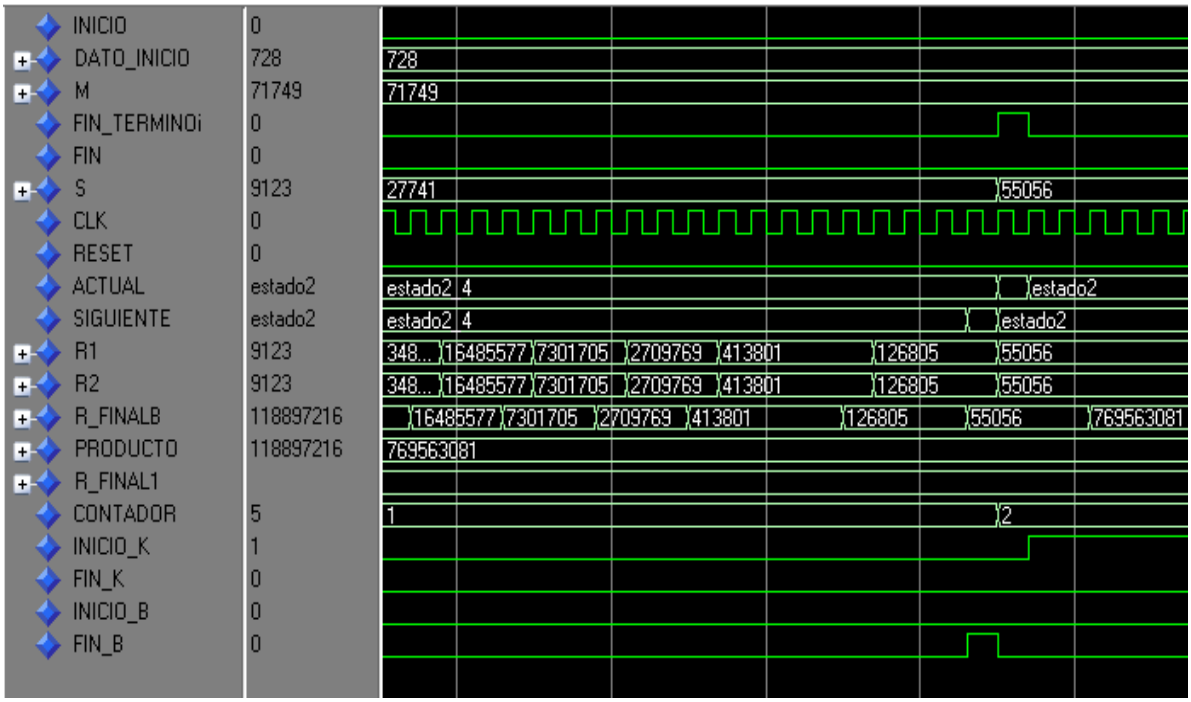


Figura 113 - SIMULACION BB&S KARATSUBA 3

Al activarse el bit de Fin_termino(i) se ha calculado el termino de la sucesión $X_{1.}$

A continuación se muestra la simulación completa para los valores siguientes:

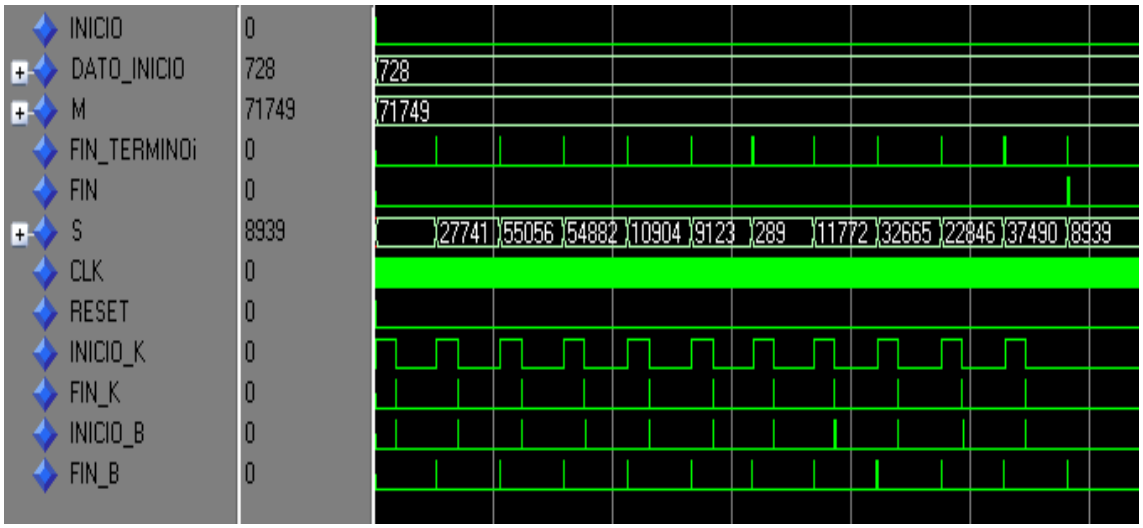


Figura 114 - SIMULACION BB&S KARATSUBA 4

En la siguiente tabla comprobamos si se ha calculado bien los numeros aleatorios:

X(i)	n	p	M=n*p	PRODUCTO	X_{-1}=728	PARIDAD	
0	457	157	71749	529984	27741	1	
1	457	157	71749	769563081	55056	0	
2	457	157	71749	3031163136	54882	0	
3	457	157	71749	3012033924	10904	0	
4	457	157	71749	118897216	9123	1	
5	457	157	71749	83229129	289	1	
6	457	157	71749	83521	11772	0	
7	457	157	71749	138579984	32665	1	
8	457	157	71749	1067002225	22846	0	
9	457	157	71749	521939716	37490	0	
10	457	157	71749	1405500100	8939	1	
						10010110001	1201

Cada vez que se ha calculado un numero de sucesión se activa el bit Fin_Sucesióni, una vez que termina el algoritmo entero se activa el bit FIN.

Realizamos otra simulación con los mismo valores que en la segunda simulación de la implementación con Montgomery.

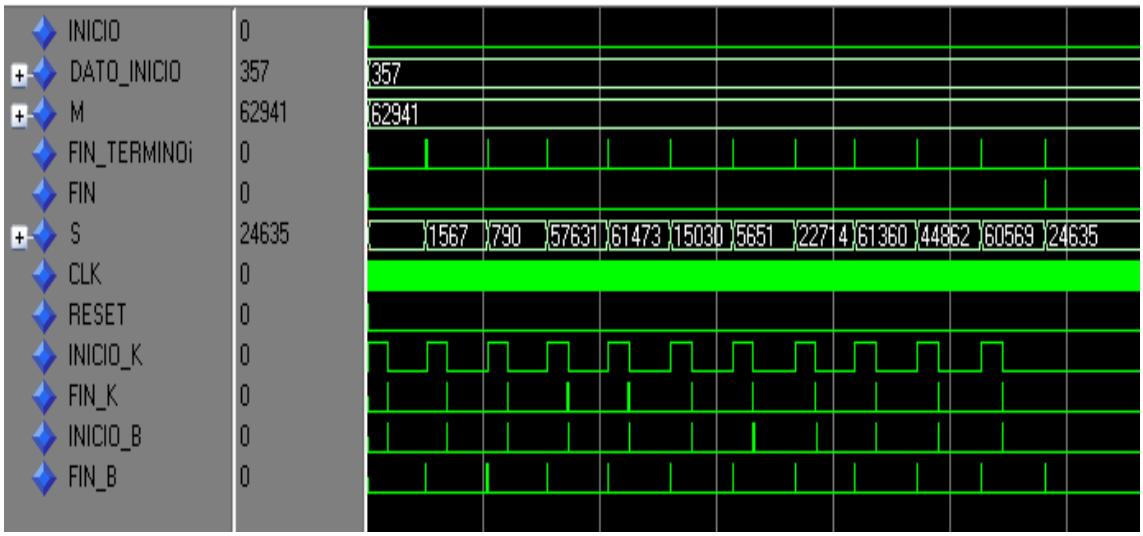


Figura 115 SIMULACION BB&S KARATSUBA 5

X(i)	n	p	M=n*p	PRODUCTO	X_{-1}=357	PARIDAD	
0	557	113	62941	127449	1567	1	
1	557	113	62941	2455489	790	0	
2	557	113	62941	624100	57631	1	
3	557	113	62941	3321332161	61473	1	
4	557	113	62941	3778929729	15030	0	
5	557	113	62941	225900900	5651	1	
6	557	113	62941	31933801	22714	0	
7	557	113	62941	515925796	61360	0	
8	557	113	62941	3765049600	44862	0	
9	557	113	62941	2012599044	60569	1	
10	557	113	62941	3668603761	24635	1	
						11000011001	1561

4.3.4 SÍNTESIS

A continuación se muestra la sintesis realizada ha esta implementación realizada para los dos casos de seguridad: baja => 160 bit; media => 512 bit.

N=512
Frecuencia Máxima = 14MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	20892	768	2720%
Number of Slice Flip Flops	16988	1536	1105%
Number of 4 input LUTs	40052	1536	2607%
Number of bonded IOBs	1541	124	1242%
Number of GCLKs	1	8	12%

N=160
Frecuencia Máxima = 37MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	5504	768	716%
Number of Slice Flip Flops	5305	1536	345%
Number of 4 input LUTs	10435	1536	679%
Number of bonded IOBs	485	124	391%
Number of GCLKs	1	8	12%

4.3.5 RESULTADOS

En la siguiente grafica se muestra el area ocupada, y el tiempo que tarda, la frecuencia tomada será de 100 Khz.

