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: CARLOS ROSELL GARCÍA

TUTOR: ENRIQUE SAN MILLÁN HEREDIA

JULIO 2009

INDICE

Universidad Carlos III de Madrid

Índice Ilustraciones

CAPÍTULO I - INTRODUCCIÓN Y OBJETIVOS

1. ¿Qué es RFID?

Durante muchos años el código de barras ha sido el sistema de identificación de objetos por excelencia, pero todo apunta a que esta tecnología, en un futuro no muy lejano, tiene los días contados.

La tecnología de auto-identificación por radiofrecuencia (RFID) está empujando con fuerza y esta empezado ya a desplazar al código de barras en algunas aplicaciones y sectores concretos.

El sistema emergente permite identificar objetos a distancia mediante etiquetas electrónicas. Ofrece múltiples ventajas y supone una solución a muchos problemas hasta ahora no resueltos.

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).

Una etiqueta RFID es un dispositivo pequeño, similar a una pegatina, que puede ser adherida o incorporada a un producto, animal o persona. Contienen antenas para permitirles recibir y responder a peticiones por radiofrecuencia desde un emisor-receptor RFID.

Las etiquetas RFID están llamadas a convertirse en el sustituto de los códigos de barras. De hecho, éstas permiten almacenar un código de identificación formado por 64 ó 96 bits, comúnmente llamado Código Electrónico de Producto (EPC). Las aplicaciones de esta tecnología en el sector de la distribución y la logística son innumerables, pues permite la localización, la identificación, el estado o cualquier otro tipo de información de los productos, incluso en movimiento, y sin necesidad de intervención humana.

Esto agiliza los procesos de inventario o gestión de stocks y permite realizar la contabilidad en tiempo real. Otras aplicaciones podrían ser el control de accesos y los inmovilizadores de vehículos.

1.1. Actuales problemas de la tecnología.

Aparte del problema moral o ético que supone la tecnología RFID, es decir, la posibilidad de tener un articulo, persona, animal o en general cualquier cosa con una etiqueta RFID, a la cual puede ser detectada por distintos receptores en cualquier parte del mundo, con la consecuencia de poder estar vigilado o localizable en cualquier parte del mundo. La mayoría de las preocupaciones giran alrededor del hecho de que las etiquetas RFID puestas en los productos siguen siendo funcionales incluso después de que se hayan comprado los productos y se hayan llevado a casa, y esto puede utilizarse para vigilancia.

Por otro lado está el problema de las comunicaciones entre etiquetas y receptores, es necesario que estas comunicaciones se hagan de forma segura, para ello se utilizan protocolos de comunicación en el cual los datos que se transmiten entre las distintas etiquetas y receptores se encriptan mediante distintos métodos.

Uno de los métodos más utilizados en criptografía son los protocolos que utilizan la algoritmos matemáticos, como algoritmos de multiplicación, multiplicación modular o las técnicas más sofisticadas utilizan exponenciación modular con protocolos de encriptación tales como el SLAP o el RSA, estos protocolos se basan en la utilización productos de números primos grandes se (mayores que 10^{100}) elegidos al azar para conformar la clave de cifrado y descifrado, emplean expresiones exponenciales en aritmética modular .

La seguridad de estos algoritmos radica en que no hay maneras rápidas conocidas de factorizar un número grande en sus factores primos utilizando computadoras tradicionales, es decir mediante software.

Debido a que este tipo de algoritmos se realizan mediante software, el principal problema que se da, es que el sistema solo puede funcionar con ancho de palabra para los datos entorno los 96 bits, con lo que en la actualidad limitan el tamaño de la información que son capaces de manejar. Con implementación de estos algoritmos en hardware pasamos a anchos de palabra mayores, de los tradicionales 96 bits a anchos de palabra que rondan los 512 bits.

Los problemas principales que tienen estos algoritmos de encriptación con mayores anchos de palabra son dos, en primer lugar problemas de espacio, ya que una etiqueta RFID dependiendo del uso el cual se le asigne tiene que ser necesariamente pequeña. Con lo que se limita el área donde colocar el chip RFID.

Por otro lado la velocidad de funcionamiento de esta, ya que en ciertas aplicaciones no tiene sentido que la etiqueta RFID tarde un largo tiempo en dar su información, estos dos aspectos son aspectos clave.

Por último el aspecto energético, ya que es interesante que estos protocolos necesiten la mínima energía para funcionar de forma correcta.

2. Objetivos del proyecto

Como hemos hablado en el anterior punto uno de los grandes problemas que se encuentran actualmente en el uso de la tecnología RFID es la seguridad. Como hemos comentado los algoritmos de seguridad, necesitan manejar grandes cadenas de bits y dentro de la propia etiqueta procesarlos y realizar las operaciones pertinentes para poder dar una respuesta al sistema RFID, siempre esta respuesta debe transmitirse de forma rápida y segura.

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.

El principal inconveniente de estos dispositivos, es el espacio, hemos visto como las diferentes etiquetas y fabricantes intentan desarrollar etiquetas cada vez más pequeñas con lo que realmente en algunos casos delimitan el hardware que se pueden añadir a ellas.

Con lo cual en algunos casos no podemos introducir todo el hardware necesario para que se cumplan los protocolos de seguridad estándares que se intentan asociar a la tecnología, sin esta seguridad esta tecnología no se puede ir imponiendo a los actuales códigos de barras.

Este tipo de algoritmos criptográficos se basan en distintas operaciones matemáticas entre las que cabe destacar la exponenciación modular, la multiplicación modular, y la multiplicación simple.

Con lo que el primer objetivo de nuestro proyecto es realizar una selección de distintos algoritmos matemáticos utilizados en los protocolos de seguridad que utiliza la tecnología RFID.

En nuestro caso estudiaremos las distintas maneras de realizar estas operaciones mediante distintos algoritmos, con el objeto de obtener una estimación en cuanto al espacio que podría ocupar el algoritmo y cuál sería su tiempo de ejecución, y poder realizar comparaciones para poder obtener las mejores propiedades de los algoritmos seleccionados.

Para implementar nuestros algoritmos, utilizaremos el lenguaje de programación de Hardware VHDL. Con lo que el segundo objetivo de nuestro proyecto, es implementar en el lenguaje de programación de Hardware, VHDL, los distintos algoritmos seleccionados para realizar las diferentes operaciones matemáticas.

Una vez implementado los algoritmos en el lenguaje de programación VHDL, debemos realizar las diferentes pruebas para poder ver las diferentes características que da cada algoritmo. Para ello hemos decidido trabajar sobre hardware reprogramable, específicamente sobre la tecnología FPGA.

Hemos decidido realizar el proyecto de esta forma y no sobre los propios simuladores y sintetizadores de estas etiquetas RFID y sus circuitos integrados. Esto se debe a que el objeto de nuestro proyecto es realizar una estimación cualitativa, con lo que un sistema reprogramable en hardware del tipo FPGA nos daría los suficientes datos para en un futuro poder realizar una implementación real en sus propios circuitos integrados específicos para las etiquetas RFID, sabiendo de antemano que algoritmos han de ser utilizados según se busquen las distintas especificaciones necesarias , ya sean requerimientos de tiempo de ejecución, o requerimientos de espacio.

De esta manera de trabajar sacamos otro dos objetivos de nuestro proyecto, en primer lugar, ver que es posible crear estos algoritmos sobre un sistema de FPGA, es decir, que realmente son sintetizables. Como cuarto objetivo comparar las distintas características que arrojan cada uno de estos algoritmos.

Una vez analizado todos los datos pasamos al último objetivo de nuestro proyecto, ver cuál de los algoritmos seleccionados es más adecuado para poder realizar en el futuro un sistema RFID real, comparando su área y su tiempo de ejecución.

Resumiendo, estos son los objetivos principales que se van a intentar desarrollar a lo largo de este proyecto:

1. Selección de distintos algoritmos matemáticos.

2. Implementación de algoritmos seleccionados.

3. Sintetización de los algoritmos matemáticos.

4. Comparación de los distintos algoritmos.

5. Elección de algoritmo adecuado según área y tiempo de ejecución

CAPÍTULO II - CONCEPTOS BÁSICOS

3. Desarrollo del proyecto

Una vez explicado los objetivos del proyecto en primer lugar se explicaran los conceptos básicos para la correcta comprensión del proyecto

Después se presentar los distintos algoritmos desarrollados. En un primer lugar se explicara el funcionamiento básico de las aplicaciones desarrolladas, como los distintos multiplicadores, multiplicadores modulares….

Seguidamente pasaremos a ver, si es necesario, una explicación del código implementado, ya sea con una explicación sobres el mismo código, o mediante un grafico explicativo en el caso de que el código contenga una maquina de estados.

Además de esto se compararan los distintos resultados que hemos obtenido del sintetizador y simulador. Como simulador se ha utilizado el programa informático ModelSim SE plus 6.3f. Como sintetizador se ha utilizado la herramienta de Xilinx – ISE 10.1, para obtener los resultados de la síntesis el programa informático ISE se puede configurar para que la síntesis la simule sobre distintos dispositivos FPGA como lo que se utilizo el modelo de FPGA de la familia Spartan 3, modelo XC3S1500L.

A la hora de manejar el sintetizador, hemos de tomar varias consideraciones.

En primer lugar las FPGA actuales llevan incorporados multiplicadores con lo cual se han tenido que desactivar los multiplicadores para que las comparaciones entre estos algoritmos sean más fidedignas y en algunos casos sean posibles las comparaciones.

Por otro lado Xilin-ISE propone dos tipos de síntesis óptimas. Podemos sintetizar de forma que se intente utilizar la menor área posible o podemos pedirle al sintetizador que intente hacer que el algoritmo funcione de la forma más rápida posible, es decir optimice el tiempo de funcionamiento. En nuestro caso es vemos más necesario optimizar el área ya que es un valor más crítico en el diseño de nuestros sistemas

De las demás opciones que propone el sintetizador, se ha utilizado las que este impone por defecto.

Para poder comparar los distintos algoritmos, hemos decidido ver dos valores fundamentales para el diseño de hardware, el área, que en nuestro caso vendrá reflejada en LUT de 4 bits, un LUT es un registro de una FPGA que contiene una tabla de verdad que dicta el funcionamiento del registro ante distintos estímulos.

El otro valor a comparar es el tiempo, mediante la frecuencia máxima que podemos poner en el reloj del circuito, así como el número de ciclos de reloj necesarios para poder realizar el algoritmo.

Se tendrá en cuenta el ancho de palabra con el cual estamos trabajando, para ello se harán síntesis con anchos de 4, 8, 16, 32, 64, 128, 256, 512 y 1024 bits, además se presentaran las síntesis más significativas para nuestro proyecto, en nuestro casos son las de 256 y 512 bits.

Por último se incluirán algunas simulaciones y simulaciones post síntesis de los distintos algoritmos desarrollados.

4. 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).

El modo de funcionamiento de los sistemas RFID es simple. La etiqueta RFID, que contiene los datos de identificación del objeto al que se encuentra adherido, genera una señal de radiofrecuencia con dichos datos. Esta señal puede ser captada por un lector RFID, el cual se encarga de leer la información y pasársela, en formato digital, a la aplicación específica que utiliza RFID.

Por tanto, un sistema RFID consta de los siguientes tres componentes:

Etiqueta RFID: compuesta por una antena, un transductor radio y un material encapsulado o chip. El propósito de la antena es permitirle al chip, el cual contiene la información, transmitir la información de identificación de la etiqueta. Existen varios tipos de etiquetas. El chip posee una memoria interna con una capacidad que depende del modelo y varía de una decena a millares de bytes.

Dentro del tipos de etiquetas cabe destacar tres tipos los cuales se clasifican según necesiten o no una fuente de energía adicional, podemos hablar de etiquetas activas, las cuales tienen una fuente de energía adicional en la propia etiqueta, semipasivas, las cuales tienen una fuente de energía adicional para el funcionamiento interno de la etiqueta pero no para la respuesta de esta, y por último las pasivas, que no tienen fuente de energía interna.

Lector de RFID: compuesto por una antena, un transceptor y un decodificador. El lector envía periódicamente señales para ver si hay alguna etiqueta en sus inmediaciones. Cuando capta una señal de una etiqueta (la cual contiene la información de identificación de ésta), extrae la información y se la pasa al subsistema de procesamiento de datos.

Subsistema de procesamiento de datos: proporciona los medios de proceso y almacenamiento de datos.

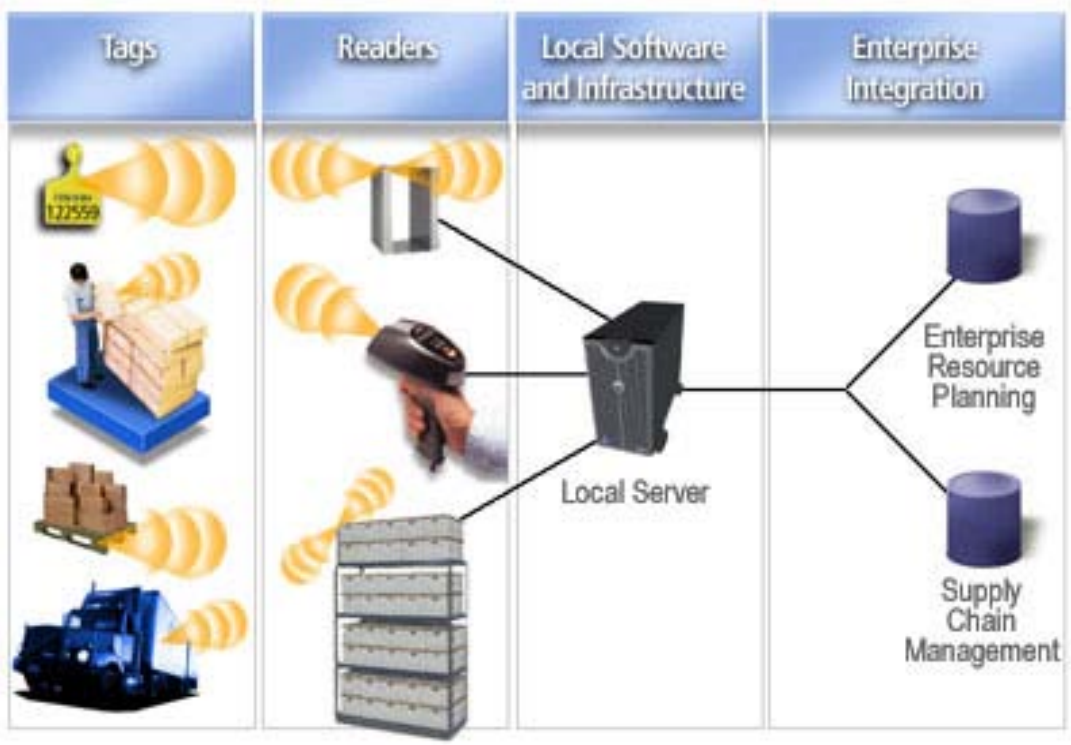

Ilustración 1: Sistema completo RFID

4.1. Tipos de etiquetas

Las Etiquetas RFID son la forma de empaquetar más habitual de los tags RFID. Según varias consultoras líderes en nuevas tecnologías, coinciden en predecir que más del 95% de los tags RFID viajarán con etiquetas en los próximos años, lo que significa que es de suma importancia la calidad de la respuesta que una etiqueta con chip pueda responder a la interacción de las ondas electromagnéticas emitidas por una antena RFID.

Las etiquetas RFID no dejan de ser tags RFID pero con unas connotaciones muy importantes como su flexibilidad, su "delgadez", la capacidad poder ser impresas con código humanamente legible en su cara frontal y las capacidades de memoria dependen del chip que lleve incorporado.

Las etiquetas RFID pueden ser activas, semipasivas (o semiactivas, también conocidas como asistidas por batería) o pasivas.

Las etiquetas pasivas no requieren ninguna fuente de alimentación interna y son en efecto dispositivos puramente pasivos (sólo se activan cuando un lector se encuentra cerca para suministrarles la energía necesaria). Los otros dos tipos necesitan alimentación, típicamente una pila pequeña.

4.1.1. Etiquetas pasivas

Las etiquetas pasivas no poseen ningún tipo de alimentación. La señal que les llega de los lectores induce una corriente eléctrica mínima que basta para operar el circuito integrado de la etiqueta para generar y transmitir una respuesta. La mayoría de las etiquetas pasivas utiliza "backscatter" sobre la portadora recibida. Esto es, la antena ha de estar diseñada para obtener la energía necesaria para funcionar a la vez que para transmitir la respuesta. Esta respuesta puede ser cualquier tipo de información, no sólo un código identificador. Una etiqueta puede incluir memoria no volátil (por ejemplo EEPROM).

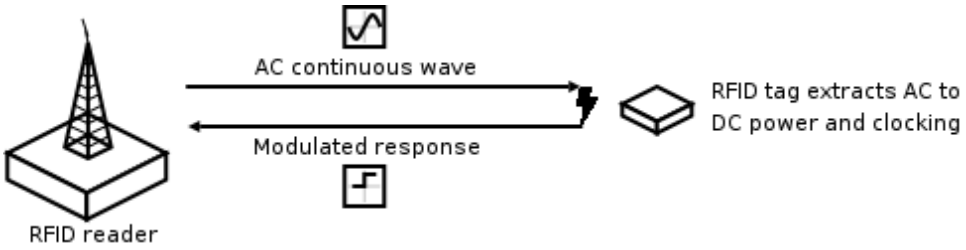

Ilustración 2: Proceso "Backscatter" de una etiqueta pasiva

Las etiquetas pasivas suelen tener distancias de uso práctico comprendidas entre los 10 cm llegando hasta unos pocos metros, según la frecuencia de funcionamiento y el diseño y tamaño de la antena. Por su sencillez conceptual, son obtenibles por medio de un proceso de impresión de las antenas. Como no precisan de alimentación energética, el dispositivo puede resultar muy pequeño: pueden incluirse en una pegatina o insertarse bajo la piel.

La respuesta de una etiqueta pasiva RFID es necesariamente breve, normalmente apenas un número de identificación. La falta de una fuente de alimentación propia hace que el dispositivo pueda ser bastante pequeño: existen productos disponibles de forma comercial que pueden ser insertados bajo la piel.

Las etiquetas pasivas, en la práctica tienen distancias de lectura que varían entre unos 10 milímetros hasta cerca de 6 metros dependiendo del tamaño de la antena de la etiqueta y de la potencia y frecuencia en la que opera el lector.

Como curiosidad, en 2007, el dispositivo disponible comercialmente más pequeño de este tipo medía 0.05 milímetros × 0.05 milímetros, y más fino que una hoja de papel; estos dispositivos son prácticamente invisibles. Este chip puede transmitir un identificador único de 128 bits fijado a él en su fabricación, que no puede modificarse y confiere autenticidad al mismo. Estos chips tienen capacidad de almacenamiento y pueden funcionar en distancias de hasta unos pocos cientos de metros. Su principal inconveniente es que su antena debe ser como mínimo 80 veces más grande que el chip.

Ilustración 3 Encapsulado parar unos en uniformidad y sector textil

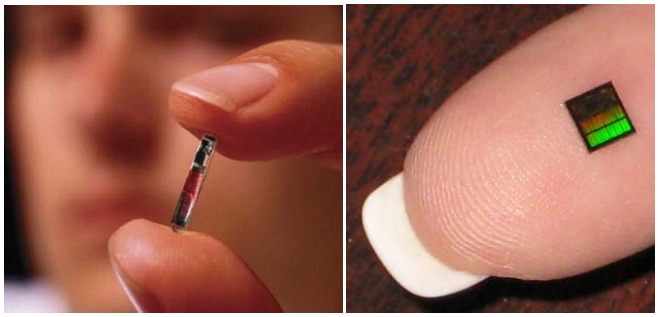

Ilustración 4 Dispositivos pasivos RFID implante subcutáneo y etiqueta pasiva

4.1.2. Etiquetas activas

A diferencia de las etiquetas pasivas, las activas poseen su propia fuente autónoma de energía, que utilizan para dar corriente a sus circuitos integrados y propagar su señal al lector. Estas etiquetas son mucho más fiables que las pasivas debido a su capacidad de establecer conexiones con el lector.

Gracias a su fuente de energía son capaces de transmitir señales más potentes que las de las etiquetas pasivas, lo que les lleva a ser más eficientes en entornos dificultosos para la radiofrecuencia. También son efectivas a distancias mayores pudiendo generar respuestas claras a partir de recepciones débiles. Por el contrario, suelen ser mayores y más caras, y su vida útil es en general mucho más corta.

Muchas etiquetas activas tienen rangos efectivos de cientos de metros y una vida útil de sus baterías de hasta 10 años. Algunas de ellas integran sensores de registro de temperatura y otras variables que pueden usarse para monitorizar entornos de alimentación o productos farmacéuticos. Las etiquetas, además de mucho más rango (500 m), tienen 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.

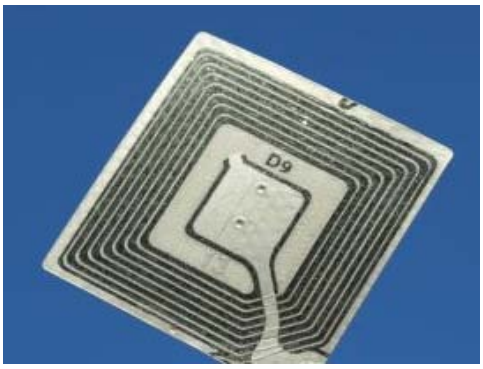

Ilustración 5: Etiqueta activa RFID

4.1.3. Etiquetas Semipasivas

Las etiquetas semipasivas se parecen a las activas en que poseen una fuente de alimentación propia, aunque en este caso se utiliza principalmente para alimentar el microchip y no para transmitir una señal. La energía contenida en la radiofrecuencia se refleja hacia el lector como en una etiqueta pasiva.

Un uso alternativo para la batería es almacenar información propagada desde el lector para emitir una respuesta en el futuro, típicamente usando backscatter. Las etiquetas sin batería deben responder reflejando energía de la portadora del lector al vuelo.

La batería puede permitir al circuito integrado de la etiqueta estar constantemente alimentado y eliminar la necesidad de diseñar una antena para recoger potencia de una señal entrante. Por ello, las antenas pueden ser optimizadas para utilizar métodos de backscattering.

Las etiquetas RFID semipasivas responden más rápidamente, por lo que son más fuertes en el ratio de lectura que las pasivas.

Este tipo de etiquetas tienen una fiabilidad comparable a la de las etiquetas activas a la vez que pueden mantener el rango operativo de una etiqueta pasiva. También suelen durar más que las etiquetas activas.

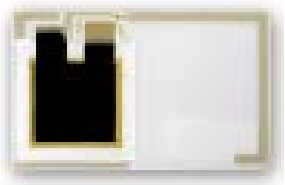

Ilustración 6: Etiqueta semipasiva RFID

4.2. Lectores RFID

El lector utiliza su antena para enviar información digital codificada a través de ondas de radiofrecuencia. Un circuito receptor en la etiqueta es capaz de detectar el campo modulado, decodificar la información y usar su propia antena para enviar una señal más débil a modo de respuesta.

Los componentes de un lector son el módulo de radio, el procesador y sus conexiones que pueden ser de varios tipos para conectar con distintos dispositivos (conector de antena, Ethernet, RS232, actuadores, sensores u otros).

Los lectores RFID se encuentran disponibles para realizar lecturas móviles con un operador, montados en unos autos elevadores o instalados en forma fija.

Ilustración 7: distintos tipos de lectores RFID

Debido a que una gran cantidad de etiquetas que podrían encontrarse en presencia de un lector, los lectores deben ser capaces de recibir y administrar varias respuestas al mismo tiempo (potencialmente cientos por segundo). La capacidad de gestionar una gran cantidad de etiquetas es utilizada para permitir que las etiquetas sean identificadas y seleccionadas individualmente.

Para leer los datos de los etiquetas, los lectores utilizan un algoritmo de encubrimiento basado en el recorrido de árboles, resolviendo las colisiones que puedan darse y procesando secuencialmente las respuestas. Existen etiquetas bloqueantes que pueden usarse para evitar que haya lectores que accedan a las etiquetas de otras aéreas, que no pertenezcan al lector.

El lector puede instruir algunas etiquetas para que se enciendan y otros para que se apaguen con el objetivo de suprimir las interferencias. Una vez que la etiqueta es seleccionada, el lector está habilitado para realizar un número de operaciones tales como leer su número de identificación o escribir información en la etiqueta, dependiendo de la aplicación.

4.3. Clasificación de los sistemas RFID

Como hemos visto anteriormente los sistemas RFID se clasifican dependiendo del rango de frecuencias que usan. Existen cuatro tipos de sistemas: de frecuencia baja (entre 125 ó 134,2 kilohercios); de alta frecuencia (13,56 megahercios); UHF o de frecuencia ultra elevada (868 a 956 megahercios); y de microondas (2,45 gigahercios). Los sistemas UHF no pueden ser utilizados en todo el mundo porque no existen regulaciones globales para su uso.

4.4. Seguridad y RFID

El gran problema que se plantea en esta tecnología es la falta de seguridad ante los ataque externos. En un principio esta tecnología se aplicara en campos tan distintos, en donde los datos que se trasfieran sean de vital importancia en cuanto a la privacidad del sujeto.

Hay que tener en cuenta que las etiquetas RFID en un futuro serán utilizadas en ámbitos tales como los controles de entrada, pasaportes, identificación personal, servicios médicos, con lo que el elegir un buen método de seguridad, será necesario para que esta tecnología prospere.

Partiendo de este punto y viendo cómo actúan los diferentes protocolos de seguridad, casi todos trabajan con operaciones aritméticas, pasamos a presentar los algoritmos necesarios que se dan en estos protocolos

5. Algoritmos Matemáticos

En este punto del proyecto analizaremos los distintos algoritmos que podemos usar para realizar las distintas operaciones matemáticas, necesarias en los distintos protocolos de encriptación.

Dentro de las posibles operaciones matemáticas que utilizan los protocolos de encriptación, para realizar nuestro proyecto hemos visto conveniente seleccionar las siguientes operaciones.

En primer lugar implementaremos algoritmo que realicen multiplicaciones, este algoritmo es el más sencillo que implementaremos, este tipo de algoritmos no son muy utilizados en protocolos de encriptación.

Las multiplicaciones modulares, son operaciones mucho más complicadas de realizar, este tipo de multiplicaciones se pueden realizar por algoritmos directos, es decir, realizan la multiplicación modular de un solo paso, o los métodos indirectos en los cuales en primer lugar se realizan una multiplicación y seguidamente un reducción.

Por último exponenciadores modulares, son los más complejos de realizar de los algoritmos matemáticos, pero son los algoritmos que actualmente se utilizan diferentes protocolos de encriptación.

Con lo que utilizaremos los siguientes algoritmos matemáticos:

Métodos de multiplicación:

1- Multiplicador secuencial por el método “Add & Shift”
2- Multiplicador por el método de “Booht”
3- Multiplicador Combinacional mediante librerías de VHDL

Para realizar los algoritmos de multiplicación modular se han elegido dos métodos para su implementación, para el primer método se ha decidido utilizar los multiplicadores anteriores y posteriormente realizar una reducción, de los cuales se ha decidido implementar alguna de las reducciones por los algoritmos de "Barret", de los cuales se han elegido los siguientes

:

Métodos de reducción:

1- Algoritmo "Naive Reduction"
2- Algoritmo "Non Restore Ring".

El segundo método implementado para realizar la multiplicación modular son los algoritmos directos de multiplicación modular para los cual se ha elegido el algoritmo de Montgomery:

Método directo:

1- Algoritmo de Montgomery.

Por último para realizar la exponenciación modular, hemos elegido el algoritmo de exponenciación modular binario denominado "left to right", este método requiere varias multiplicaciones modulares, con lo que para realizarlas se utilizara el algoritmo de Montgomery, que es el que requiere el algoritmo "left to right".

5.1. Algoritmos de multiplicación

En este punto del proyecto pasamos a ver los distintos tipos de algoritmos de multiplicación que se han seleccionado anteriormente, posteriormente se realizar un estudio de estos para ver cuál de ellos se adapta mas a nuestras necesidades.

5.1.1. Multiplicador por desplazamiento Add & Shift

Este algoritmo matemático de multiplicación es el que solemos utilizar cuando queremos multiplicar dos números expresados en base decimal. Si lo extrapolamos a una base binaria, el algoritmo para poder multiplicar dos números binarios es el siguiente.

Pariendo de un multiplicando “a”, y un multiplicador “b”, numero binarios de n-bits, y siendo “p” el resultado de la multiplicación con un tamaño de 2*n bits, el algoritmo de multiplicación se realiza de la siguiente forma

				a3	a2	a1	a0
				X b3	b2	b1	b0
0	0	0	0	**p03**	**p02**	**p01**	**p00**
0	0	**0**	**p13**	**p12**	**p11**	**p10**	0
0	0	**p23**	**p22**	**p21**	**p20**	0	0
0	**p43**	**p42**	**p41**	**p40**	0	0	0
p7	**p6**	**p5**	**p4**	**p3**	**p2**	**p1**	**p0**

Ilustración 8: Multiplicación suma y desplazamiento

Ahora pasamos a ver un diagrama de bloques para ver de una forma más clara como se comporta el algoritmo “Add & Shift”. Este algoritmo solo contiene un registro de desplazamiento, para desplazar el multiplicando y registro un sumador, donde se van realizando las sumas parciales. El control se realiza por medio de una maquina de estados y un contador.

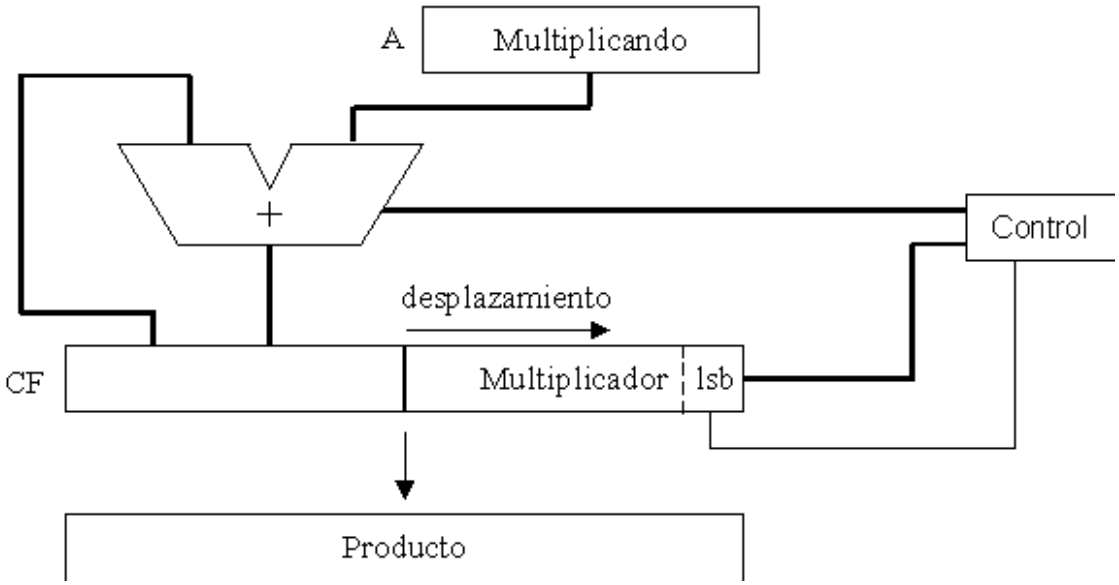

Ilustración 9: Diagrama de bloques multiplicador Add & Shift

Esta figura muestra un diagrama a bloques del procedimiento de multiplicación. Se coloca el multiplicando en un registro A y el multiplicador se coloca en un registro CF del doble de la longitud del multiplicador (la parte alta se llena con ceros). El bloque de control analizará el Bit menos significativo del registro B, si este es uno, realizará la suma de la parte alta del registro CF mas el registro A y después un desplazamiento, si el Bit es cero, solamente realizará el desplazamiento. Después de n ciclos (con n igual al número de bits del multiplicador) el resultado queda disponible en los 2n bits de orden bajo del registro CF.