N º BITS	AREA (LUT)	TIEMPO (us)
160	10433	29000
512	40052	92800

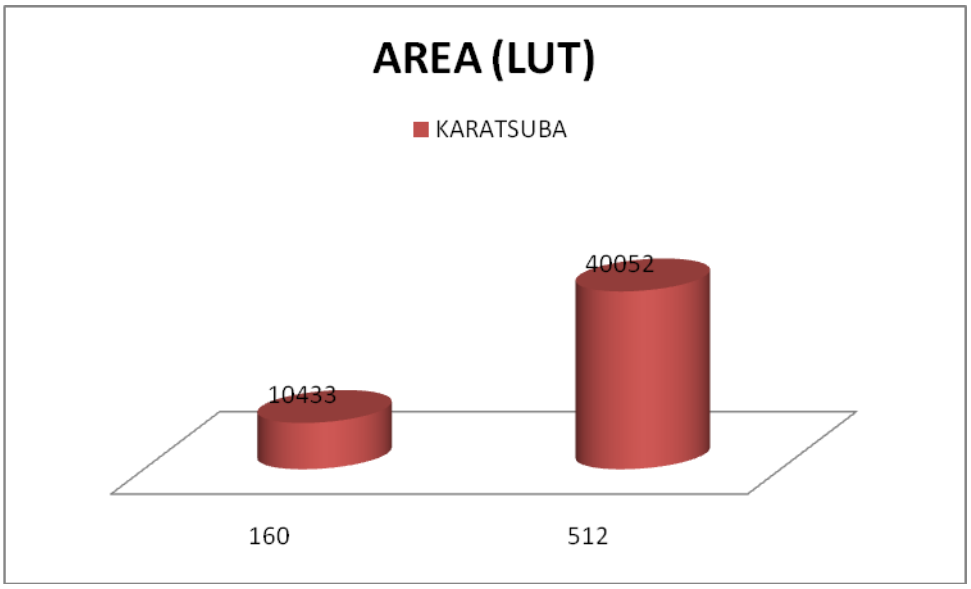


Figura 116 - AREA BB&S KARATSUBA

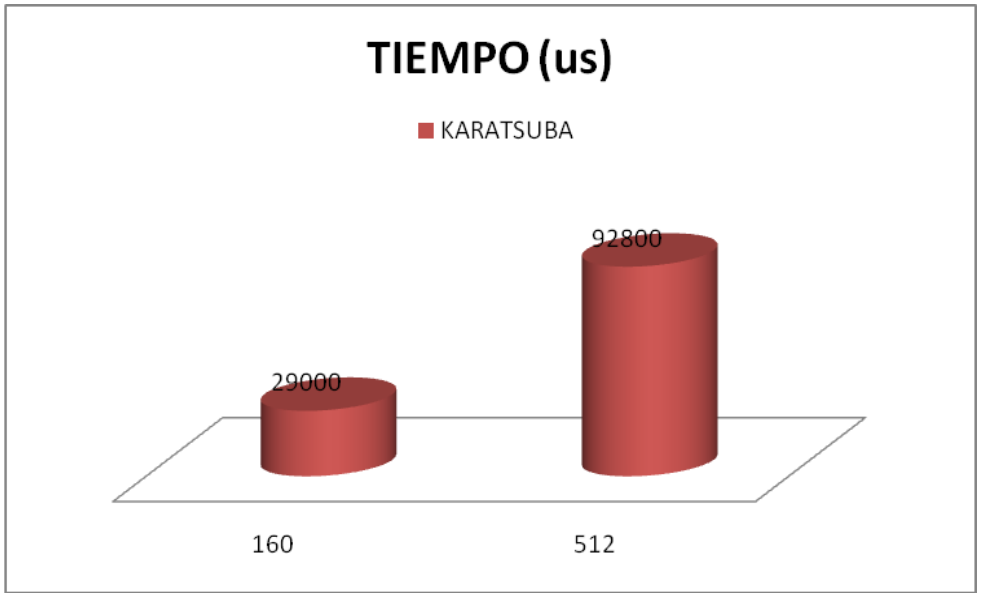


Figura 117 - TIEMPO BB&S KARATSUBA

GENERADOR DE NÚMEROS ALEATORIOS

[BLUM BLUM AND SHUB => CLÁSICO]

IMPLEMENTACIÓN – SIMULACIÓN – SÍNTESIS – RESULADOS

4.4 BLUM BLUM AND SHUB (Clasico)

4.4.2 IMPLEMENTACIÓN EN VHDL

La entidad del circuito es igual que en las anteriores implementaciones la cual se muestra acontinuación:

```
entity blum_clasico IS
    GENERIC(N:INTEGER:=160);
    PORT(

        DATO_INICIO: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        M: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        S: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        INICIO: IN STD_LOGIC;
        FIN: OUT STD_LOGIC;
        CLK,RESET: IN STD_LOGIC
        );
    END blum_clasico;
```

Figura 118 - ENTIDAD BB&S CLASICO

El control entre el bloque del multiplicador clasico y el barret2 se realiza de la misma manera que en la implementación con Karatsuba.

Codigo =>

```
-   PARTE DEL CODIGO DE LA MAQUINA DE ESTADOS
    ……..
    WHEN ESTADO1=>
                    FIN_TERMINOi<='0';
                    FIN<='0';
                    INICIO_M<='1';
                    INICIO_B<='0';
                    SIGUIENTE<=ESTADO2;
```

```
WHEN ESTADO2=>
                    FIN_TERMINOi<='0';
                    FIN<='0';
                    INICIO_M<='1';
                    INICIO_B<='0';

                    IF FIN_M='1' THEN

                       SIGUIENTE<=ESTADO2_2;
                    ELSE
                      SIGUIENTE<=ACTUAL;
                   END IF;


                  WHEN ESTADO2_2=> --CARGAR LOS REGITROS EN BARRET
                    FIN_TERMINOi<='0';
                    FIN<='0';
                    INICIO_M<='0';
                    INICIO_B<='0';
                    SIGUIENTE<=ESTADO2_3;

                  WHEN ESTADO2_3=>    -- SE INICIA BARRET
                    FIN_TERMINOi<='0';
                    FIN<='0';
                    INICIO_M<='0';
                    INICIO_B<='1';
                    SIGUIENTE<=ESTADO2_4;

                  WHEN ESTADO2_4=>     --SE HACE BARRET
                     FIN_TERMINOi<='0';
                     INICIO_M<='0';
                     INICIO_B<='0';
                     FIN<='0';

                     IF FIN_B='1' THEN
                       SIGUIENTE<=ESTADO2_5;
                       ELSE SIGUIENTE<=ACTUAL;
                    END IF;

                   WHEN ESTADO2_5=>
                     FIN_TERMINOi<='1';
                     INICIO_M<='0';
                     INICIO_b<='0';

                        IF CONTADOR=NUMERO_SUCESION+1 THEN
                           SIGUIENTE<=ESTADO3;
                           ELSE SIGUIENTE<=ESTADO2;
                        END IF;

                  WHEN ESTADO3=>
                    FIN_TERMINOi<='1';
                    FIN<='0';
                    INICIO_M<='0';
                    INICIO_B<='0';
                    SIGUIENTE<=E_FINAL;

                  WHEN E_FINAL=>
                    FIN_TERMINOi<='0';
                    INICIO_M<='0';
                    INICIO_B<='0';
                    FIN<='1';
                    SIGUIENTE<=REPOSO;
                  END CASE;
                END PROCESS;
```

```
--PARTE SECUENCIAL

 ELSIF ACTUAL=ESTADO1 THEN
                          R1<=DATO_INICIO;
                          R2<=DATO_INICIO;


                       ELSIF ACTUAL=ESTADO2_2 THEN
                           PRODUC_BARRET<=R_FINAL1;









                       ELSIF ACTUAL=ESTADO2_4 THEN
                          R1<=R_FINALB;
                          R2<=R_FINALB;
                        IF FIN_B='1' THEN CONTADOR<=CONTADOR+1;
                         S<=R_FINALB;
                         END IF;

                       ELSIF ACTUAL=ESTADO2_5 THEN

                       ELSIF ACTUAL=ESTADO3 THEN

                       ELSIF ACTUAL=E_FINAL THEN
```

La parte de control entre los modulos de multiplicación y reducción se realizan de la misma forma que en la implementación con Karatsuba.
La cantidad de numeros aleatorios se realiza a través de la comparación del contador el numero de sucesión el cual indica los numeros aleatorios que se quiera calcular, igual en los casos anteriores.

El diagrama de flujo es el que se muestra a continuación:

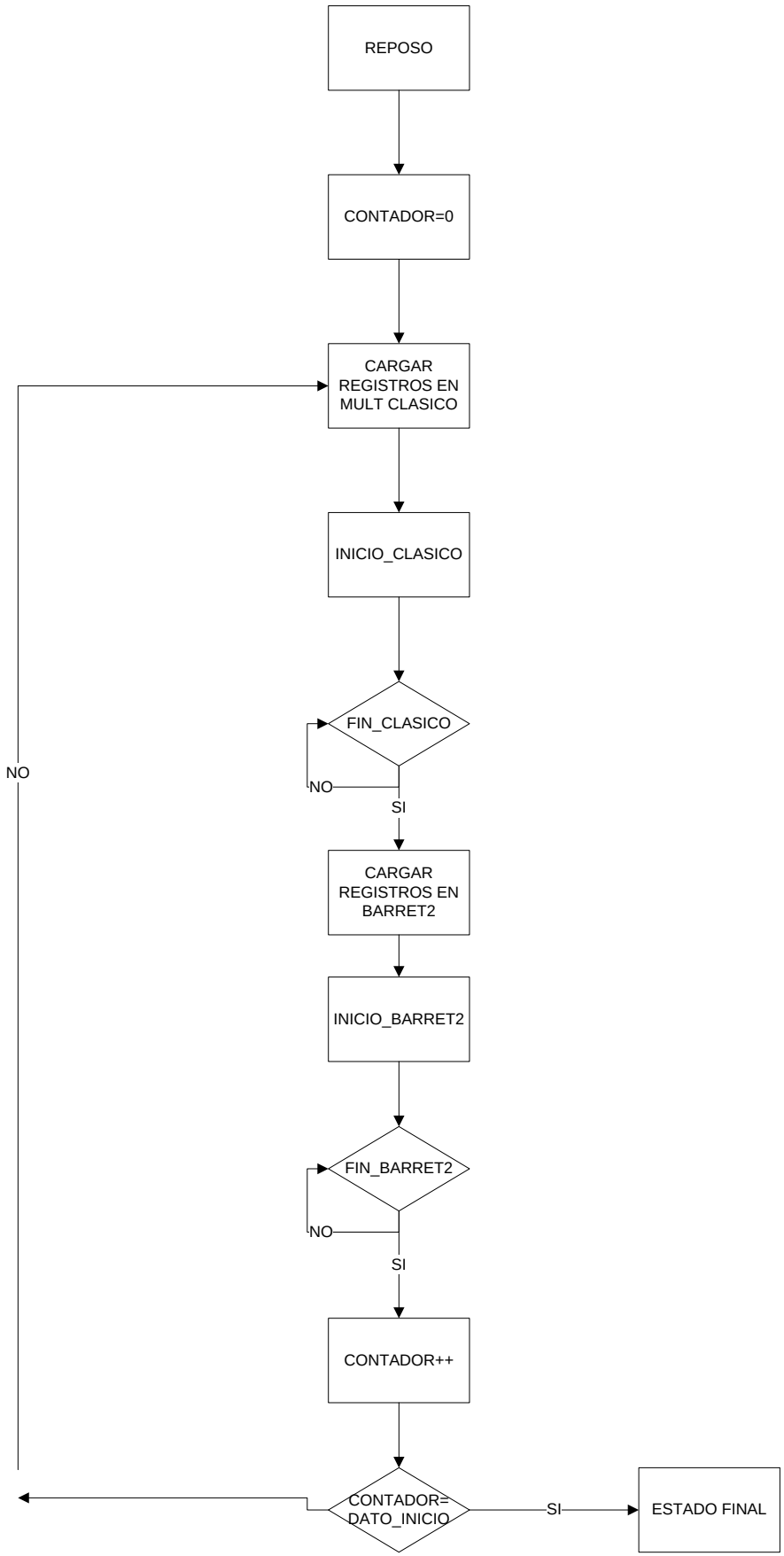


Figura 119 - D.ESTADOS BB&S CLASICO

4.4.3 SIMULACIÓN

A continuación se muestra una parte de la simulación total en la cual se muestra las señales de control entre el bloque de multiplicación y el bloque de obtener el modulo.