5.1.2. Multiplicador Booth

Ahora pasaremos a explicar el algoritmo de la multiplicación de Booth este algoritmo permite multiplicar números con y sin signo, con lo que puede ser interesante para algunas otras aplicaciones, pero lo que nos conlleva en nuestra aplicación esta característica no nos es relevante, ya que en principio solo multiplicaremos números positivos.

Pariendo de un multiplicando “a”, y un multiplicador “b”, numero binarios de n-bits, y siendo “p” el resultado de la multiplicación con un tamaño de 2*n bits, el algoritmo de multiplicación se realiza de la siguiente forma.

Para realizar el algoritmo de forma que cuando realicemos la multiplicación, nos fijaremos en el multiplicador viendo los bits de dos en dos: bi y bi-1, de forma tenemos cuatro posibles secuencias, determinarán el valor de b’i, con lo que codificaremos el multiplicando ‘b’ realizaremos las acciones indicadas:

00 ó 11: b’i = 0: Solo desplazaremos el multiplicador --> poner ceros.

01: b’i = 1: Realizaremos el producto por 1 y desplazado.

10: b’i = -1: Realizaremos el complemento a dos del multiplicador con extensión de signo y desplazado.

Pero surge el problema del primer bit, ya que es necesario coger parejas de bits, para lo cual introducimos un bit previo a b0, el b-1, este bit será un cero.

Veamos un ejemplo para entender mejor este procedimiento de multiplicación.

a =0 1 0 1 1 0 1 (45)

b =0 0 1 1 1 1 0 (30)

Codificamos “b” como ya se ha indicado:

b’ = 0 1 0 0 0 -1 0

Luego realizamos la operación de sumas parciales como en el caso del multiplicador por sumas parciales haciendo el complemento a dos de los multiplicandos que sean necesarios restar.

```
                  0 1 0 1 1 0 1
                  0+1 0 0 0-1 0
0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 0 1 0 0 1 1   (complemento a dos)
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 1 0 1
0 0 0 0 0 0 0 0
0 0 0 1 0 1 0 1 0 0 0 1 1 0
```

Ilustración 10: Algoritmo Booth

Ahora pasamos a ver un diagrama de bloques para ver de una forma más clara como se comporta el algoritmo de "Booth". Este algoritmo solo contiene un registro de desplazamiento, para desplazar el multiplicando y registro un sumador-restador, donde se van realizando las sumas y restas parciales. El control se realiza por medio de un contador y una maquina de estados, esta analizara la codificación, como la operación a realizar según sea el multiplicador.

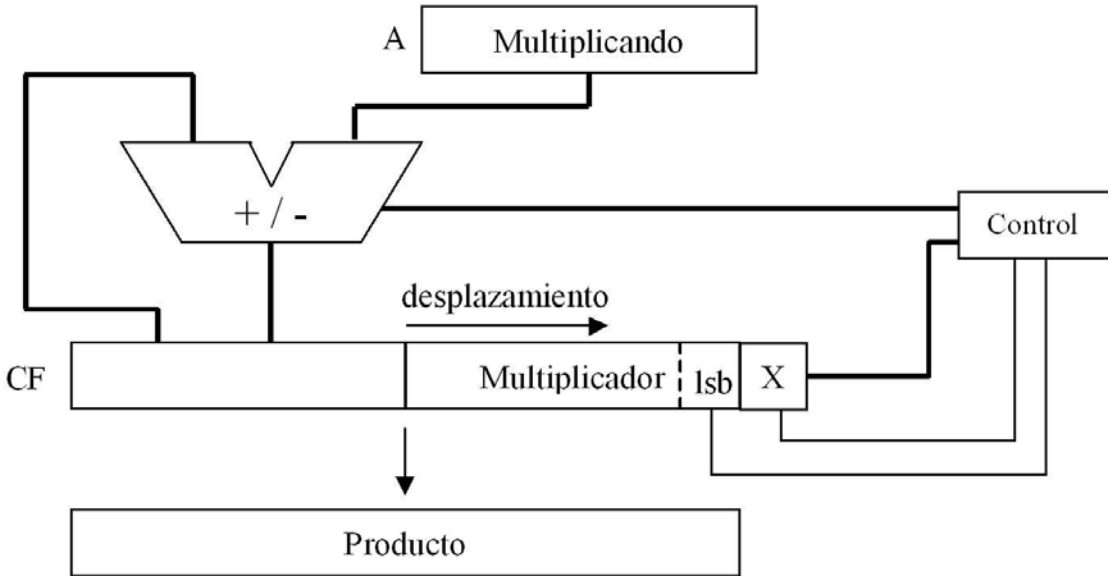

Ilustración 11: Diagrama de bloques multiplicador Booth

La figura muestra el diagrama de bloques del proceso de multiplicación por el algoritmo de "Booth", el funcionamiento es muy similar al de la multiplicación clásica pero en este caso, el bloque de control debe revisar dos bits y decidir si se debe realizar una suma, una suma en complemento a dos o simplemente un desplazamiento

Para la implementación de este algoritmo es necesario hacer una pequeña modificación al dato del multiplicador, como se dijo anteriormente, el análisis de "Booth" hay que agregarle un bit al multiplicando para que se realice la operación d forma correcta.

5.1.3. Multiplicador AxB

Por último realizaremos un multiplicador basándonos en el lenguaje de programación VHDL.

Este multiplicador es el multiplicador más simple que podemos implementar en VHDL, para realizar el algoritmo solo es necesario utilizar las librerías pertinentes donde se encuentran las diferentes operaciones matemáticas, con lo que este multiplicador se reduce a la siguiente línea de lenguaje

S<= a*b;

Si configuramos de forma correcta el programa con el cual realizaremos las diferentes síntesis del circuito, conseguimos un multiplicador totalmente combinacional, que solo necesita un ciclo de reloj para dar el resultado,

Al igual que en los demás multiplicadores, las variables “a” multiplicando y “b” multiplicador, tiene un ancho de palabra “n” bits, el resultado “s”, tiene un ancho de palabra "2*n” bits.

5.2. Multiplicadores modulares “C= A mod B”

En este punto del proyecto pasamos a analizar distintos algoritmos que producen la operación “P = AxB mod M”.

La multiplicación modular 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.

Ilustración 12 multiplicación modular

Hemos analizado tres tipos de algoritmos distintos, agrupados en dos grandes bloques, podemos hablar de los métodos indirectos, usando los diferentes multiplicadores con un reductor, para la reducción se ha utilizad los algoritmos de reducción de Barret (Barret reduction Method), y dentro de sus distintos algoritmos hemos utilizado el Naive reduction y el algoritmo Restore Ring Reduction. Por otro lado hemos implementado el algoritmo de multiplicación modular de Montgomery, que pertenece a los métodos directos.

La gran diferencia entre estos dos bloques, Barret y Montgomery, es que el método de Montgomery realiza la multiplicación modular de forma directa, en cambio el método de Barret solo realiza lo operación de reducción sin la multiplicación, con lo que es necesario añadir un multiplicador de los anteriores para poder realizar una multiplicación modular.

Como en el caso del Barret es necesario utilizar un multiplicador adicional, con lo que para comparar con el algoritmo de Montgomery será necesario añadir un el multiplicador “Add & Shift”, o el multiplicador “a*b”, que son los mas característicos.

A continuación pasaremos a explicar el funcionamiento de los distintos algoritmos para poder realizar un análisis de cuál es la mejor combinación posible para nuestro propósito.

5.2.1. Barret Reduction Method

Ahora veremos algunos de los algoritmos de “Barret” para realizar la reducción de la operación de multiplicación. Veremos los algoritmos de “Naive” y el algoritmo “No Restore Ring”.

5.2.1.1. Naive Reduction

Una vez realizada la multiplicación podemos pasar a realizar la reducción para ello existen varios métodos pero en nuestro caso expondremos dos, en primer lugar el algoritmo para la división “Naive Reduction”.

```
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
```

Ilustración 13 Algoritmo Naive

Este algoritmo hace restas sucesivas, si tenemos que ‘P’ es el resultado de la multiplicación que anteriormente hemos realizado, ‘M’ es el dividendo y ‘R’ es el resto de la división, con lo que “R= P mod M”. Obtendremos ‘R’ cuando se produzca la enésima resta tal que ‘M’ sea mayor que ‘P’.

Para entender mejor el funcionamiento de algoritmo de Naive, pasamos a ver el diagrama de bloques.

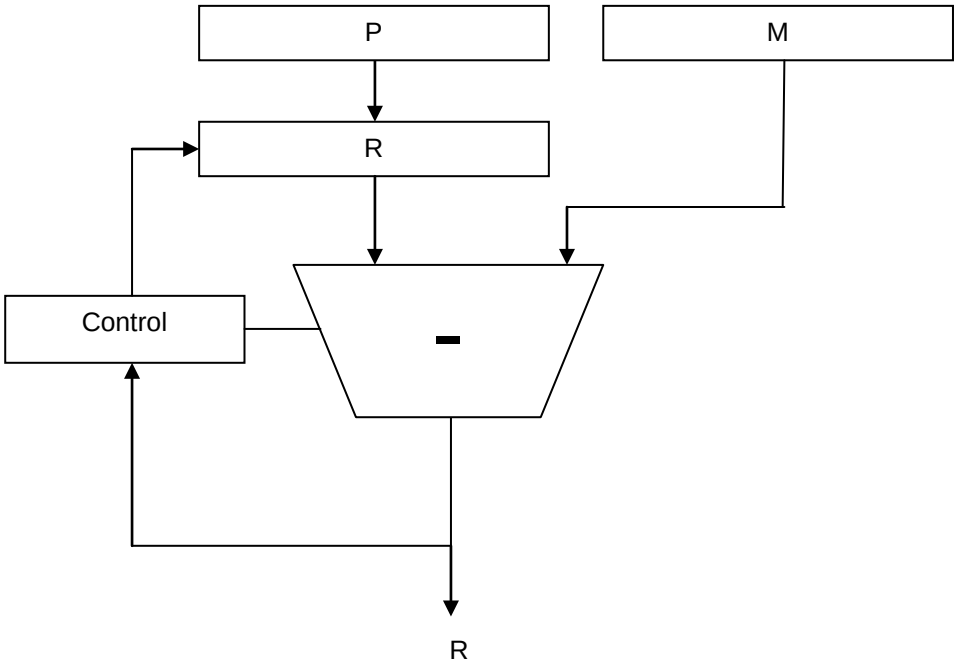

Ilustración 14: Diagrama de bloques reductor Naive

Como podemos ver este tipo de algoritmo se basan en un registros, y un simple restador, el control se encarga de ver si el resultado 'R' es más pequeño que el modulo 'M', con lo que si esto ocurre se dará por finalizada la operación de reducción.

El principal problema de este algoritmo esta, en que según sea el valor de los operandos, así será el tiempo que tarde en arrojar un resultado, este método realiza restas sucesivas, hasta "R" es más pequeño que el modulo, en este caso la variable "M", con lo que cuanto mayor sea la diferencia entre estos dos números, mayor será el tiempo en el que obtengamos un resultado correcto.

5.2.1.2. No Restore Ring Reduction

Como anteriormente hemos comentado, una vez utilizado alguno de los algoritmos anteriores de multiplicación podemos pasar a realizar la reducción, para poder obtener una multiplicación modular, en este caso utilizamos el método de “No Restore Ring Reduction” que a continuación pasamos a explicar.

Nuestro objetivo es hacer la operación matemática c= q mod d, siendo ‘q’ el resultado de la multiplicación citada, que se debe hacer anteriormente, además del modulo, ‘d’ numero binario cualquiera.

La forma más sencilla del explicar este algoritmo es mediante su flujograma acompañado de un ejemplo, y no mediante su diagrama de bloques, ya que resulta demasiado complicado. Pasamos a explicar el procedimiento con el siguiente flujograma y posteriormente veremos un ejemplo detallado.

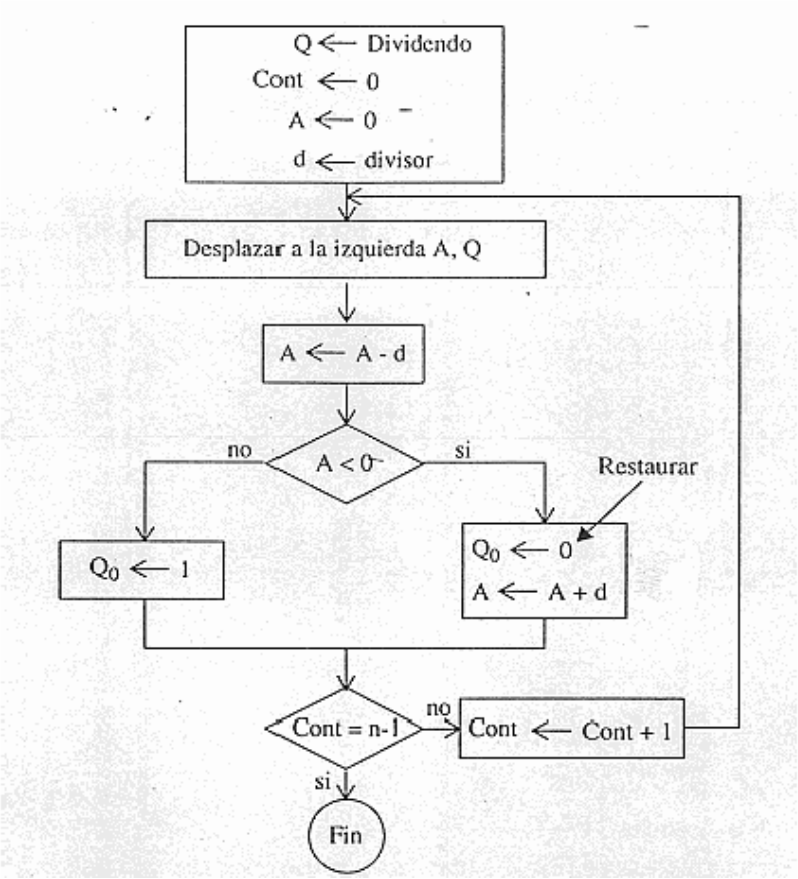

Ilustración 15: Flujograma algoritmo No Restore Ring Reduction

Podemos ver un ejemplo de aplicación del método Restore Ring Reduction con los siguientes datos. La operación a realizar será la siguiente:

A =Q mod d

n: numero de bits = 4.
A: registro auxiliar de n +1 bits, en donde se almacenara el resto, donde * corresponde al bit 'A'(n+1).
Q: resultado de la multiplicación anterior = 11.
d: denominador = 5
cont: contador.

Paso	Acción *		A	Q	d	Cont
0	Inicializar registros	0	0000	1011	0101	0
1	Desplazar A,Q a izqda	0	0001	0110		
	$A \leftarrow A - d = A + C2(d)$	1	1100	0110		
	$A < 0 \Rightarrow Q_0 = 0$	1	1100	0110		
	$A \leftarrow A + d$ (Restaurar)	0	0001	0110		
	$Cont \leftarrow Cont + 1$					1
2	Desplazar A,Q a izqda	0	0010	1100		
	$A \leftarrow A - d = A + C2(d)$	1	1101	1100		
	$A < 0 \Rightarrow Q_0 = 0$	1	1101	1100		
	$A \leftarrow A + d$ (Restaurar)	0	0010	1100		
	$Cont \leftarrow Cont + 1$					2
3	Desplazar A,Q a izqda	0	0101	1000		
	$A \leftarrow A - d = A + C2(d)$	0	0000	1000		
	$A \geq 0 \Rightarrow Q_0 = 1$	0	0000	1001		
	$Cont \leftarrow Cont + 1$					3
4	Desplazar A,Q a izqda	0	0001	0010		
	$A \leftarrow A - d = A + C2(d)$	1	1100	0010		
	$A < 0 \Rightarrow Q_0 = 0$	1	1100	0010		
	$A \leftarrow A + d$ (Restaurar)	0	0001	0010		
	Fin		Resto	Cociente		

1=11 mod 5.

Ilustración 16: Ejemplo algoritmo No Restore Ring Reducction

Con lo que vemos el comportamiento de este algoritmo, de forma que se ha realizado un ejemplo de cómo funciona el algoritmo "No Restore Ring Reduction".