N = 331 P = 509 M = 168479 X_{-1} = 250

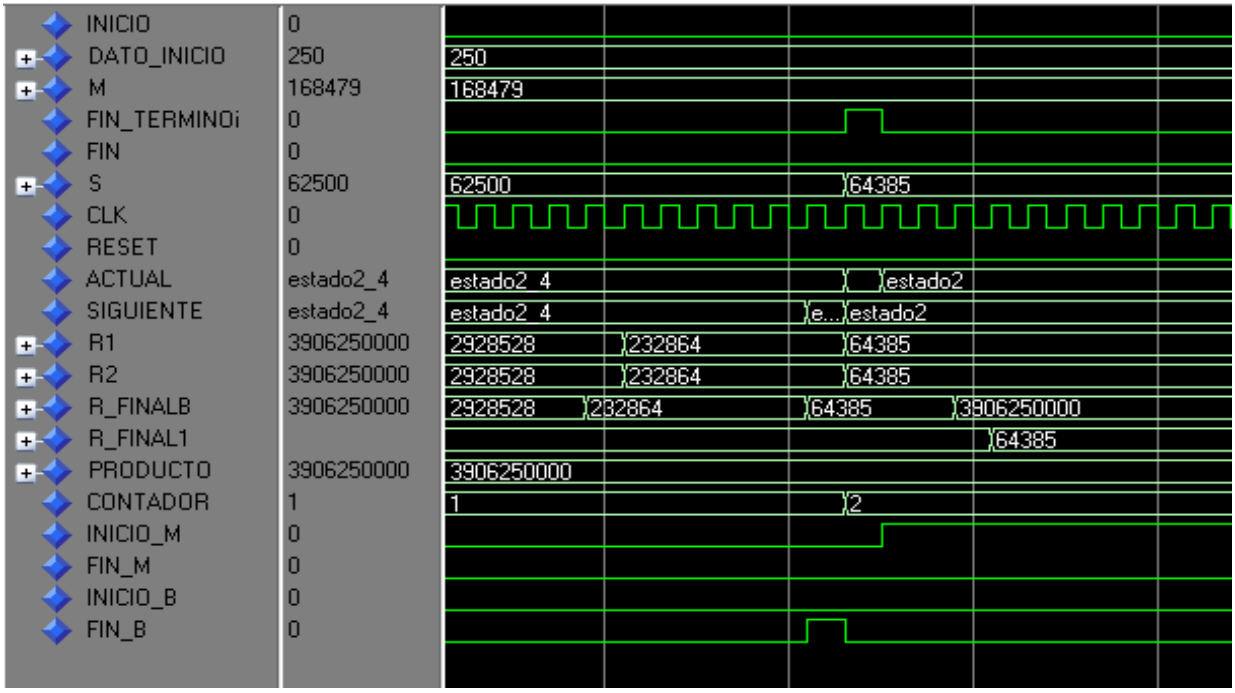


Figura 120 - SIMULACION BB&S CLASICO 1

Se activa el bit de Inicio_M para obtener primero la multiplicación de los valores de R1 y R2 que en este caso valen 64385.

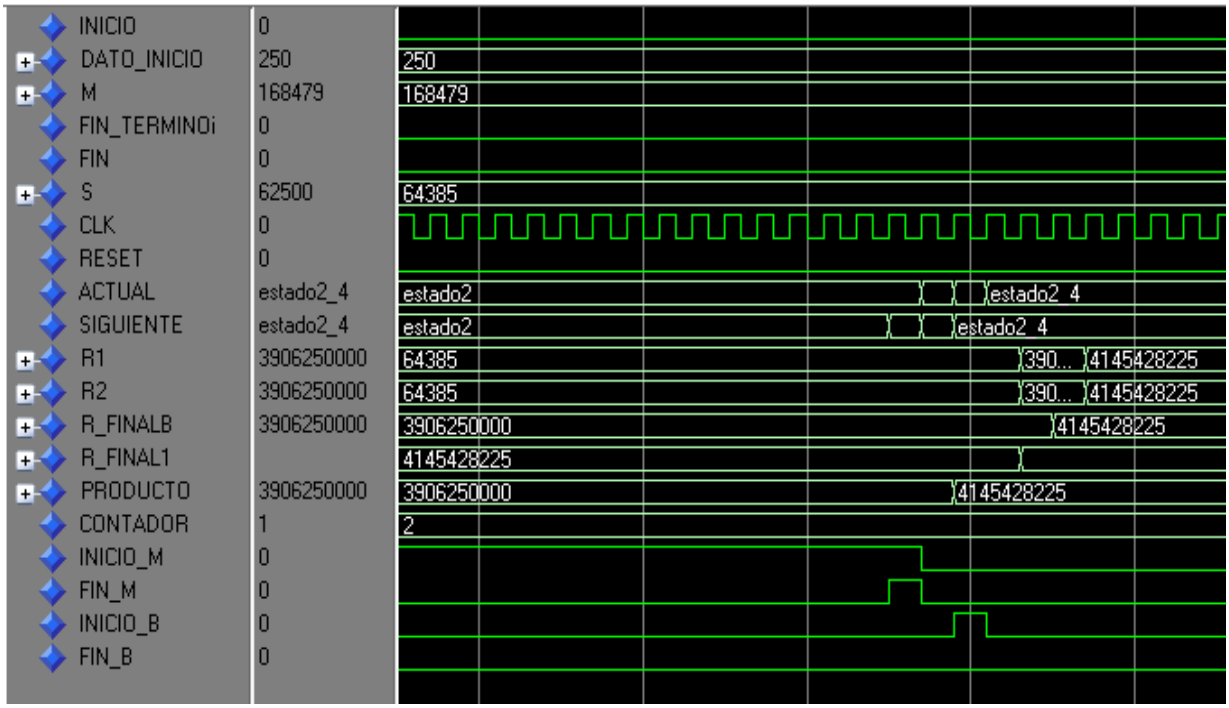


Figura 121 - SIMULACION BB&S CLASICO 2

Una vez terminada la multiplicación se obtiene la reducción aplicando barret 2, se activa el bit de Inicio_barret2 y se activa el bit de Fin_M. En el registro Porducto se almacena el resultado de la multiplicación de R1 *R2.

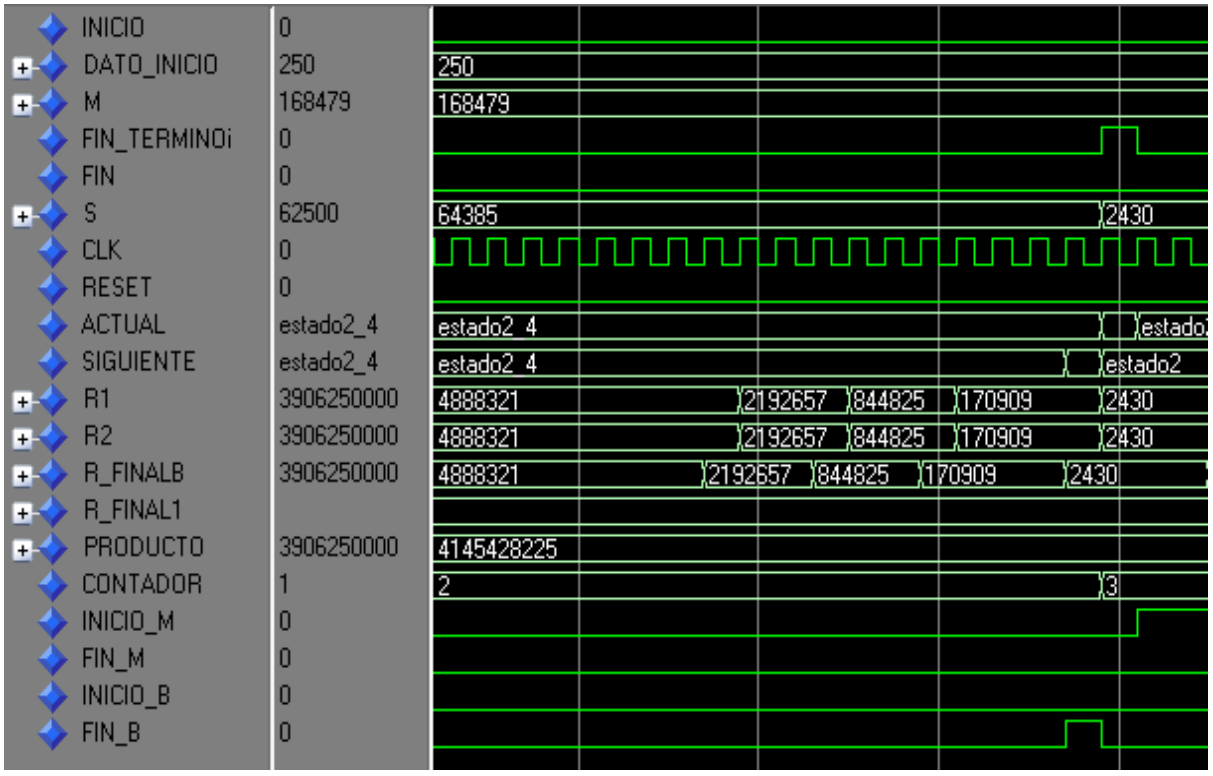


Figura 122 - SIMULACION BB&S CLASICO 3

Finalmente se obtiene el resultado del termino i que queramos calcular en este caso de la simulación el termino 2. Se activa el bit de Fin_terminoi.

A continuación se muestra la simulación completa con los valores anteriores.