5.2.2. Algoritmo de Montgomery

A continuación pasamos a explicar el funcionamiento del algoritmo de Montgomery, este algoritmo de multiplicación modular se basa principalmente en dos paso, el primero de ellos en el que calcula el producto P = A x B, en el segundo paso realiza la reducción P mod M.

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

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

Ilustración 17: Requisitos algoritmo Montgomery

Estos enteros deben cumplir una serie de condiciones, Las condiciones del método de Montgomery son las siguientes:

El multiplicando y el multiplicador necesitan ser más pequeños que M.

Como se usa la representación binaria en los operandos, el modulo M necesita ser impar para satisfacer la primera condición.

Como hemos apuntado anteriormente, este método se da en dos pasos con lo que el algoritmo para la multiplicación y primera reducción 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.
```

Ilustración 18: Algoritmo Montgomery

Es necesario aplicar este algoritmo dos veces para poder realizar de forma correcta la multiplicación modular. Con lo que el algoritmo que hemos utilizado es el siguiente.

```
algorithm ModularMult(A, B, M, n)
   const C := 4^n mod M;
   int R := 0;
   R := Montgomery(A, B, M);
   return Montgomery(R, C, M);
end ModularMult.
```

Ilustración 19: Algoritmo multiplicación modular de Montgomery

Hay que tener en cuenta que la constate 'C' hay que calcularla anteriormente, es decir esta será una constate integrada en el sistema, esto es posible ya que en principio tanto 'n' numero de bits de los operandos, como 'M' son valores fijos. Por otro lado solo es necesaria la aplicación del primer algoritmo dos veces para que la multiplicación modular de Montgomery sea correcta.

6. Exponenciación modular

En este punto pasamos a ver el último punto de nuestro estudio, en los puntos anteriores hemos estudiados distintos métodos para realizar, en primer lugar, multiplicadores. Basándonos en estos algoritmos hemos realizado varios multiplicadores modulares, y por ultimo vamos a realizar un exponenciador modular.

Para realizar la exponenciación modular se ha elegido el algoritmo de exponenciación modular binaria “Left to Rigth”, el cual pasamos a detallar en el siguiente grafico.

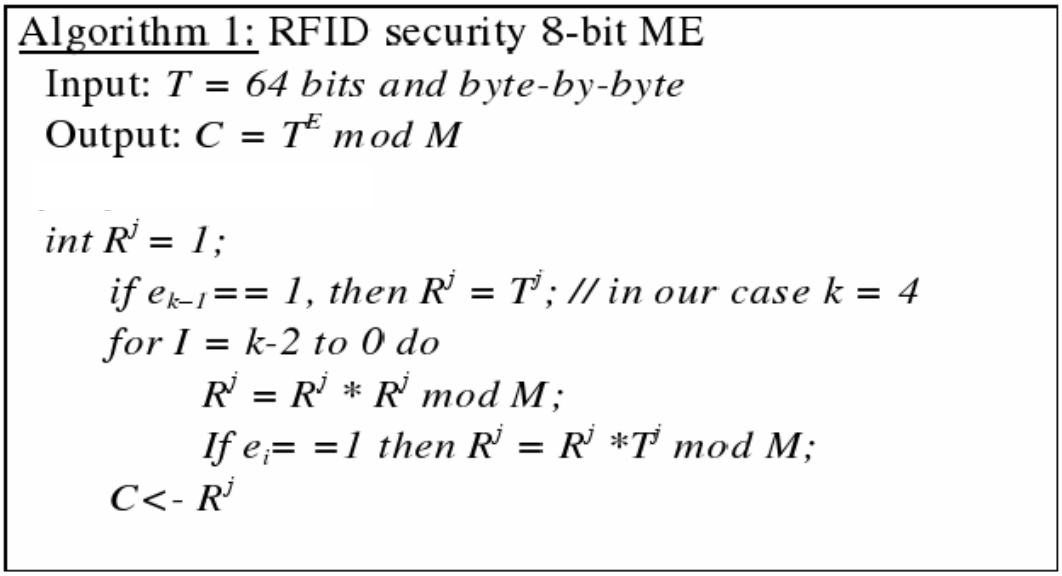

Ilustración 20: Algoritmo Exponenciación modular

Podemos ver en primer lugar que la entrada es un vector de 64 Bits, el cual va entrando en el algoritmo en Bytes, es decir, en ocho partes de ocho Bits, en nuestro caso para realizar el algoritmo de una forma más practica hemos decidido que la entrada sea solo un vector de 8 Bits, con lo que la primera instrucción del algoritmo no la hemos implementado.

Por otro lado la salida del algoritmo, ‘C’, será del mismo ancho de palabra que la entrada. También hay que tener en cuenta que el exponente ‘E’, es un vector de 3 bits. Otro requisito a cumplir es que ‘M’ tiene que ser un número de n bits (ancho de palabra igual que la salida y entrada) que cumpla que sea impar y primo.

CAPÍTULO III -IMPLEMENTACIÓN DE HARDWARE

7. Implementación de Hardware

En este capítulo del proyecto pasamos a ver las distintas soluciones que se han tomado para implementar los diferentes algoritmos.

7.1. Implementación de Multiplicadores

En este punto del proyecto pasamos a ver los distintos tipos de algoritmos de multiplicación que se van a implementar para realizar un estudio de estos y ver cual de ellos se adapta mas a nuestras necesidades

.

7.1.1. Multiplicador “Add &Shift”

Este algoritmo matemático de multiplicación es el que solemos utilizar cuando queremos multiplicar dos números expresados en base decimal. Si lo extrapolamos a una base binaria, el algoritmo sirve para poder multiplicar números binarios.

En primer lugar veremos un flujograma, con este flujograma nos apoyaremos para realizar la implementación de este algoritmo en el lenguaje VHDL.

Con el cual podremos realizar más tarde la implementación del hardware.

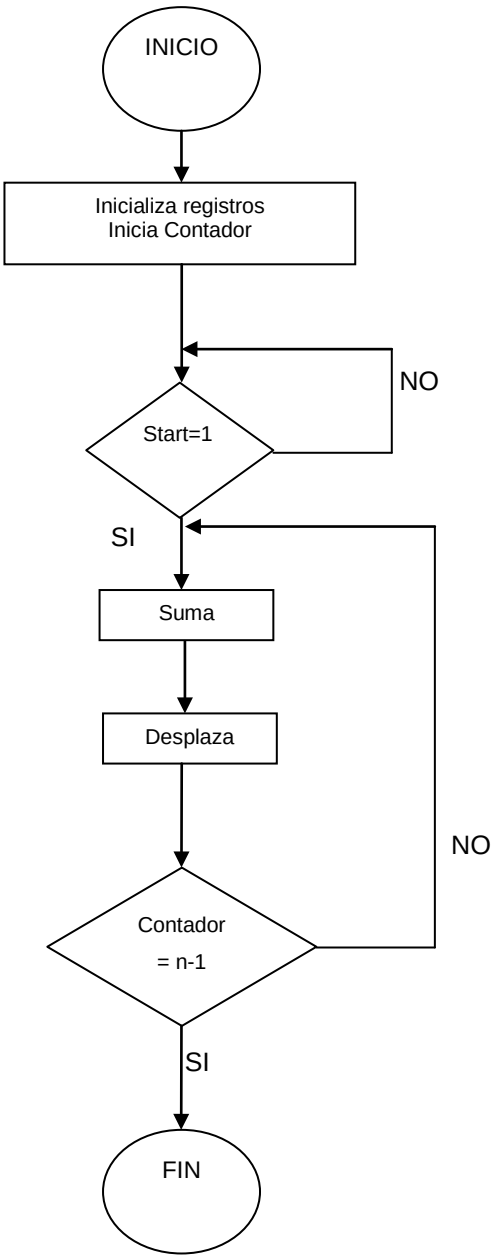

Ilustración 21 : Flujograma multiplicador Add & Shift

Para implementar este algoritmo en VHDL se pueden tomar varias alternativas, hemos elegido realizar una maquina de estados para poder implementar el algoritmo “Add & Shift”. De la máquina de estados podemos destacar el siguiente grafico en el cual se representan los estados por los cuales pasa el proceso de multiplicación.

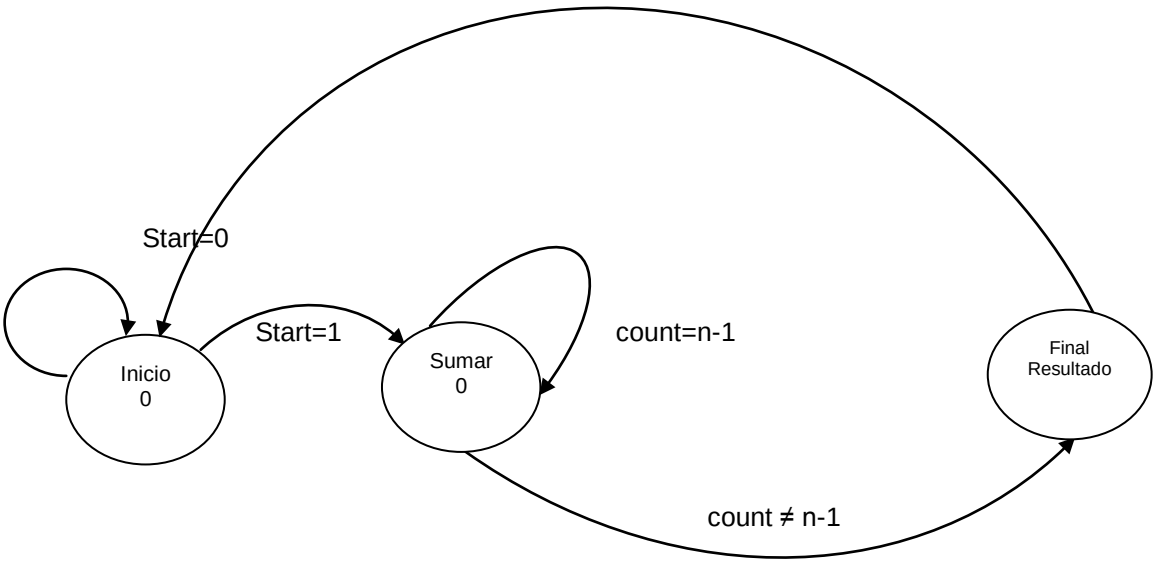

Ilustración 22: Máquina de estados Add & Shift

La parte más importante del código es la que pertenece al funcionamiento de la máquina de estados en la siguiente ilustración podemos ver este código.

```
PROCESS(actual,a,b,start,count,sum)

BEGIN

        CASE actual IS

        WHEN inicio =>
        fin<='0';
        resultado<=(OTHERS=>'0');
        enable<="10";

        IF start = '1' THEN --si hay un '1' en la entrada 'start', comienza el
proceso
                ff<="00";
                siguiente<=sumar;

        ELSE
                ff<="00";
                siguiente<=inicio;
        END IF;

        WHEN sumar =>
        fin<='0';
        enable<="01";
        resultado<=(OTHERS=>'0');

        IF count < n-1 THEN --¿contador desbordado?

                IF b(count)= '1' THEN -- Si un Bit de B = 1
                ff<="01";-- Suma y desplaza
                siguiente<=sumar;

                ELSE
                ff<="10"; -- Suma cero y desplaza
                siguiente<=sumar;

                END IF;

        ELSE

                IF b(count)= '1' THEN
                ff<="01";
                siguiente<=final;

                ELSE
                ff<="10";
                siguiente<=final;

                END IF;
        END IF;

        WHEN final =>
        fin<='1';
        enable<="11";
        resultado<=sum;
        ff<="11";
        siguiente<=inicio;

        END CASE;


END PROCESS;
```

Ilustración 23: Resumen Código “Add & Shift”

Ahora podemos ver un grafico de la entidad que produce el sintetizador, con sus entradas y salidas, según la entidad que hemos descrito en el código.

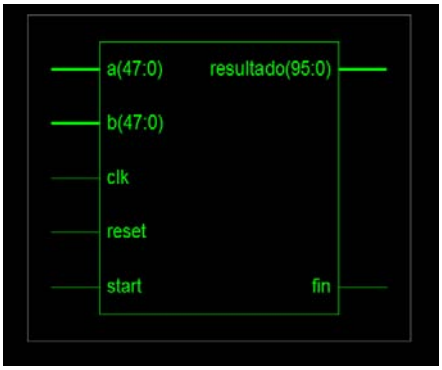

Ilustración 24: Entidad Add & Shift

Por último pasamos a ver la simulación del circuito que se ha implementado, para ello representaremos las señales de control y las señales de la máquina de estados para que se pueda apreciar el funcionamiento del circuito.

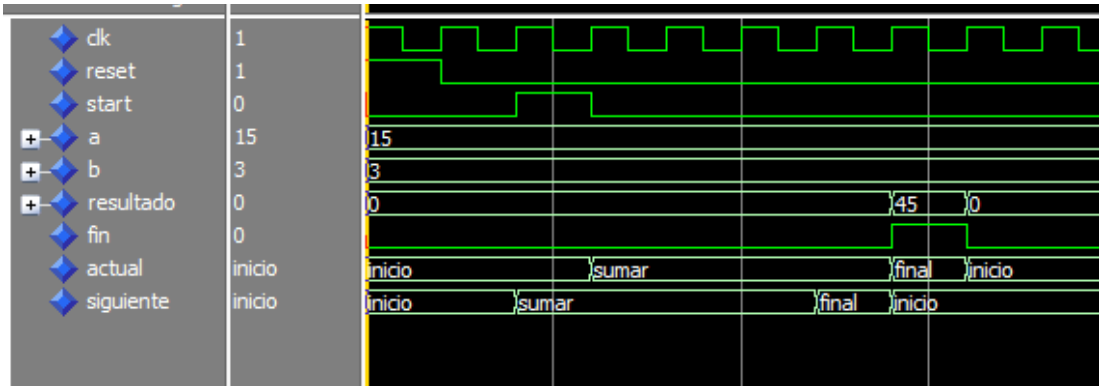

Ilustración 25: Simulación algoritmo Add & Shift de 4 bits

7.1.2. Multiplicador "Booth"

Una vez explicado cómo funciona el algoritmo pasamos a implementarlo en VHDL, hemos optado por crear una maquina de estados, ya que en principio es la forma más sencilla de crear la secuencia de pasos necesaria para realizar el algoritmo, con lo cual pasamos a ver el siguiente grafico en el cual vemos el funcionamiento de la máquina de estados.

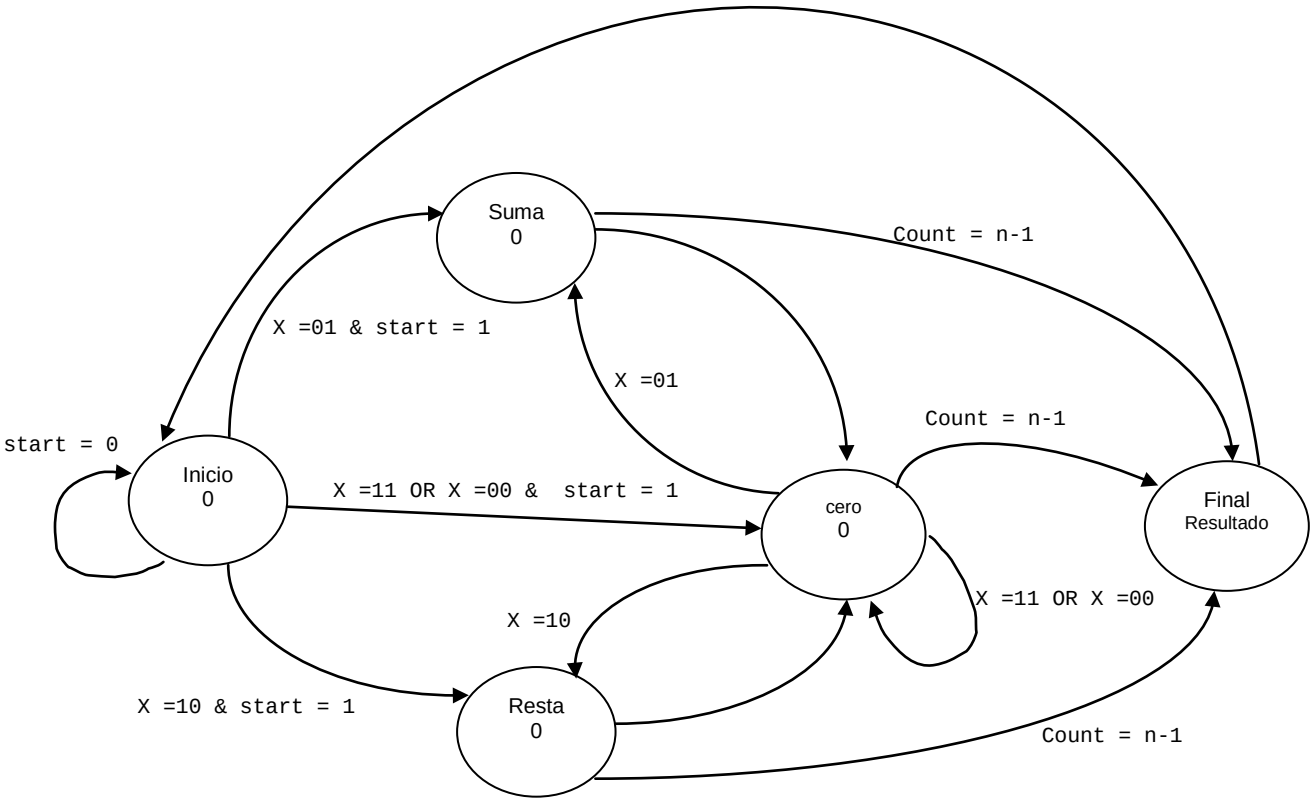

Ilustración 26: Máquina de estados Add & Shift

Una parte interesante del código, es como se ha realizado la codificación de la variable "b", ahora presentamos una pequeña parte del código para ver esta codificación.