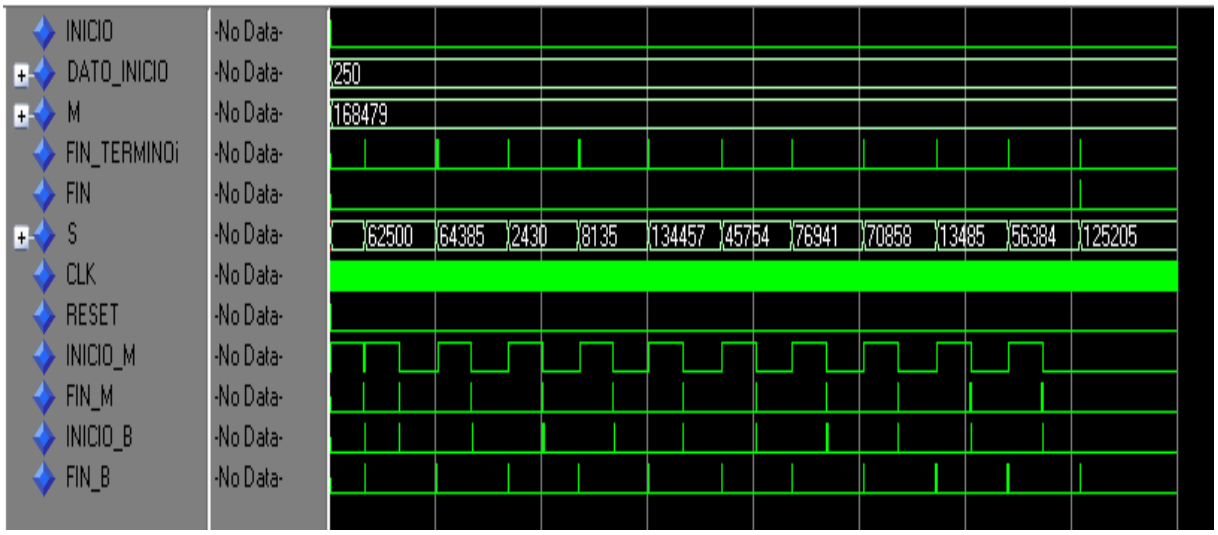


Figura 123 - SIMULACION BB&S CLASICO 3

Comparamos con la siguiente tabla si los datos son correctos.

X(i)	n	p	M=n*p	PRODUCTO	X_{-1}=250	PARIDAD	
0	331	509	168479	62500	62500	1	
1	331	509	168479	3906250000	64385	1	
2	331	509	168479	4145428225	2430	0	
3	331	509	168479	5904900	8135	1	
4	331	509	168479	66178225	134457	1	
5	331	509	168479	18078684849	45754	0	
6	331	509	168479	2093428516	76941	1	
7	331	509	168479	5919917481	70858	0	
8	331	509	168479	5020856164	13485	1	
9	331	509	168479	181845225	56384	0	
10	331	509	168479	3179155456	125205	1	
						10101011011	1371

Realizamos otra simulación con los mismo valores que en la segunda simulación de la implementación con Montgomery y karatsuba

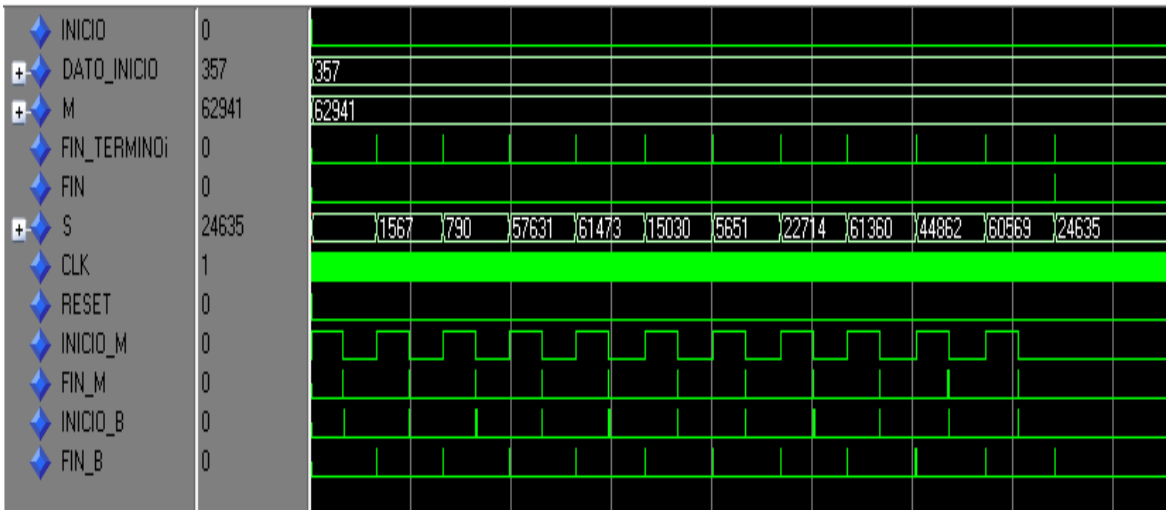


Figura 124 - SIMULACION BB&S CLASICO 4

.

Comparamos los resultados con la siguiente tabla.

X(i)	N	p	M=n*p	PRODUCTO	X_{-1}=357	PARIDAD	
0	557	113	62941	127449	1567	1	
1	557	113	62941	2455489	790	0	
2	557	113	62941	624100	57631	1	
3	557	113	62941	3321332161	61473	1	
4	557	113	62941	3778929729	15030	0	
5	557	113	62941	225900900	5651	1	
6	557	113	62941	31933801	22714	0	
7	557	113	62941	515925796	61360	0	
8	557	113	62941	3765049600	44862	0	
9	557	113	62941	2012599044	60569	1	
10	557	113	62941	3668603761	24635	1	
						11000011001	1561

4.4.4 SINTESIS

La sintesis de la implementación es la siguiente para un nivel de seguridad baja 160 bit y nivel de seguridad medio 512 bit.

N=160
Frecuencia Máxima = 35MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	3329	768	433%
Number of Slice Flip Flops	3067	1536	199%
Number of 4 input LUTs	6176	1536	402%
Number of bonded IOBs	484	124	390%
Number of GCLKs	1	8	12%

N=512
Frecuencia Máxima = 14Mhz

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	11816	768	1538%
Number of Slice Flip Flops	9783	1536	636%
Number of 4 input LUTs	21780	1536	1417%
Number of bonded IOBs	1540	124	1241%
Number of GCLKs	1	8	12%

4.4.5 RESULTADOS

En la siguiente grafica se muestra los resultados en cuanto a area ocupada y tiempo requerido a una frecuencia de 100Khz.

N º BITS	AREA (LUT)	TIEMPO (us)
160	6176	37000
512	21780	118400

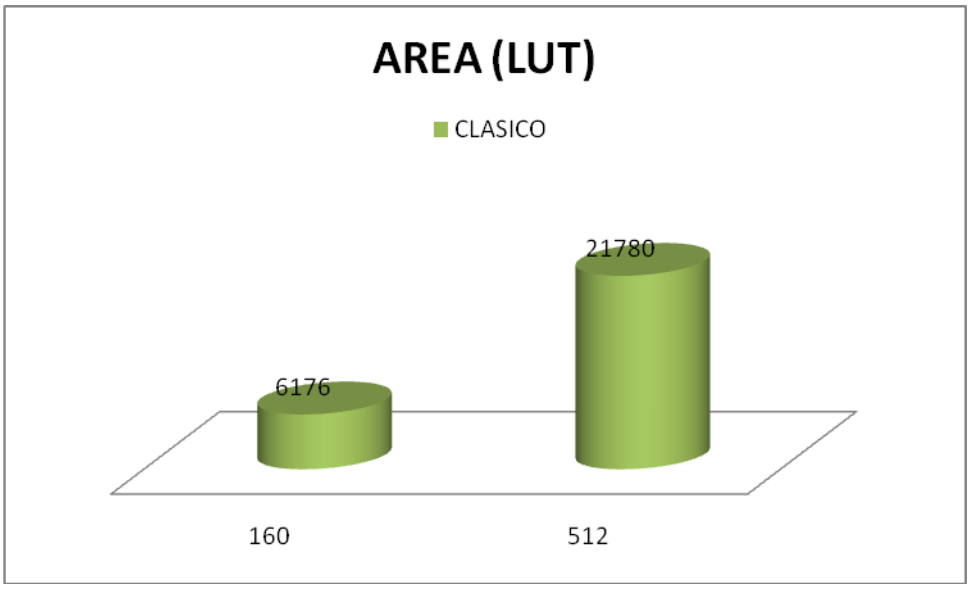


Figura 125 - AREA BB&S CLASICO

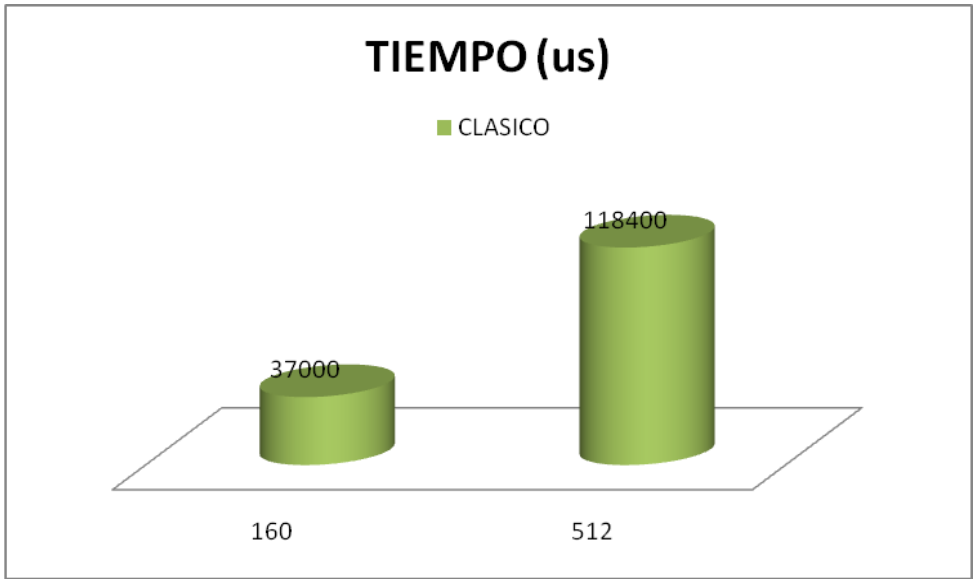


Figura 126 - TIEMPO BB&S CLASICO

GENERADOR DE NÚMEROS ALEATORIOS

[CONCLUSIÓN GENERADOR DE NÚMEROS ALEATORIOS]

ÁREA OCUPADA – TIEMPO – ÁREA X TIEMPO

4.5 CONCLUSIÓN BLUM BLUM & SHUB

A continuación se va a comparar lso tres metodos para este caso práctic en función del area ocupada , el tiempo que tarda y el factor Área x Tiempo.