```
VARIABLE x:STD_LOGIC_VECTOR(n DOWNTO 0);--variable donde se guardan 2 bits de 'b'

        BEGIN

        x:=b&'0';

        CASE actual IS

        WHEN inicio =>
        fin<='0';
        resultado<=(OTHERS=>'0');
        enable<="00"; -- contador parado
        IF start = '0' THEN
                ff<="011";-- determina la operacion a realizar
                siguiente<=inicio;

        ELSE -- El valor de 'X' determina el comportamiento.
                IF x(count+1 downto count) = "01" THEN
                        ff<="000";
                       siguiente<=suma;

                ELSIF x(count+1 downto count) = "10" THEN
                        ff<="000";
                       siguiente<=resta;
                ELSE
                        ff<="000";
                       siguiente<=cero;

                END IF;
        END IF;
```

Ilustración 27: Codificación del dato 'b'

Como vemos en el cuadro anterior, cabe destacar para entender la codificación de la variable, que se utiliza la señal 'count' para recorrer los distintos pares de bits de la variable 'b', esta señal lleva la cuenta de un contador que se encuentra en el mismo circuito.

Esta parte de código pertenece a la parte combinacional de la máquina de estados, donde se define el comportamiento del estado 'inicio', con lo que podemos ver que si la señal 'start', que es una señal de entrada, no está activada, el proceso no arranca.

Ahora podemos ver un grafico de la entidad que produce el sintetizador, con sus entradas y salidas, según la entidad que hemos descrito en el código.

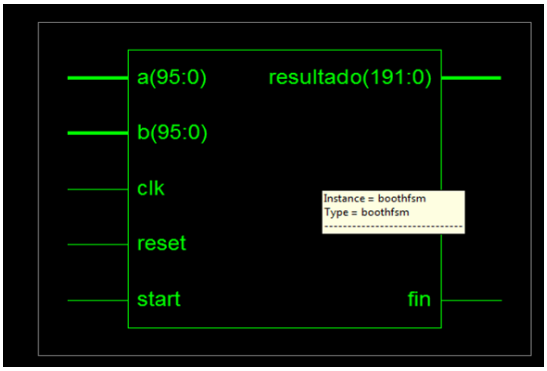

Ilustración 28: Entidad Booth

Por último veremos una simulación del funcionamiento del circuito.

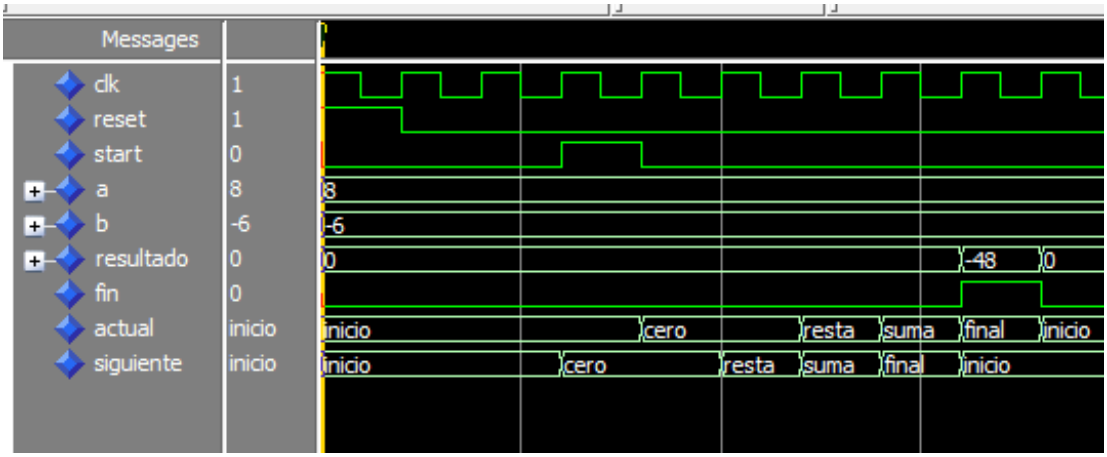

Ilustración 29: Simulación algoritmo Booth 4 Bits

7.1.3. Multiplicador "AxB"

Este multiplicador esta realizado con las librerías propias del lenguaje VHDL. El código pertinente lo podemos ver en el anexo de códigos del proyecto.,

Como en los demás algoritmos podemos ver un grafico de la entidad que produce el sintetizador, con sus entradas y salidas, según la entidad que hemos descrito en el código.

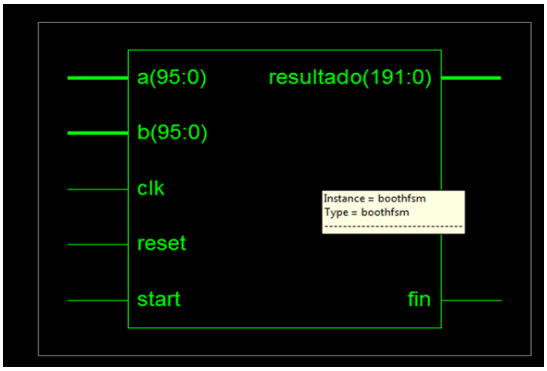

Ilustración 30: Entidad AxB

Por último podemos ver una simulación del funcionamiento del circuito, en el cual se ve perfectamente que este multiplicador realiza la multiplicación en un solo ciclo de reloj.

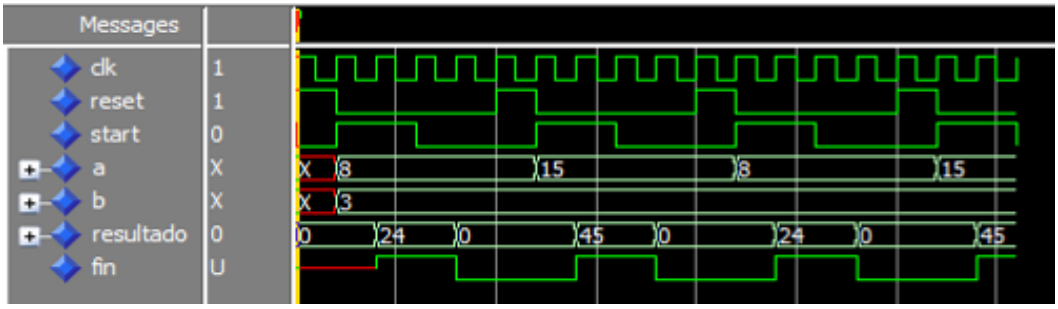

Ilustración 31: Simulación algoritmo AxB 4 Bits

7.2. Implementación Reductores "Barret"

Ahora veremos cómo hemos implementado los algoritmos de "Barret". Estos algoritmos realizan operaciones de reducción. Veremos los algoritmos de "Naive" y el algoritmo "No Restore Ring".

7.2.1. Algoritmo "Naive"

La implementación de este algoritmo se puede ver en el siguiente flujograma, con lo que de esta forma podremos entender de una mejor forma entender la implementación de hardware realizada para el dispositivo FPGA.

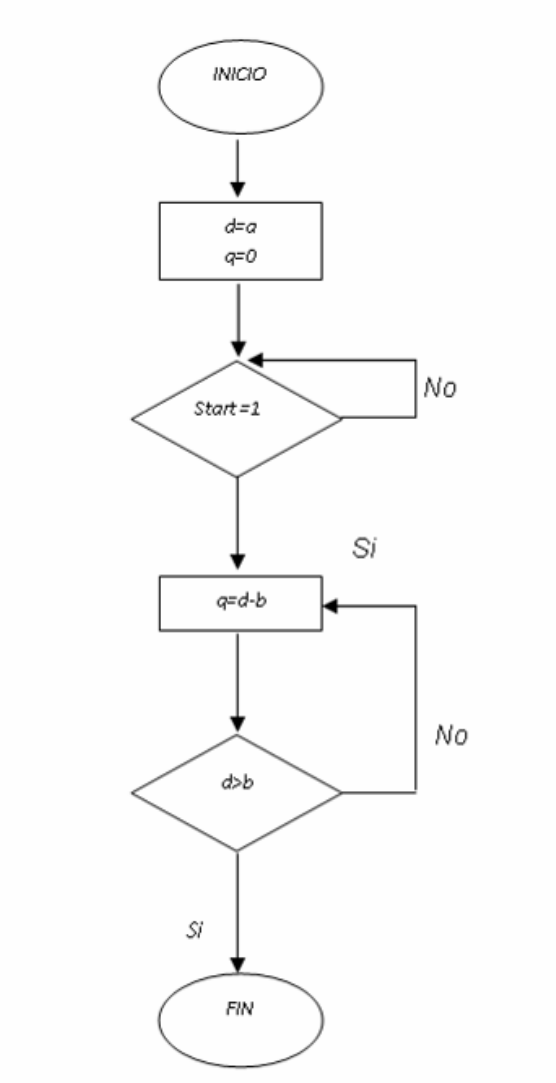

Ilustración 32: Flujograma algoritmo Naive

Para poder realizar de forma correcta el circuito hemos utilizado un registro síncrono auxiliar, con una entrada llamada ‘d’ y su salida correspondiente ‘q’. La única peculiaridad de este tipo de algoritmos está en que el tiempo en que tarda en realizar la operación es variable, y este depende de los operandos ‘a’ y ‘b’, ya que pongamos un ejemplo, no tardara lo mismo en realizar la operación si a= 16 y b = 4, en el cual tardara 4 ciclos de reloj, que si a sigue siendo igual a 16 y en cambio b = 1, con lo que tardaremos 16 ciclos de reloj en obtener el resultado. En la siguiente imagen se puede ver el código del algoritmo, ver claramente como se da esta peculiar dependencia entre el tamaño de los operando y el tiempo que tarda el algoritmo en dar un resultado.

Podemos observar en el siguiente grafico la parte más representativa del código, en el cual podemos cómo se comporta la parte combinacional de la máquina de estados.

```
-------------------------------------------------
--Proceso combinacional de la máquina de estados--
-------------------------------------------------

PROCESS(start,actual,a,b,q,d)

  BEGIN

     CASE actual IS

         WHEN reposo =>
            resto<=(OTHERS=>'0');
            fin<='0';
            d<=a;
            enable<='1'; -- escrbio el registro
            IF start = '1' THEN
                siguiente<=resta;
            ELSIF a < b THEN
                siguiente<=final;
            ELSE
               siguiente<=reposo;

            END IF;

         WHEN resta =>
            resto<=(OTHERS=>'0');
            fin<='0';
            d<=q-b;
            enable<='1';
            IF d < b THEN
               siguiente<=final;

            ELSE

              siguiente<=resta;

            END IF;

         WHEN final=>

            resto<=q;
           d<=(OTHERS=>'0');
           enable<='1';
           fin<='1';
           siguiente<=reposo;

       END CASE;
  END PROCESS;
```

Ilustración 33: Proceso combinacional máquina de estado Naive

7.2.2. Algoritmo “No Restore Ring”

Una vez explicado cómo funciona el algoritmo, para implementarlo en VHDL, hemos optado por crear una maquina de estados, ya que en principio es la forma más sencilla de crear la secuencia de pasos necesaria para realizar el algoritmo, con lo cual pasamos a ver el siguiente grafico en el cual vemos el funcionamiento de la máquina de estados.

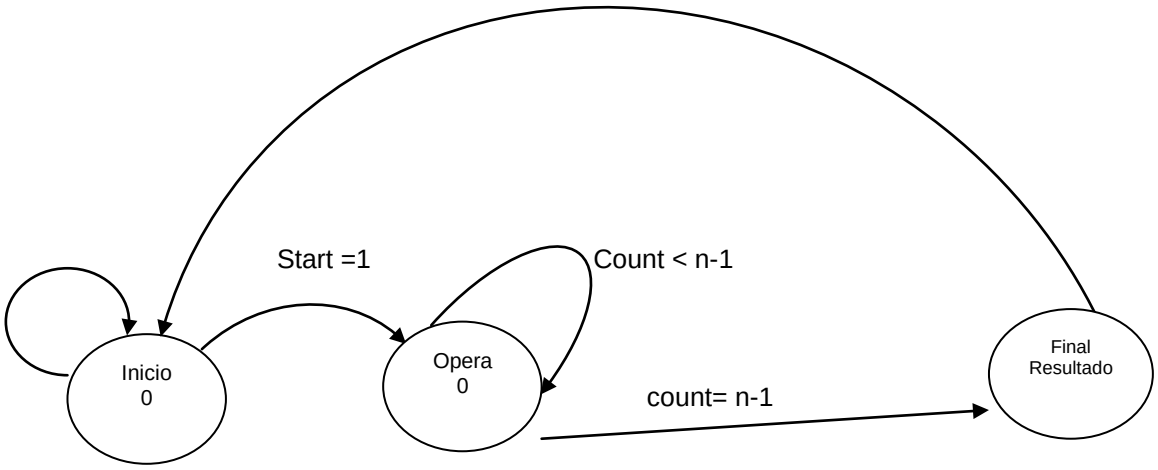

Ilustración 34: Máquina de estado algoritmo No Restore Ring Reducction

Ahora vemos parte del código en el cual se realiza la comparación de a(n+1) para poder pasar a la restauración o a incrementar el contador, esta parte quizás sea la parte más complicada de entender del código por la distinta notación que hemos empleado en el esquema da la máquina de estados, y por tenernos que ayudar de un variable para hacer más fácil su comprensión.

```
--------------------------------------
--registro para realizar operaciones.
--------------------------------------
PROCESS(clk,reset)
    VARIABLE r1,m1: STD_LOGIC_VECTOR(2*n -1 DOWNTO 0);
    BEGIN

    IF reset = '1' THEN
        b<=(OTHERS=>'0');
        r1:=(OTHERS=>'0');
        m1:=(OTHERS=>'0');

    ELSIF clk'EVENT AND clk = '1' THEN

        IF ff= "00" THEN
            b<=b(2*n-1 DOWNTO n) & p ;
            r1:=r1(2*n-1 DOWNTO n) & p ;
            m1:= m & m1(n-1 DOWNTO 0);

        ELSIF ff="01" THEN

           IF b > 0 THEN

           r1:=r1-m1;
           m1:= '0' & m1(2*n-1 DOWNTO 1);
           IF r1 < 0 THEN
              r1:=r1 + m1;
           END IF;
           b<=r1;

           ELSIF b = 0 THEN

            r1:=r1;
            m1:=m1;
            b<=b;

           ELSE

           r1:=r1+m1;
           m1:= '0' & m1(2*n-1 DOWNTO 1);
           IF r1 < 0 THEN – Proceso de restauracion
              r1:=r1 + m1;
           END IF;
           b<=r1;
           END IF;

        ELSE


        r1:=r1;
        m1:=m1;
        b<=b;

        END IF;
    END IF;
    res<=r1(n-1 DOWNTO 0);-- devuelve la operacion
    END PROCESS;
```

Ilustración 35: Código algoritmo No Restore Ring Reducction

Veamos un ejemplo de simulación de la implementación de este algoritmo matemático, en el cual podemos observar los distintos aspectos, tales como el contador o la máquina de estados.

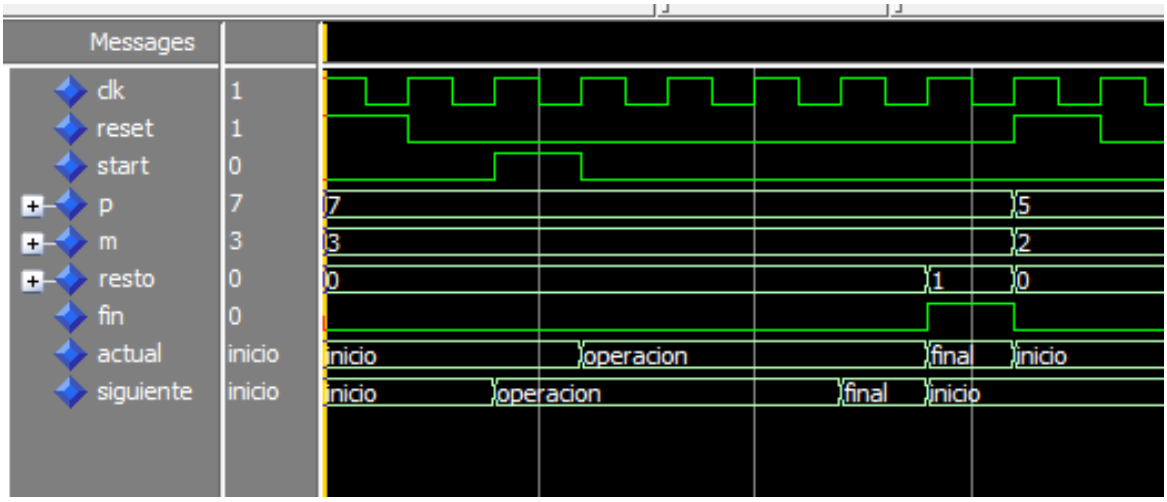

Ilustración 36: Simulación algoritmo No Restore Ring Reducction de 4 Bits

7.3. Implementación Multiplicación Modular

Hemos comentado anteriormente que la multiplicación modular la podremos hacer de dos formas diferentes con lo que hemos llamado algoritmos directos y con los algoritmos indirectos.

De los algoritmos indirectos para realizar la una operación de multiplicación modular, podemos realizar la multiplicación modular en dos partes, por un lado una multiplicación y seguidamente aplicar una reducción. Con lo que necesitamos por separado un algoritmo para multiplicación y una algoritmo para la reducción.

Los algoritmos elegidos para realizar la multiplicación han sido los algoritmos “Add & Shift”, que veremos más adelante es el algoritmo de multiplicación que mejores características tiene. Y el algoritmo que hemos llamado “AxB”, ya que este algoritmo es el que menos tiempo de ejecución tiene.

Como reductor hemos seleccionado al algoritmo “No Restore Ring”, debido a que de este algoritmo conocemos su comportamiento de antemano, sea cual sea el tipo de dato que se valla a utilizar.

De lo que hemos llamado los métodos directos, se implementara el algoritmo de Montgomery.

La disposición de los circuitos es la siguiente, en primer lugar el circuito multiplicador realiza la multiplicación cuando este acaba manda una señal al circuito reductor para que este empiece el proceso de reducción. Aquí podemos ver una representación de la entidad del circuito.

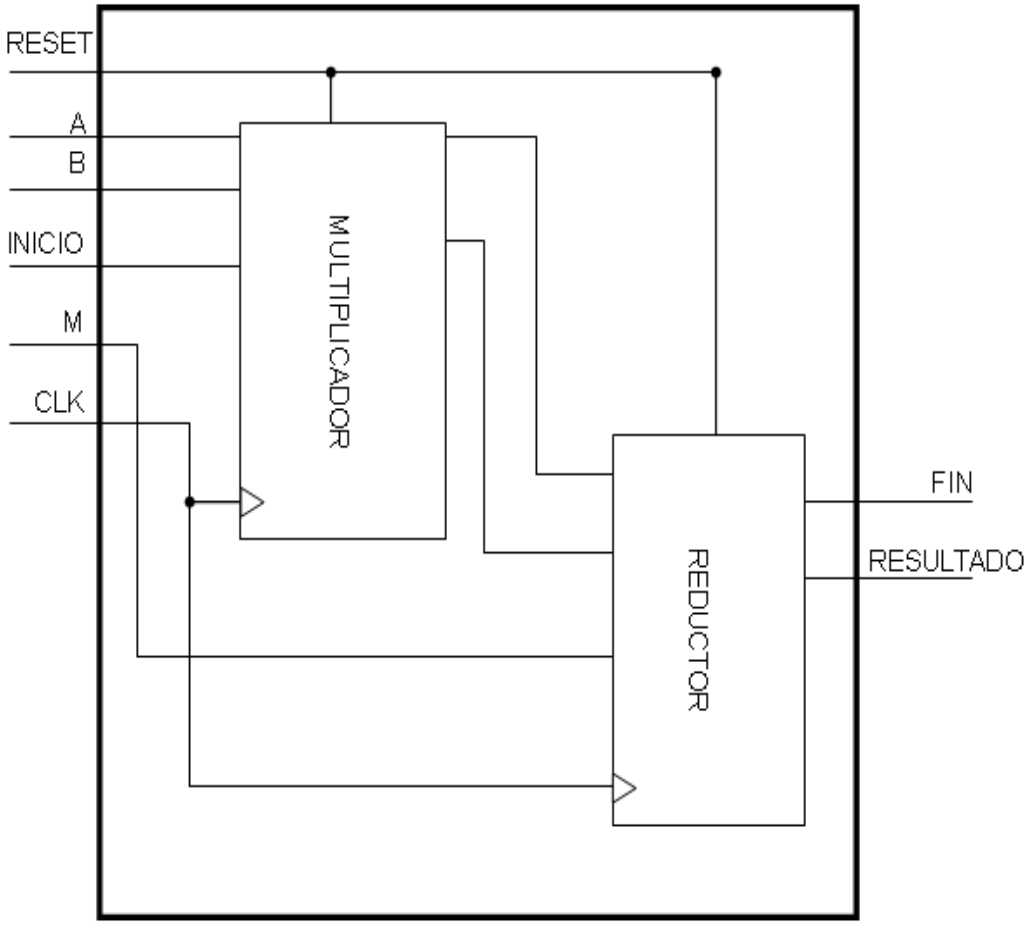

Ilustración 37: Diagrama de bloques multiplicador modular

7.3.1. Método indirecto Multiplicador "Add & Shfit" con reductor "No Restore Ring"

En primer lugar vemos la combinación de multiplicador por el método de suma y desplazamiento "Add and Shift" con el circuito reductor de Barret por el algoritmo "No Restore Ring".

En este siguiente grafico podemos observar la arquitectura superior del algoritmo, en esta parte solo se instancian los diferentes componentes, para poder realizar la multiplicación modular. Para este caso vemos como hemos utilizado los anteriores algoritmos implementados.

```
----------------------------------------------------------------
--Algoritmo multiplicacion y reduccion  de forma indirecta.
----------------------------------------------------------------

ENTITY aasnr IS
    GENERIC (n:INTEGER:=4);-- numero de bits
    PORT(
       clk,reset,start: IN STD_LOGIC;-- señales de reloj, reset e inicio
       a,b: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);--operandos
       m: IN STD_LOGIC_VECTOR(2*n-1 DOWNTO 0);
       fin:OUT STD_LOGIC;-- señal de fin de operacion
       resultado: OUT STD_LOGIC_VECTOR(2*n-1 DOWNTO 0)-- resultado de a*b mod m
        );
        END aasnr;

ARCHITECTURE algoritmo OF aasnr IS
    COMPONENT multiplicadorfsm1 IS
       PORT(
       clk,reset,start: IN STD_LOGIC;--señales de control,reloj,reset e inicio
       a,b: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);--multiplicando,multiplicador
       resultado: OUT STD_LOGIC_VECTOR(2*n-1 DOWNTO 0);--resultados
       finm: OUT STD_LOGIC--señal de fin de cuenta
       );
    END COMPONENT;

   COMPONENT resredc IS
        PORT(
        clk,reset,start: IN STD_LOGIC; -- señales de reloj, reset e inicio
        p,m: IN STD_LOGIC_VECTOR (2*n-1 DOWNTO 0);-- operandos
        resto: OUT STD_LOGIC_VECTOR (2*n-1 DOWNTO 0);-- resultado de a mod b
        finr: OUT STD_LOGIC --señal de fin de la operacion
        );
    END COMPONENT;

   SIGNAL fin_multiplicador: STD_LOGIC;
   SIGNAL resultado_multiplicador: STD_LOGIC_VECTOR (2*n-1 DOWNTO 0);

   BEGIN
multi: multiplicadorfsm1 PORT MAP
(clk,reset,start,a,b,resultado_multiplicador,fin_multiplicador);

rest: resredc PORT MAP
(clk,reset,fin_multiplicador,resultado_multiplicador,m,resultado,fin);

END algoritmo;
```

Ilustración 38: Código multiplicador modular "Add & Shift" - "No restore Ring"

Por último podemos ver una simulación del circuito, esta imagen corresponde a realizar una simulación cuando el número de bits que utilizamos en los operandos es de 4 bits y este es el resultado que obtenemos

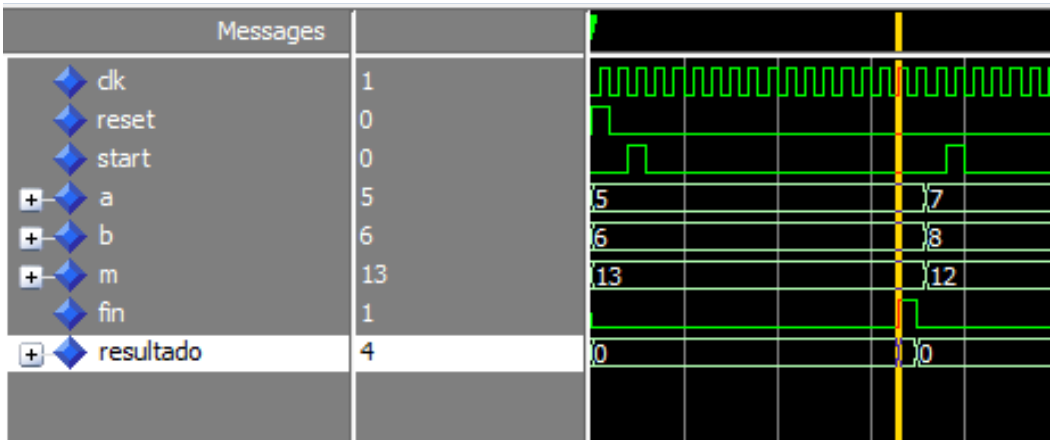

Ilustración 39: Simulación algoritmo Multiplicador "Add and Shift" Reductor "No Restore Ring" 4 bits

7.3.2. Método indirecto Multiplicador "AxB" con reductor "No Restore Ring"

En esta segunda prueba de multiplicador y reductor, podemos ver como hemos utilizado el multiplicador que aportan las bibliotecas de VHDL, al que hemos llamado "AxB" y el mismo reductor utilizado en el apartado anterior.

En este siguiente gráfico podemos observar la arquitectura superior del algoritmo, en esta parte solo se instancian los diferentes componentes, para poder realizar la multiplicación modular. Para este caso vemos como hemos utilizado los anteriores algoritmos implementados.

```
ARCHITECTURE algoritmo OF axbnr IS

    COMPONENT multiplicador_simple IS
       PORT(
       clk,reset,start: IN STD_LOGIC;--señales de control,reloj,reset e inicio
       a,b: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);--multiplicando,multiplicador
       resultado: OUT STD_LOGIC_VECTOR(2*n-1 DOWNTO 0);--resultados
       fin: OUT STD_LOGIC--señal de fin de cuenta
       );
    END COMPONENT;

   COMPONENT resredc IS
        PORT(
        clk,reset,start: IN STD_LOGIC; -- señales de reloj, reset e inicio
        a,b: IN STD_LOGIC_VECTOR (2*n-1 DOWNTO 0);-- operandos
        resto: OUT STD_LOGIC_VECTOR (2*n-1 DOWNTO 0);-- resultado de a mod b
        fin: OUT STD_LOGIC --señal de fin de la operacion
        );
    END COMPONENT;

   SIGNAL fin_multiplicacion: STD_LOGIC;
   SIGNAL resultado_multiplicacion: STD_LOGIC_VECTOR (2*n-1 DOWNTO 0);

  BEGIN

   multi:multiplicador_simple PORT MAP
  (clk,reset,start,a,b,resultado_multiplicacion,fin_multiplicacion);

  rest: resredc PORT MAP
 (clk,reset,fin_multiplicacion,resultado_multiplicacion,m,resultado,fin)