Comparación area: (LUT)

N º BITS	CLASICO	KARATSUBA	MONTGOMERY
160	6176	10433	3052
512	21780	40052	9953

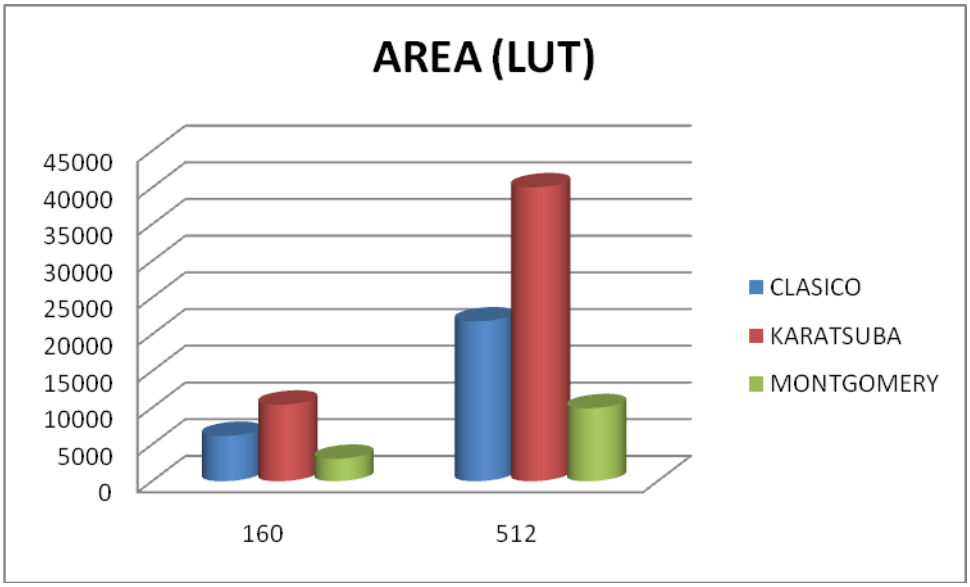


Figura 127 - AREA BB&S CLASICO_KARAT_MONTG

Basandonos en los resultados en caso de necesitar una implemetación en la cual nos ocupe poca area se elegiría la implementación de Montgomery.

Comparación de tiempo (us)

N º BITS	CLASICO	KARATSUBA	MONTGOMERY
160	37000	29000	35630
512	118400	92800	112000

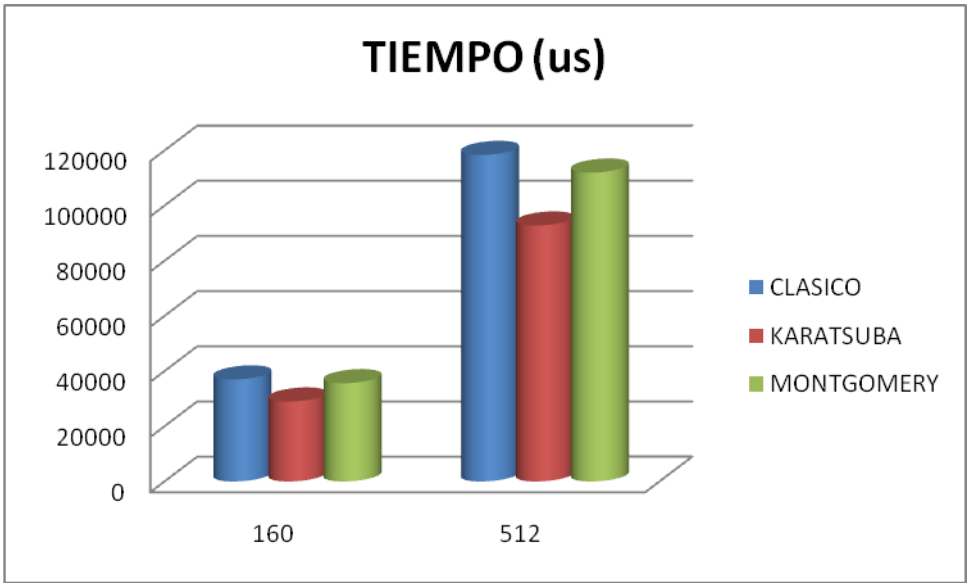


Figura 128 - TIEMPO CLASICO_KARAT_MONTG

Para una implementación en la que se busque rapidez se elegiría la implementación de Karatsuba.

Para una comparación global compararemos el factor tiempo x area:

N º BITS	CLASICO	KARATSUBA	MONTGOMERY
160	228512000	302557000	108742760
512	2578752000	3716825600	1114736000

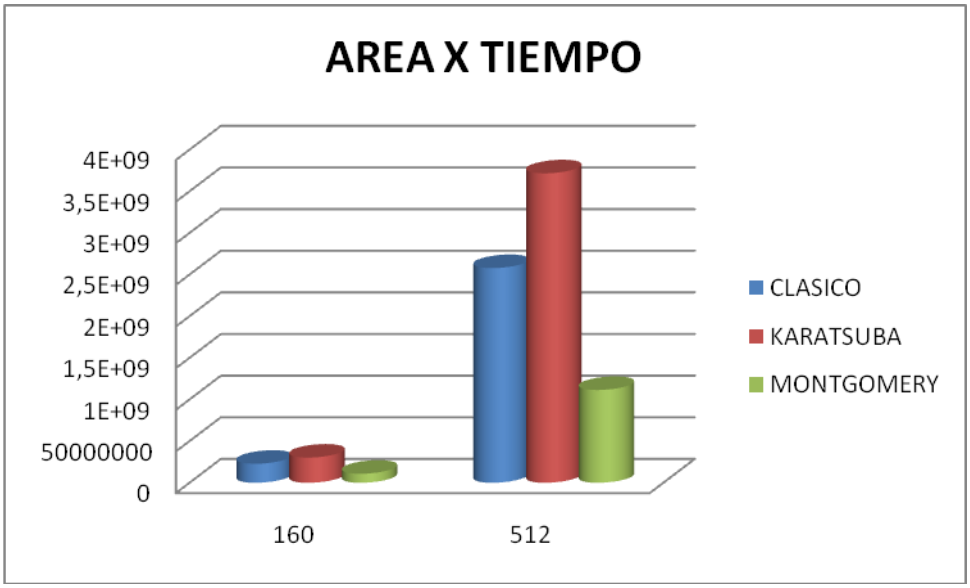


Figura 129 - AREA X TIEMPO CLASICO_KARAT_MONTG

En la grafica se muestra como la implementación de Montgomery para estas circunstancias de 100 Khz y 160 o 512 bits es la mas favorable, por tanto es la que se elegiría en caso de necesitar un circuito en compensación de Área con el Tiempo.

IMPLEMENTACIÓN ALGORITMO

[EXPONENCIACIÓN MODULAR]

IMPLEMENTACIÓN – SIMULACIÓN – SÍNTESIS – RESULADOS

5. IMPLEMENTACIÓN EXPONENCIACIÓN MODULAR

Se ha realizado una implementación para realizar la exponenciación modular [8] el cual se puede usar para calcular uno de los términos del generador de números aleatorios en el algoritmo Blum Blum and Shub. A través de la siguiente fórmula se puede hallar cualquier término de la sucesión sin necesidad de realizar el cálculo de todos los término anteriores.

$$x_i = \left(x_0^{2^i \bmod (p-1)(q-1)}\right) \bmod M.$$

Siendo p y q los números primos elegidos. Como se oberva se necesitan dos multiplicaciones exponenciación modular de ahí que se necesite implementar un algoritmo para poder calcular el dato.

El algoritmo el cual se implementará será la exponenciación modular serie el cual se describe a continuación:

Algoritmo. SequentialBinary (t,E,M)
Int R:=1
If e_{m-1}=1 then R:=T;
For i:=m-1 downto 0 do
R:=R x R mod M
If e_i = 1 then R:= R x T mod M ;
Return R;
End

Para realizar la parte de la multiplicación modular utilizaremos el algoritmo de Montgomery debido a lo expuesto en el apartado X, como el algoritmo se realiza 2 veces en total se realizará 4 multiplicaciones de montgomery, por tanto el número de ciclos de reloj que se tarda en realizar todo el algoritmo es 4*Nciclos de Montgomery

[8] [7] Bibliografía

5.2 IMPLEMENTACIÓN EN VHDL

En la siguiente parte del codigo se muestra la parte del algoritmo:

Codigo=>

```
--Porceso secuencial
ELSIF ACTUAL=ESTADO1 THEN
                    CONTADOR<=CONTADOR;
                    IF E(N-1) = '1' THEN R<=T;
                   END IF;
                      R1<=R;
                      R2<=R;

                    ELSIF ACTUAL=ESTADO1_1 THEN
                     R1<=R;
                     R2<=R;

                    ELSIF ACTUAL=ESTADO2 THEN
                      -- SE REALIZA MONTGOMERY

                      CONTADOR<=CONTADOR;
                      R<=FF;


                   ELSIF ACTUAL=ESTADO3 THEN

                      CONTADOR<=CONTADOR;
                      IF E(CONTADOR)='1' THEN
                        R1<=R;
                        R2<=T;

                     END IF;


                    ELSIF ACTUAL=ESTADO4 THEN
                       --SE REALIZA MONT => RXT MOD M
                        CONTADOR<=CONTADOR-1;
                      --  R1<=R;
                      --  R2<=R;


                     ELSIF ACTUAL=ESTADO4_1 THEN
                       R<=FF;

    --Parte combinacional

   WHEN ESTADO1=>
                INICIO_M<='0';
                FIN_E<='0';
                SIGUIENTE<=ESTADO1_1;

              WHEN ESTADO1_1=>
                 INICIO_M<='1';
                 FIN_E<='0';

                IF CONTADOR=-1 THEN
                  SIGUIENTE<=ESTADO5;
                  ELSE

                  SIGUIENTE<=ESTADO2;
                  END IF;


             WHEN ESTADO2=> --SE HACE OTRA VEZ EL ALGORITMO CON (R,E,M)
                INICIO_M<='0';
                FIN_E<='0';
```

```
IF FIN_M='1' THEN
                SIGUIENTE<=ESTADO3;

              ELSE SIGUIENTE<=ACTUAL;
            end if;

            WHEN ESTADO3=>
               FIN_E<='0';
               INICIO_M<='0';
               SIGUIENTE<=ESTADO4;

            WHEN ESTADO4=>
               FIN_E<='0' ;
               INICIO_M<='1';

               IF E(CONTADOR)='1' THEN
               SIGUIENTE<=ESTADO4_1;
               ELSE SIGUIENTE<=ESTADO1_1;
             END IF;

            WHEN ESTADO4_1=>
               FIN_E<='0';
               INICIO_M<='0';

               if FIN_M='1' THEN
               SIGUIENTE<=ESTADO1_1;
               ELSE SIGUIENTE<=ACTUAL;
             END IF;

            WHEN ESTADO5=>

               INICIO_M<='0';

               IF FIN_M<='1' THEN
                  SIGUIENTE<=FINAL;
                  FIN_E<='1' ;
               ELSE
                 SIGUIENTE<=ESTADO5;
                 FIN_E<='0';
              END IF;
```

En la parte combinacional se puede observar como se va activando el bit de Inicio_Montgomery para que realice el cálculo, una vez que este termina activa el bit de Fin_Montgomery para avisar al modulo de la exponenciación que ya puede cambiar de estado. También se observa en la parte secuencia como se cargan los registro de R1 y R2 para usarlos en la implementación de Montgomery de la siguiente manera (R1,R2,M).