END algoritmo;
```

Ilustración 40: Código multiplicador modular modular “A x B” - “No restore Ring”

Por último podemos ver una simulación del circuito, esta imagen corresponde a realizar una simulación cuando el número de bits que utilizamos en los operandos es de 4 bits y este es el resultado que obtenemos

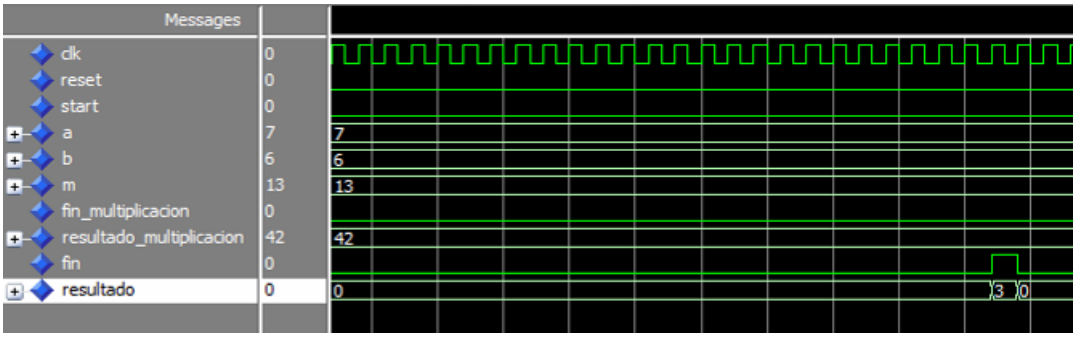

Ilustración 41: Simulación algoritmo Multiplicador “AxB” Reductor “No Restore Ring” 4 bits

7.3.3. Implementación "Montgomery"

En este punto pasamos a explicar una parte del código el cual realizar el algoritmo de Montgomery, en este punto decir que este algoritmo no calcula de forma correcta la multiplicación modular, con lo que es necesario aplicar dos veces este algoritmo para que la multiplicación modular de resultados validos, pero esto se verá un poco más adelante.

Una vez explicado cómo funciona el algoritmo, para implementarlo en VHDL, hemos optado por crear una maquina de estados, ya que en principio es la forma más sencilla de crear la secuencia de pasos necesaria para realizar el algoritmo, con lo cual pasamos a ver el siguiente grafico en el cual vemos el funcionamiento de la máquina de estados.

Dentro de las variables que controlan el proceso podemos hablar de la variable 'start', es una señal externa que sirve para iniciar el proceso, y de 'count', que es la variable que pertenece a un contador, el cual sirve para llevar el número de bits por el cual se hace la suma.

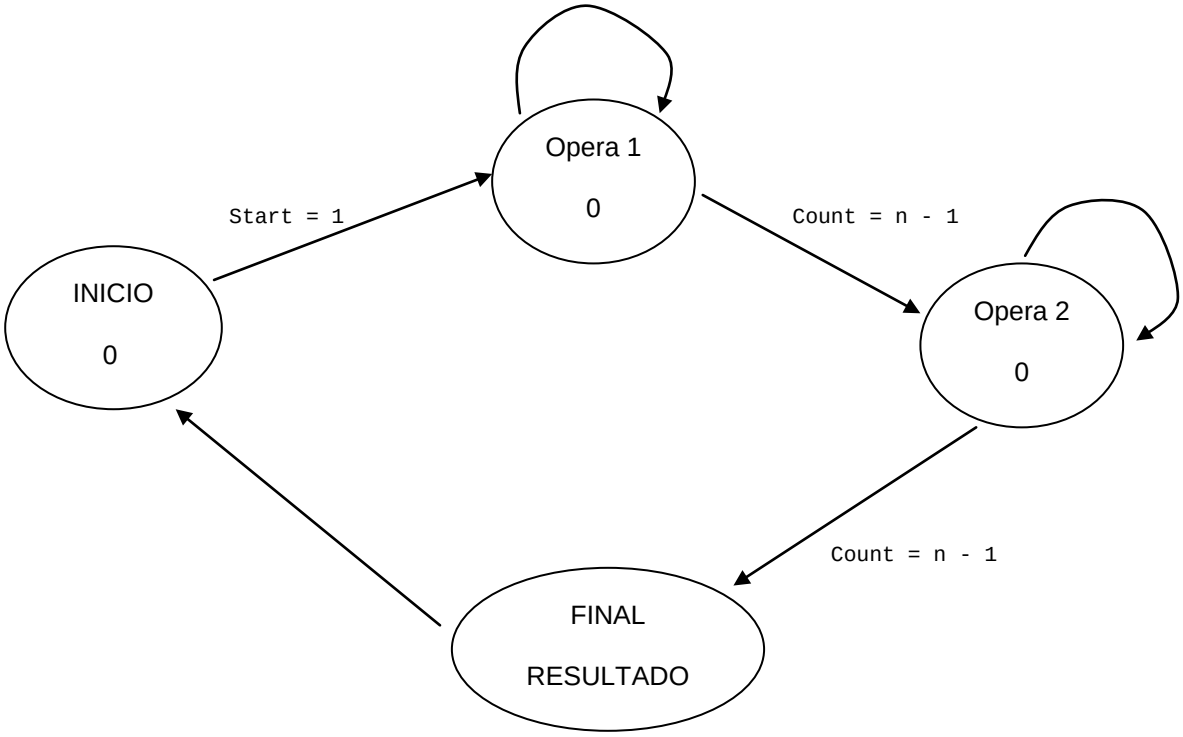

Ilustración 42: Maquina de estados Algoritmo de Montgomery

Podemos representar un flujograma del funcionamiento completo del algoritmo para poder ver de forma resumida el funcionamiento de la multiplicación de Montgomery.

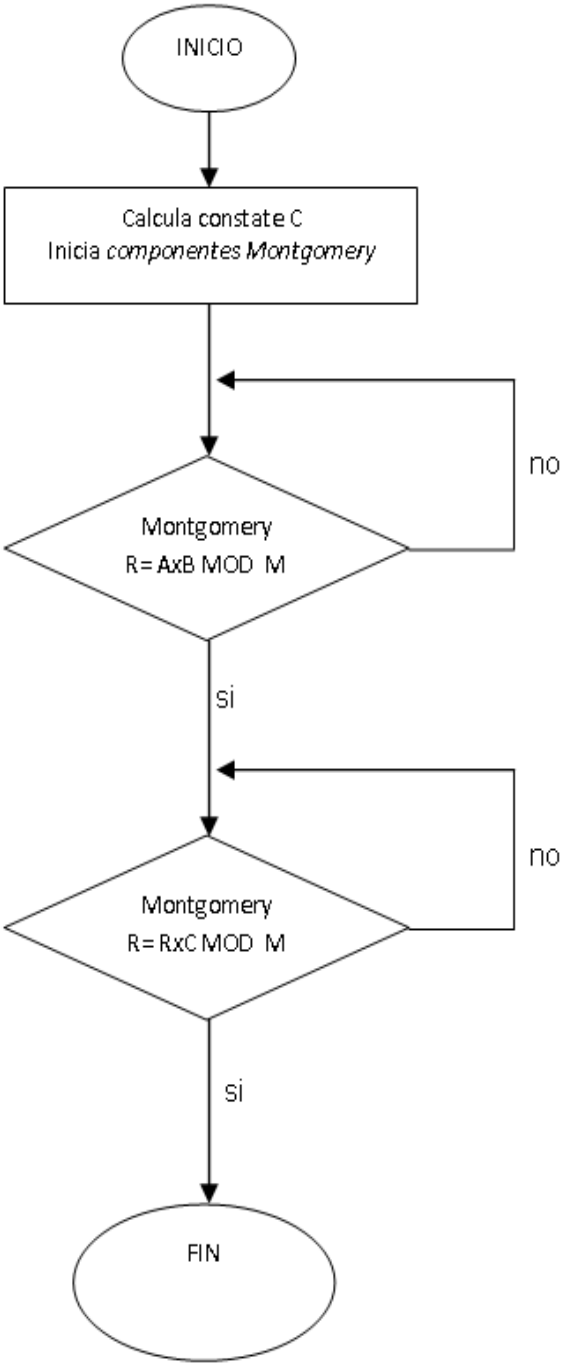

Ilustración 43: Diagrama de flujo algoritmo Montgomery

Por último Veamos un ejemplo de simulación de la implementación de este algoritmo matemático, en el cual podemos observar el funcionamiento del algoritmo.

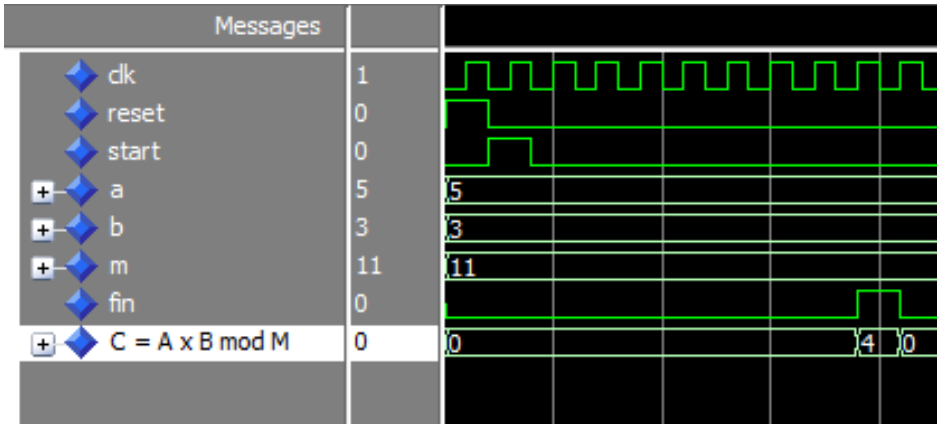

Ilustración 44: Simulación algoritmo Montgomery de 4 bits

7.4. Implementación Exponenciación Modular

Para entender de una forma más clara el funcionamiento de la Exponenciación modular mediante el algoritmo “Left to Right” pasamos a ver el siguiente flujograma, es necesario saber que ‘e’ es el exponente que en principio es de 3 bits, y ‘R’ es un registro auxiliar para ir almacenando los resultados parciales.

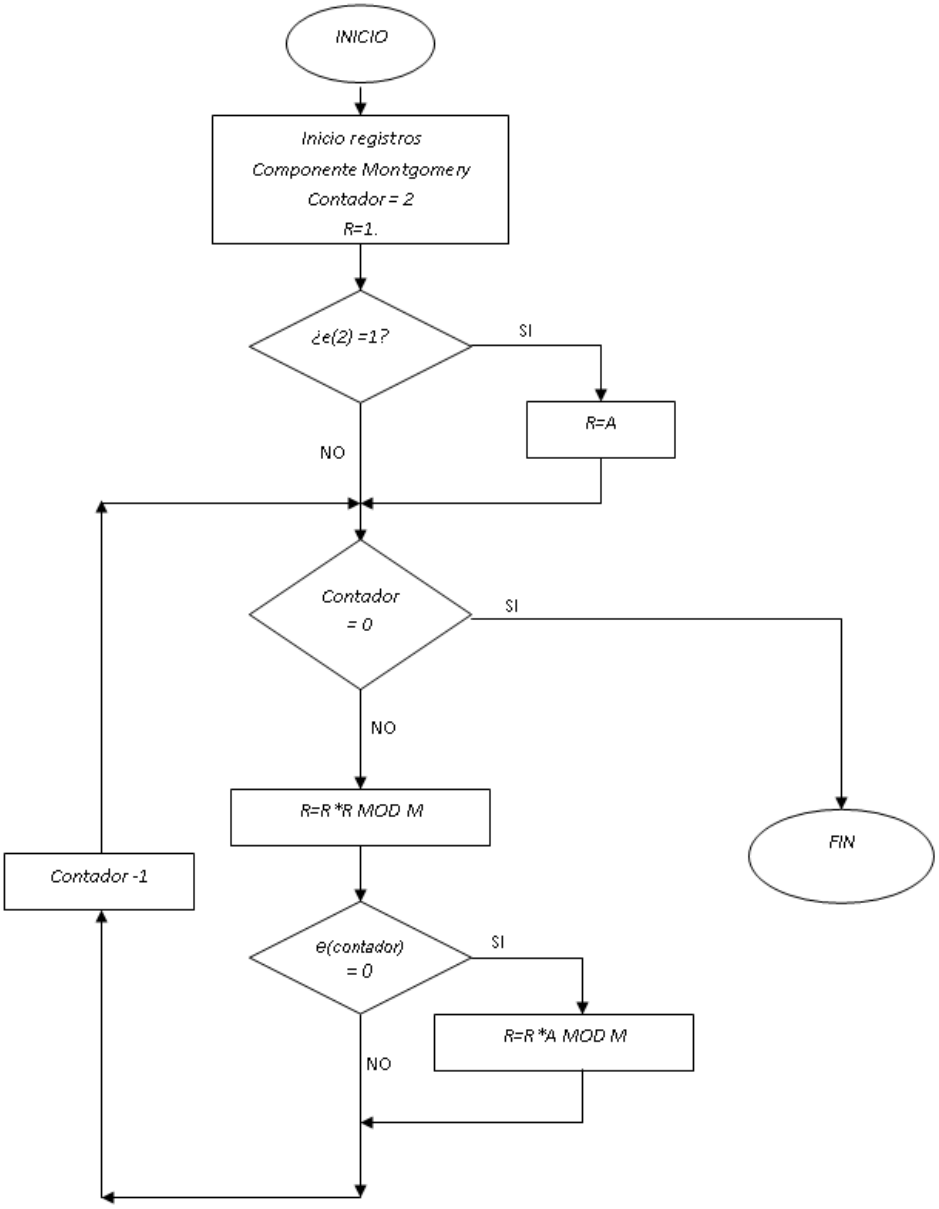

Ilustración 45: Flujograma algoritmo Exponenciador modular

Para programar el algoritmo en el lenguaje de programación de hardware VHDL, hemos elegido realizar una maquina de estado, ya que considéranos la forma más sencilla de implementar el algoritmo, en primer lugar se presenta la parte central del código, donde se contiene la parte combinacional de la máquina de estado. Seguidamente pasamos a ver un grafico explicativo donde se puede ver de una forma gráfica el comportamiento de la máquina de estado para así poder comprender de una mejor forma el código.

```
-----------------------------------------------
-- Proceso combinacional de la maquina de estados
-----------------------------------------------

PROCESS(actual,finmont,count,start,e)
   BEGIN

   CASE actual IS

       WHEN inicio =>
           enable<='0';
           inimont<='0';
           fin<='0';

           IF start = '0' THEN
               reg<="010";
               siguiente<=inicio;

            ELSE
               IF e(2)='1' THEN
                  reg<="001";
                  siguiente<=expomod1;
               ELSE
                  reg<="000";
                  siguiente<=expomod1;

               END IF;
           END IF;

        WHEN expomod1 =>
           enable<='0';
           inimont<='1';
           reg<="111";
           fin<='0';
           siguiente<=espera1;

        WHEN espera1 =>
           inimont<='0';
           fin<='0';


           IF finmont = '0' THEN
               enable<='0';
               reg<="111";
               siguiente<=espera1;

           ELSE

              IF e(count)='1' THEN
                 enable<='0';
                 reg<="100";
                 siguiente<=expomod2;
```

```
ELSE

               IF e(count)='1' THEN
                  enable<='0';
                  reg<="100";
                  siguiente<=expomod2;

               ELSE


                 IF count = 0 THEN
                     reg<="010";
                     enable<='0';
                     siguiente<=final;
                 ELSE
                     reg<="110";
                     enable<='1';
                     siguiente<=expomod1;
                 END IF;


               END IF;


            END IF;

            WHEN expomod2 =>
            enable<='0';
            inimont<='1';
            reg<="111";
            fin<='0';
            siguiente<=espera2;

            WHEN espera2 =>
            inimont<='0';
            fin<='0';

            IF finmont = '0' THEN
                enable<='0';
                                    reg<="111";
                siguiente<=espera2;

            ELSE

                IF count = 0 THEN
                    reg<="010";
                    enable<='0';
                    siguiente<=final;
                ELSE
                    enable<='1';
                    reg<="110";
                    siguiente<=expomod1;
                END IF;
             END IF;

            WHEN final =>
            enable<='0';
            inimont<='0';
            reg<="010";
            fin<='1';
            siguiente<=inicio;