El diagrama de flujo es el siguiente:

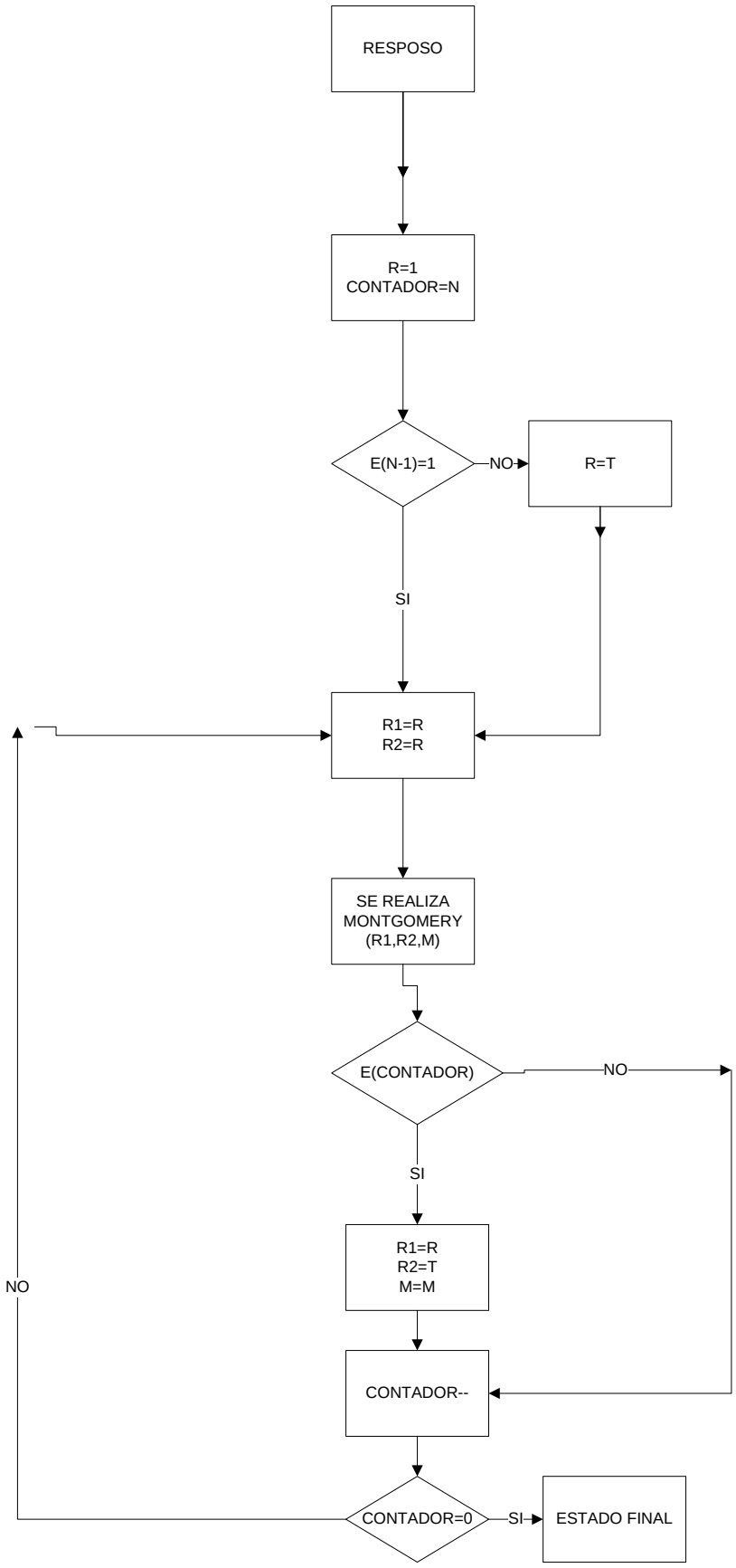


Figura 130 - D.ESTADOS EXPONENTIATION

5.3 SIMULACIÓN

A continuación se realiza la simulación para los valores (T,E,M)= (5,3,7), el resultado debe ser 6.

Se observa como este algoritmo necesita realizar 4 veces un algoritmo de multiplicación-modulación como es el caso de Montgomery o Buckley, en esta simulación se ha simulado usando Montgomery.

Cada vez que se necesita realizar la multiplicación moduar se activa el bit de Inicio_M, y cada vez que este algoritmo termina se activa el bit de Fin_M.

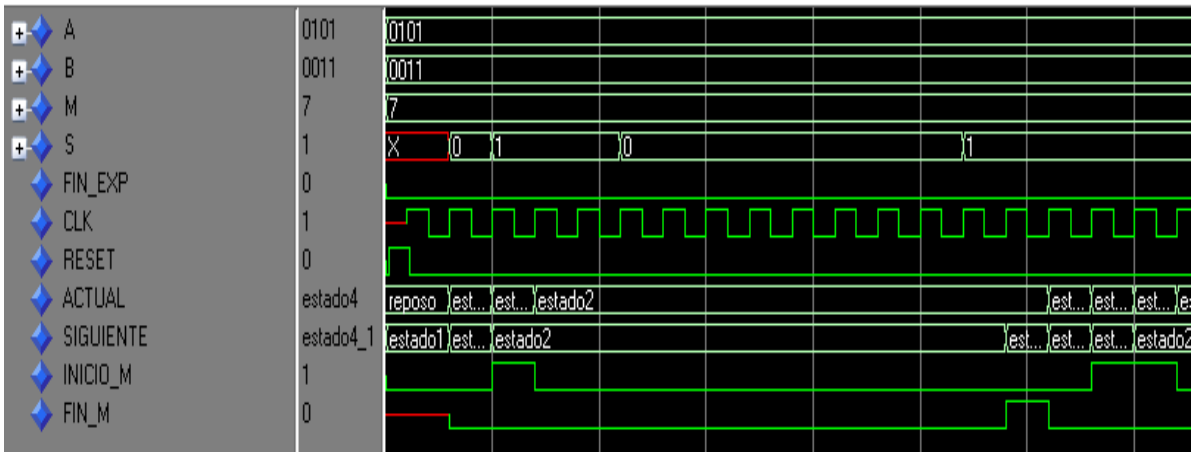


Figura 131 - SIMULACION EXP 1

En el estado 4_1 se realiza de nuevo la multiplicación modular con los nuevos valores.

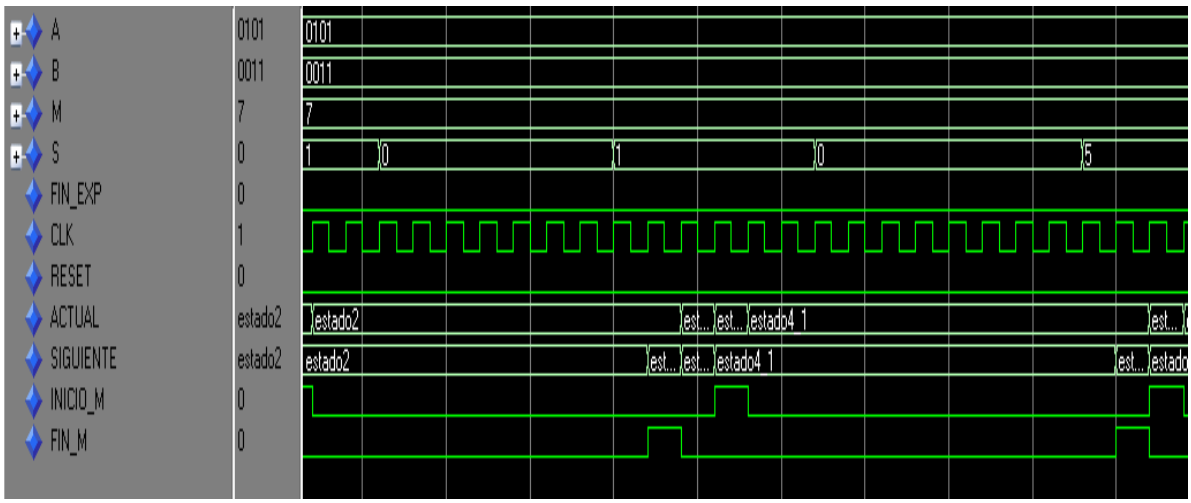


Figura 132 - SIMULACION EXP 2

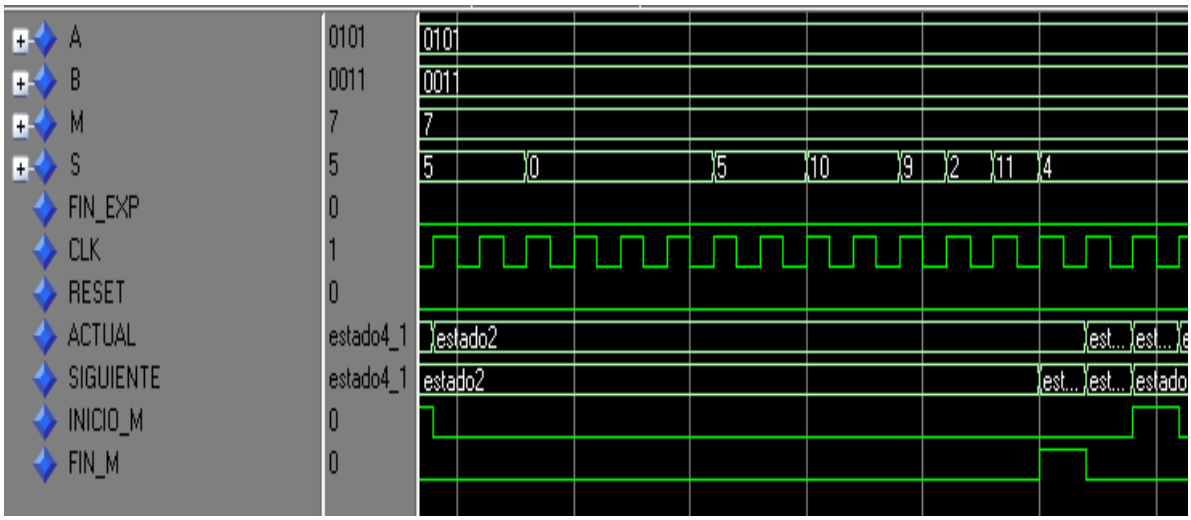


Figura 133 - SIMULACION EXP 3

Una vez termina de realizar el algoritmo se activa el bit de Fin_Exp, el resultado es 6 que es el resultado de realizar la multiplicación exponencial modular de 5^3 mod 7.

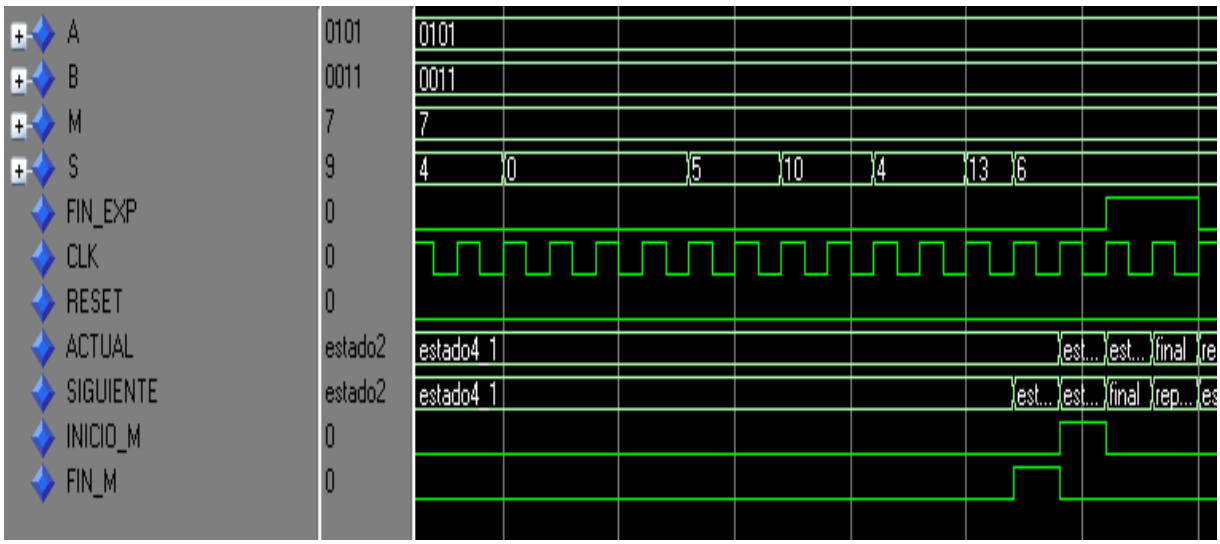


Figura 134 - SIMULACION EXP 4

A continuación se muestra una simulación de 16 bit para los valores A = 3, B = 8 y M = 323 cuyo resultado debe ser de 101. Solo se muestra el inicio de la simulación y el final.

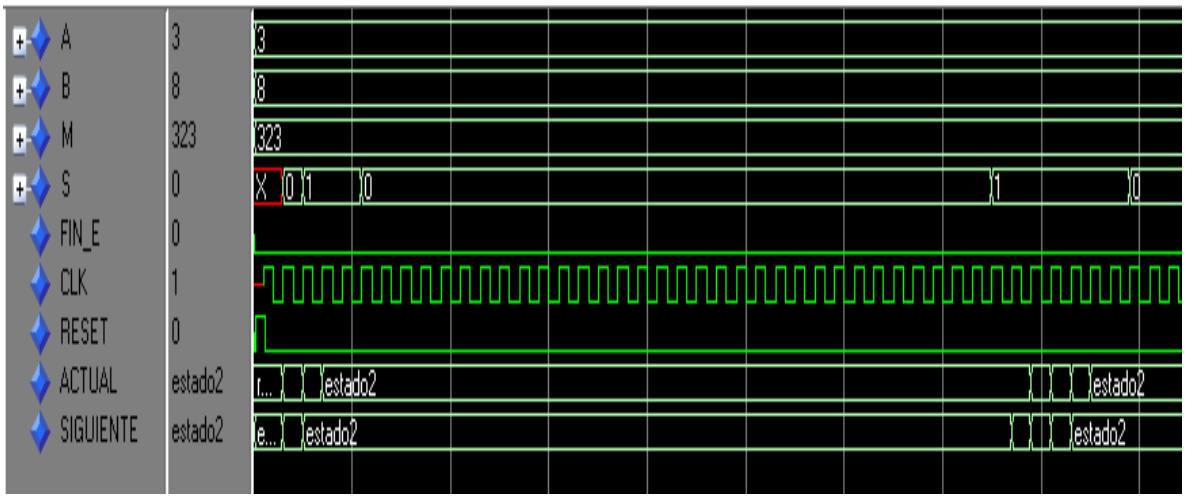


Figura 135 - SIMULACION EXP 5

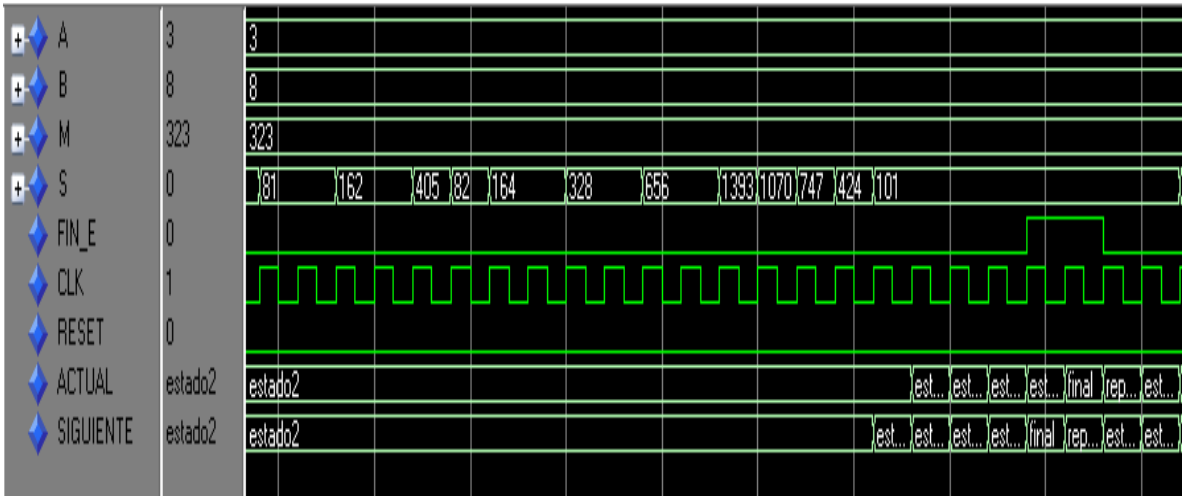


Figura 136 - SIMULACION EXP 6

5.4 SINTESIS

A continuación se muestra los resultados de la síntesis:

N=4
Frecuencia Máxima = 150MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	63	768	8%
Number of Slice Flip Flops	47	1536	3%
Number of 4 input LUTs	121	1536	7%
Number of bonded IOBs	20	124	16%
Number of GCLKs	1	8	12%

N=8
Frecuencia Máxima = 126MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	112	768	14%
Number of Slice Flip Flops	75	1536	4%
Number of 4 input LUTs	216	1536	14%
Number of bonded IOBs	36	124	29%
Number of GCLKs	1	8	12%

N=16
Frecuencia Máxima = 124MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	162	768	21%
Number of Slice Flip Flops	133	1536	8%
Number of 4 input LUTs	312	1536	20%
Number of bonded IOBs	68	124	54%
Number of GCLKs	1	8	12%

N=32
Frecuencia Máxima =103MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	302	768	39%
Number of Slice Flip Flops	251	1536	16%
Number of 4 input LUTs	579	1536	37%
Number of bonded IOBs	132	124	106%
Number of GCLKs	1	8	12%

N=64
Frecuencia Máxima = 75MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	582	768	75%
Number of Slice Flip Flops	477	1536	31%
Number of 4 input LUTs	1104	1536	71%
Number of bonded IOBs	260	124	209%
Number of GCLKs	1	8	12%

N=128
Frecuencia Máxima = 48MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1138	768	148%
Number of Slice Flip Flops	929	1536	60%
Number of 4 input LUTs	2142	1536	139%
Number of bonded IOBs	516	124	416%
Number of GCLKs	1	8	12%

N=256
Frecuencia Máxima =28MHZ

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	2257	768	293%
Number of Slice Flip Flops	1834	1536	119%
Number of 4 input LUTs	4236	1536	275%
Number of bonded IOBs	1028	124	829%
Number of GCLKs	1	8	12%

5.5 RESULADOS

Nº BITS	ÁREA (LUT)	FMAX (MHZ)	CICLOS DE RELOJ
4	121	150	36
8	216	126	68
16	312	124	132
32	579	103	260
64	1104	75	516
128	2142	48	1028
256	4236	28	2052

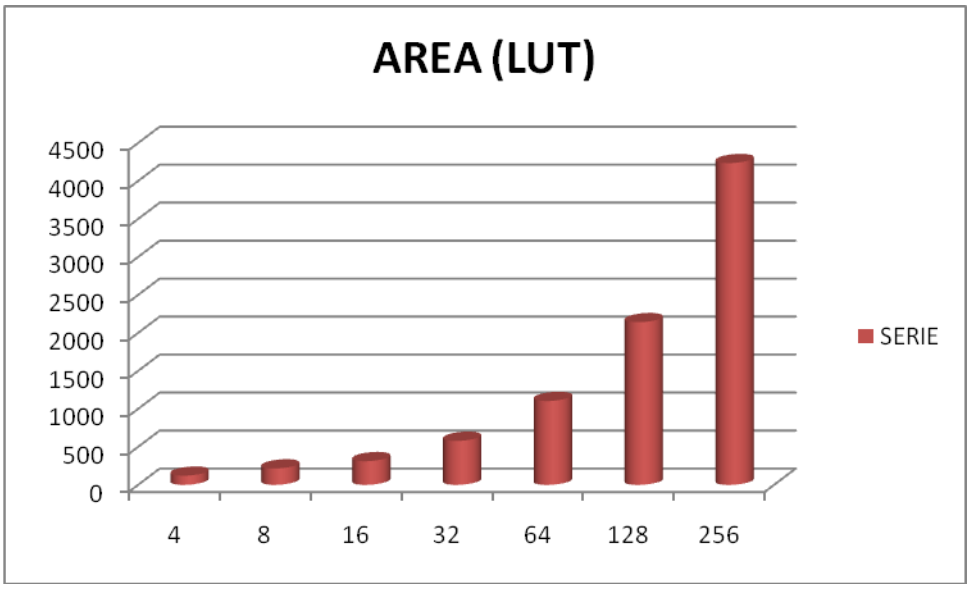


Figura 137 - AREA EXPONENTIATION

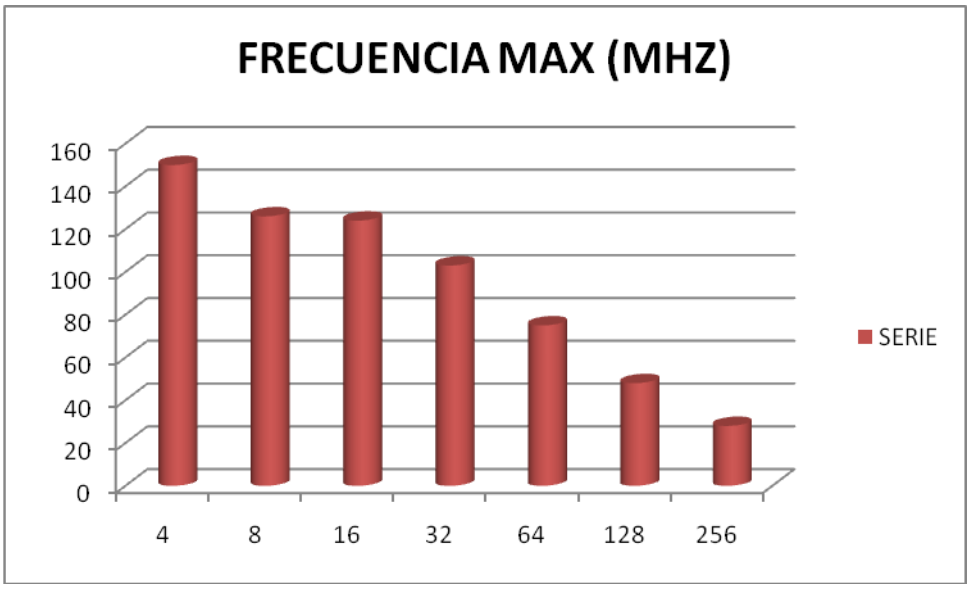


Figura 138 - FRECUENCIA EXP

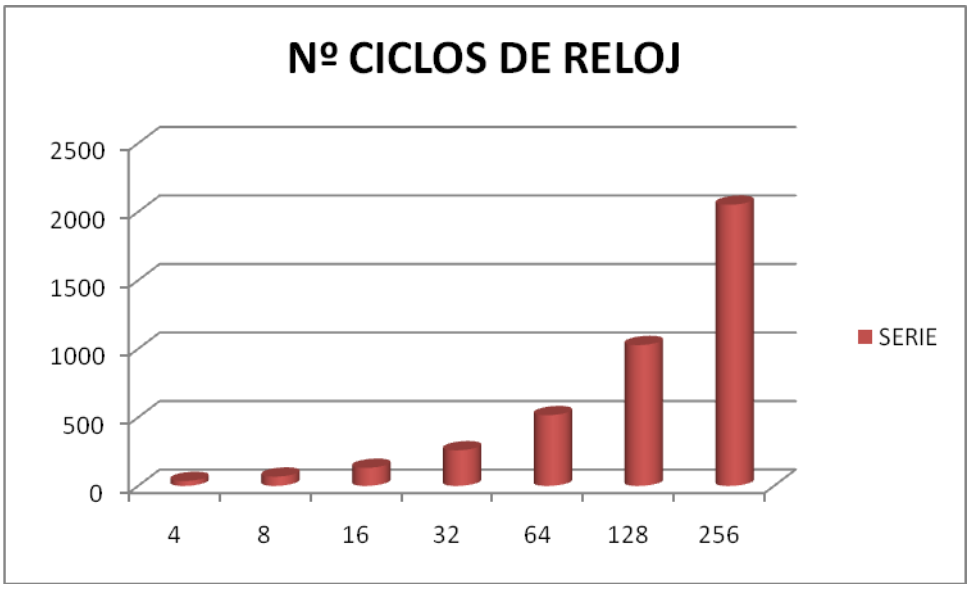


Figura 139 - TIEMPO EXP

Existen otras implementaciones para realizar la exponenciación modular las cuales no se han realizado en este proyecto, entre las cuales se podrían realizar para poder comparar están la exponenciación modular paralelo y la exponenciación modular sistolic.

6. CONCLUSIÓN DEL PROYECTO

En la primera parte del proyecto hemos descrito aspectos básicos del sistema RFID. A la vez se ha mostrado un ejemplo de protocolo de encriptación de datos (Protocolo SLAP). Seguidamente se ha realizado una introducción al lenguaje VHDL haciendo un repaso por la evolución que ha tenido este lenguaje en la historia.

En la segunda parte del proyecto nos hemos centrado en la implementación de los algoritmos de multiplicación modular. En un principio se han realizado las implementaciones de multiplicación: clásico, combinacional y karatsuba-ofman. De estos tres multiplicadores se ha descartado la implementación combinacional debido a la excesiva área que llega a ocupar para número de bits elevados.

Las implementaciones clásico y karatsuba necesitaban de algoritmos de reducción, los cuales se han implementado dos algoritmos de reducción de barret denominados barret1 y barret2, de estos, se ha descartado el de barret1 debido al elevado tiempo que puede llegar a tardar, por tanto se ha decidido que a las implementaciones de multiplicación clásico y karatsuba se le añadirá la implementación de barret2 para obtener la multiplicación modular.

De los algoritmos de multiplicación modular directa se ha realizado las implementaciones de Montgomery y Buckley. Una vez comparado los resultados de estos dos algoritmos se ha decidido que la implementación basada en el algoritmo de Montgomery era más eficaz debido al factor Área x Tiempo.

Posteriormente se realizado la implementación de un generador de números aleatorios, Blum Blum & Shub, en este generador de números aleatorios hemos simulado el funcionamiento de las implementaciones de multiplicación modular previamente elegidas: clásico+barret2, karatsuba+barret2 y Montgomery. Una vez visto el comportamiento de estos circuitos en el generador de números aleatorios hemos descartado en un principio la implementación clásico debido a que no era la que menos tiempo tardaba ni tampoco la que menos área ocupaba. De las otras dos implementaciones restantes, la de karatsuba era la que menos tiempo tardaba mientras que la implementación de Montgomery ocupaba menos, por tanto se ha comparado estas dos en un factor Área x Tiempo y se ha podido comprobar que este factor era mejor para la implementación de Montgomery que para la basada en el algoritmo de

Karatsuba-ofman. Por tanto podemos concluir que la mejor implementación para realizar el algoritmo de Multiplicación Modular para el caso de nuestra aplicación (tags para la tecnología RFID) es la implementación basada en el algoritmo de Montgomery.

7. PROPUESTAS FUTURAS

En los trabajos fututos lo que se considera que se puede realizar son las siguientes partes:

a) La implementación de un protocolo de encriptación completo como por ejemplo el protocolo de encriptación SLAP
b) Búsqueda de más generadores de números aleatorios, los cuales se adapten al sistema que queremos utilizar, realizando comparaciones y estudios detallados, similares a los que se han realizado en este proyecto con las diferentes implementaciones de multiplicadores, sobre cuál es mejor para el caso de la tecnología RFID.
c) Otra propuesta futura que consideramos que puede resultar interesante es la de realizar otras implementaciones de exponenciación modular, en este proyecto se ha realizado una implementación de exponenciación modular exponencial en serie, por tanto consideramos que se podría realizar la implementación exponencial modular sistolic y paralelo y compararla con la realizada en este proyecto para obtener una implementación óptima de este algoritmo.

8. BIBLIOGRAFÍA

[1] RFID : applications, security, and privacy Garfinkel, Simson, Adison-Wesley [2006]

[2] RFID : la tecnología de identificación por radiofrecuencia Muñoz Frías, José Daniel

[3] RFID security **,** Thornton, Frank. Syngress [2006]

[4] Nedjah, N. and Mourelle, L.M., Simulation model for hardware implementation of modular multiplication, **In:** Mathematics nad Simulation with Biological, Economical and Musicoacoustical
Applications, C.E. D'Attellis, V.V. Kluev, N.E. Mastorakis Eds. WSEAS Press, 2001, pp. 113-118.

[5] Nedjah, N. and Mourelle, L.M., Reduced hardware architecture for the Montgomery modular multiplication, WSEAS Transactions on Systems, **1**(1):63-67.

[6] Nedjah, N. and Mourelle, L.M., Two Hardware implementations for the Montgomery modular multiplication: sequential versus parallel, Proceedings of the 15th. Symposium Integrated Circuits and Systems Design, Porto Alegre, Brazil, IEEE Computer Society Press, pp. 3-8, 2002

[7] Nedjah, N. and Mourelle, L.M., Reconfigurable hardware implementation of Montgomery modular multiplication and parallel binary exponentiation, Proceedings of the EuroMicro Symposium on Digital System Design − Architectures, Methods and Tools, Dortmund, Germany, IEEE Computer Society Press, pp. 226-235, 2002

[8] Nedjah, N. and Mourelle, L.M., Efficient hardware implementation of modular multiplication and exponentiation for public-key cryptography, Proceedings of the 5th. International Conference on High Performance Computing for Computational Science, Porto, Portugal, Lecture Notes in Computer Science, **2565**:451-463, Springer-Verlag, 2002

[9] Nedjah, N. and Mourelle, L.M., Hardware simulation model suitable for recursive computations: Karatsuba-Ofman's multiplication algorithm, Proceedings of ACS/IEEE

International Conference on Computer Systems and Applications, Tunis, Tunisia, July 2003.

[10] Nedjah, N. and Mourelle, L.M., A Reconfigurable recursive and efficient hardware for Karatsuba-Ofman's multiplication algorithm, Proceedings of IEEE International Conference on Control and Applications, Istambul, Turkey, June 2003, IEEE System Control Society Press Rivest, R., Shamir, A. and Adleman, L., A method for obtaining digital signature and public-key cryptosystems, Communications of the ACM, **21**:120-126, 1978.

[11] Walter, C.D., Systolic modular multiplication, IEEE Transactions on Computers, **42**(3):376-378, 1993.

[12] Schneier, Bruce. Applied Cryptography Second Edition: Protocols, algorithms, and source code in C. 1996, John Wiley & Sons, Inc.

9. ANEXO

En el anexo se han incluido los códigos de los programas que se han implementado en vhdl. Estos códigos se encuentran en el cd adjunto a este proyecto.