   END CASE;
   END PROCESS;
```

Ilustración 46: Código Algoritmo Exponenciación modular

Podemos ver en el anterior imagen cuales son los distintos estados por el cual pasa la máquina de estados para realiza el algoritmo, pasamos a dar una breve explicación de lo que realiza cada estado.

El primer estado, “inicio” prepara los distintos registros, contador y elementos necesarios para realizar la exponenciación modular, si tenemos la entrada de “Start” activada, pasamos a los siguientes estados. El segundo estado,”expomod1”, básicamente realiza la primera multiplicación modular por el algoritmo de Montgomery, una vez que esta ha empezado, pasamos el tercer estado,”espera1”, este es un estado de espera, en el cual simplemente esperamos a que la multiplicación modular acabe y pasamos el siguiente estado, en el cual pasamos a realizar la segunda multiplicación modular, este estado es el estado “expomod2”, el siguiente estado, es el estado de “espera2”, este estado espera a que termine la multiplicación modular anterior. Por último tenemos el estado final que es el último estado en el cual arroja el resultado de la exponenciación modular.

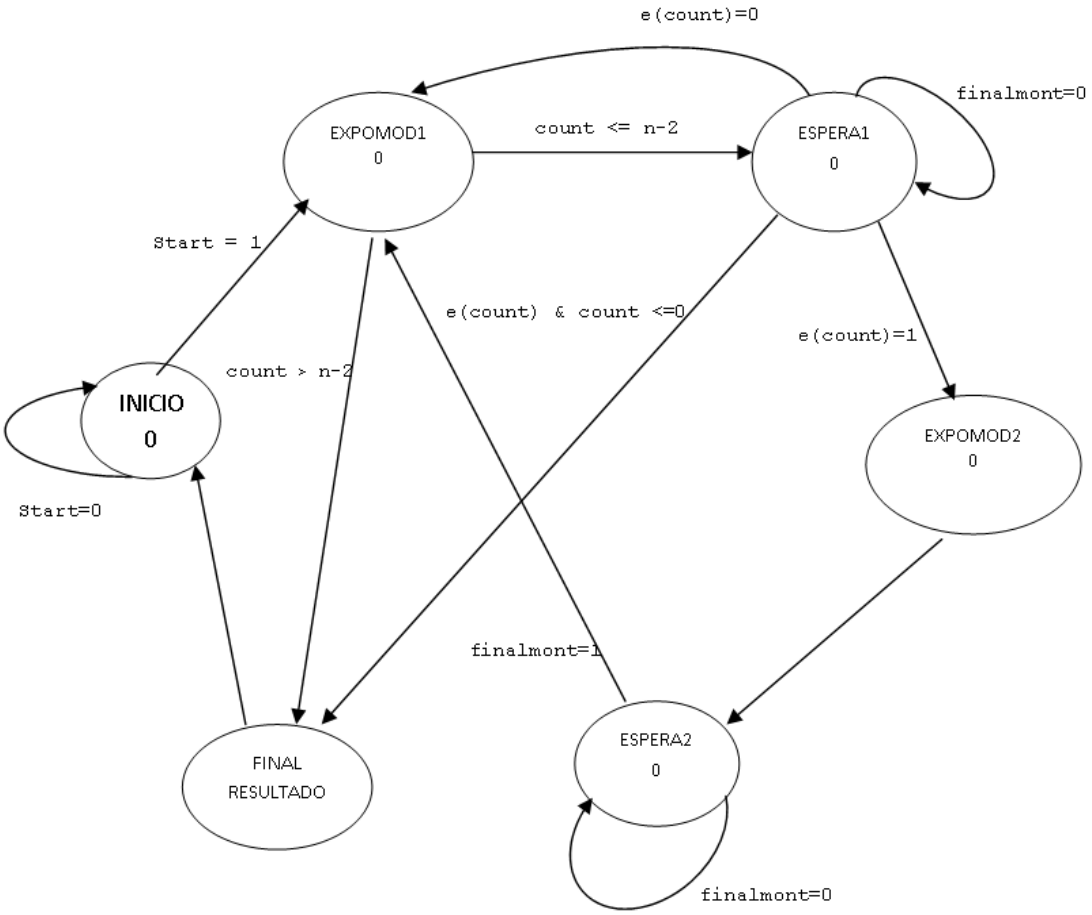

Ilustración 47: Maquina de estado Exponeneciación modular

Por último vemos una simulación del funcionamiento del circuito, para una ancho de palabra de 4 bits, en este grafico nos hacemos una idea de cómo realmente trabaja el algoritmo implementado.

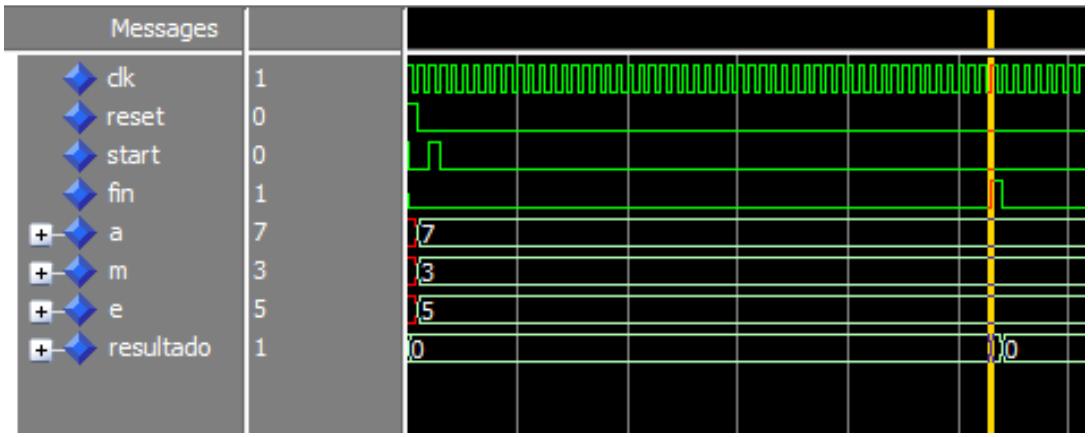

Ilustración 48: Simulación algoritmo Exponenciación modular 4 bits

CAPÍTULO IV - RESULTADOS

8. Resultado

En este punto del proyecto pasamos a ver los resultados experimentales de los distintos algoritmos. En primer lugar veremos los datos que lanza el sintetizador cuando sintetizamos los algoritmos con un ancho de palabra de 256 Bits y 512 Bits, que son los más adecuados para nuestro propósito. Los demás resultados se pueden ver en el anexo 1, el cual contiene todos los resultados de las diferentes síntesis.

Por otro lado veremos diferentes tablas resúmenes donde podemos ver distintos aspectos tales como área ocupada de los algoritmos, ciclos de reloj necesarios para que se den resultados, frecuencias máximas de funcionamiento.

También veremos graficas, en donde se podrán apreciar de una forma más sencilla todos estos datos.

8.1. Resultado Multiplicadores

8.1.1. Resultados multiplicadores "Add & Shitf"

Una vez realizado el código pertinente, que se puede ver en el anexo de códigos, pasamos a analizar los datos más relevantes que arroja el sintetizador los demás resultados se pueden observar en el anexo de síntesis. Hay que tener en cuenta que este algoritmo de multiplicación realiza una multiplicación en "2n" ciclos de reloj, siendo 'n' el número de bits los operandos de entrada.

N=256

Minimum period: 38.311ns (Maximum Frequency: 26.102MHz)

Minimum input arrival time before clock: 12.759ns

Maximum output required time after clock: 11.701ns

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1218	13312	9%
Number of Slice Flip Flops	1039	26624	3%
Number of 4 input LUTs	2203	26624	8%
Number of bonded IOBs	1028	221	465%
Number of GCLKs	1	8	12%

Ilustración 49: Resultados síntesis Add & Shift de 256 bits

N=512

Minimum period: 71.079ns (Maximum Frequency: 14.069MHz)
Minimum input arrival time before clock: 13.624ns
Maximum output required time after clock: 12.085ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	2424	13312		18%
Number of Slice Flip Flops	2068	26624		7%
Number of 4 input LUTs	4386	26624		16%
Number of bonded IOBs	2052	221	928%	
Number of GCLKs	1	8		12%

Ilustración 50: Resultados síntesis Add & Shift de 512 bits

En la siguiente tabla podemos ver un resumen de los datos más relevantes que nos da el sintetizador cuando realizamos la síntesis del algoritmo para los distintos anchos de palabra. Podemos ver un resumen tanto del área Flip Flor y Luts que ocupan el algoritmo así como la frecuencia máxima de funcionamiento como el factor área x tiempo

Add & Shift	4 Bits	8 Bits	16 Bits	32 Bits	48 Bits	64 Bits
LUTS	24	48	88	231	302	400
FLIP FLOP	21	38	73	140	206	272
LUTS x ns	685	2994	12421	91195	194841	396979
Frec. Max. (Mhz)	140,233	128,238	113,353	81,057	74,399	64,487

Add & Shift	96 Bits	128 Bits	256 Bits	512 Bits	1024 Bits
LUTS	463	1109	2203	4386	8749
FLIP FLOP	393	524	1039	2068	4136
LUTS x ns	792552	3112582	21606178	159617277	1223948424
Frec. Max. (Mhz)	56,082	45,606	26,102	14,069	7,320

Ilustración 51: Resumen Multiplicador “Add and Shift”

Podemos ver un resumen de la frecuencia máxima de funcionamiento contra el ancho de palabra del algoritmo.

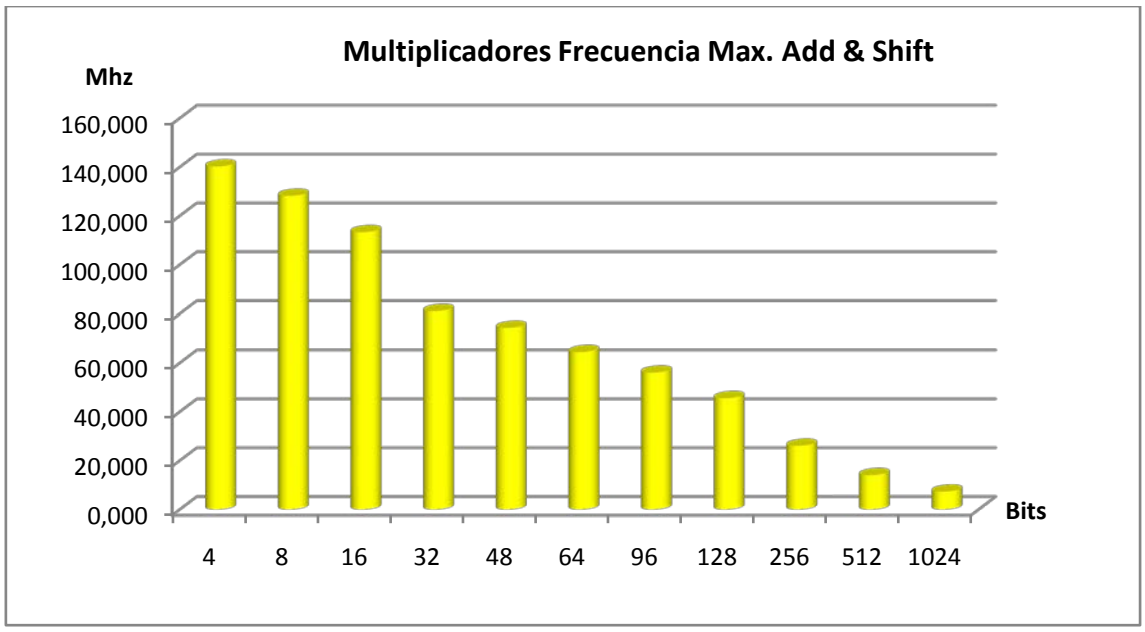

Ilustración 52: Frecuencia máxima "Add and Shif"

8.1.2. Resultados multiplicador de Booth

Una vez realizado el código pertinente, que se puede ver en el anexo de códigos, pasamos a analizar los datos que nos ofrece el sintetizador. Este multiplicador realiza una multiplicación en 2*n – 1 ciclos de reloj, siendo 'n' el número de bits los operandos de entrada.

N=256 BITS

Minimum period: 51.620ns (Maximum Frequency: 19.372MHz)
Minimum input arrival time before clock: 49.715ns
Maximum output required time after clock: 11.724ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	1291	13312		9%
Number of Slice Flip Flops	1039	26624		3%
Number of 4 input LUTs	2345	26624		8%
Number of bonded IOBs	1028	221	465%	
Number of GCLKs	1	8		12%

Ilustración 53: Resultados síntesis Booth de 256 bits

N= 512 BITS

Minimum period: 85.109ns (Maximum Frequency: 11.750MHz)

Minimum input arrival time before clock: 83.210ns

Maximum output required time after clock: 12.119ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	2565	13312		19%
Number of Slice Flip Flops	2069	26624		7%
Number of 4 input LUTs	4664	26624		17%
Number of bonded IOBs	2052	221	928%	
Number of GCLKs	1	8		12%

Ilustración 54: Resultados síntesis Booth de 96 bits

En la siguiente tabla podemos ver un resumen de los datos más relevantes que nos da el sintetizador cuando realizamos la síntesis del algoritmo para los distintos anchos de palabra. Podemos ver un resumen tanto del área Flip Flop y Luts que ocupan el algoritmo así como la frecuencia máxima de funcionamiento como el factor área x tiempo

Booth	4 Bits	8 Bits	16 Bits	32 Bits	48 Bits	64 Bits
LUTS	48	88	161	308	453	599
FLIP FLOP	21	38	71	136	200	265
LUTS x ns	1863	7508	33843	166182	760692	850292
Frec. Max. (Mhz)	103,082	93,765	76,115	59,308	48,300	45,086

Booth	96 Bits	128 Bits	256 Bits	512Bits	1024 Bits
LUTS	887	1177	2345	4664	9299
FLIP FLOP	393	524	1039	2069	4128
LUTS x ns	2390302	4932779	30988518	203237569	1448580068
Frec. Max. (Mhz)	35,624	30,542	19,372	11,750	6,573

Ilustración 55: Resumen Multiplicador Booth

Podemos ver un resumen de la frecuencia máxima de funcionamiento contra el ancho de palabra del algoritmo.

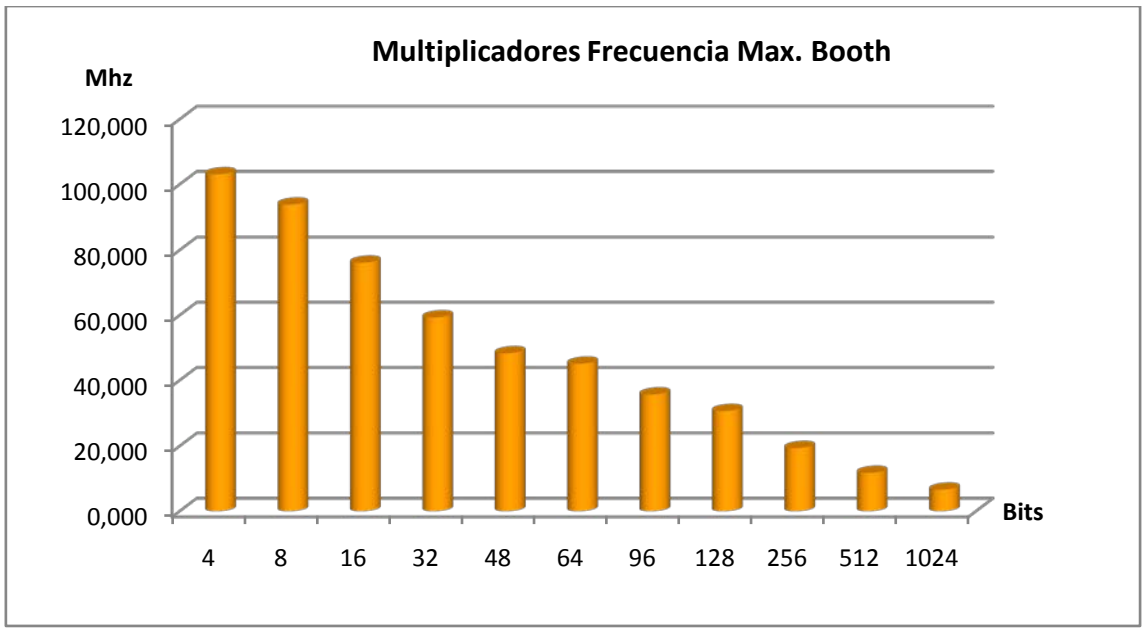

Ilustración 56: Frecuencia máxima Booth

8.1.3. Resultados multiplicador "A x B"

Una vez implementado el código necesario para el correcto funcionamiento del multiplicador pasamos a presentar los resultados obtenidos en el sintetizador, el código se puede ver en el anexo de códigos. Los demás resultados obtenidos en la sintetización se pueden ve en el anexo de síntesis. Este multiplicador realizar la multiplicación en un solo ciclo de reloj independientemente del ancho de palabra de los datos utilizados

Como se puede ver, en el anexo de síntesis, la síntesis de este circuito solo ha sido posible realizarla hasta un ancho de palabra de 256 bits, ya que si intentábamos probar un mayor ancho de palabra, ya sea 512 o 1024 bits el sintetizador después de un largo periodo de tiempo, en torno a 45 minutos, daba un error de memoria con lo que estas pruebas no se han podido realizar.

N=128 BITS

Maximum Frequency:
Minimum input arrival time before clock: 27.026ns
Maximum output required time after clock: 7.165ns

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1286	13312	9%
Number of 4 input LUTs	2548	26624	9%
Number of bonded IOBs	196	221	88%
Number of GCLKs	1	8	12%

Ilustración 57: Resultados síntesis AxB de 128 bits

N=256 BITS

Maximum Frequency:
Minimum input arrival time before clock: 35.452ns
Maximum output required time after clock: 7.165ns

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	4909	13312	36%
Number of 4 input LUTs	9772	26624	36%
Number of bonded IOBs	388	221	175%
Number of GCLKs	1	8	12%

Ilustración 58: Resultados síntesis AxB de 256 bits

En la siguiente tabla podemos ver un resumen de los datos más relevantes que nos da el sintetizador cuando realizamos la síntesis del algoritmo para los distintos anchos de palabra. Podemos ver un resumen tanto del área Flip Flor y Luts que ocupan el algoritmo así como la frecuencia máxima de funcionamiento como el factor área x tiempo.

A x B	4 Bits	8 Bits	16 Bits	32 Bits	48 Bits	64 Bits
LUTS	23	86	316	1173	2548	445
LUTS x ns	209	1085	5336	26289	94280	132114
Frec. Max. (Mhz)	110,266	120,757	59,217	44,619	37,001	33,645

A x B	96 Bits	128 Bits	256 Bits
LUTS	9772	17172	67274
LUTS x ns	346437	708654	4137351
Frec. Max. (Mhz)	28,207	24,232	16,260

Ilustración 59: Resumen multiplicador "A x B"

Podemos ver un resumen de la frecuencia máxima de funcionamiento contra el ancho de palabra del algoritmo.

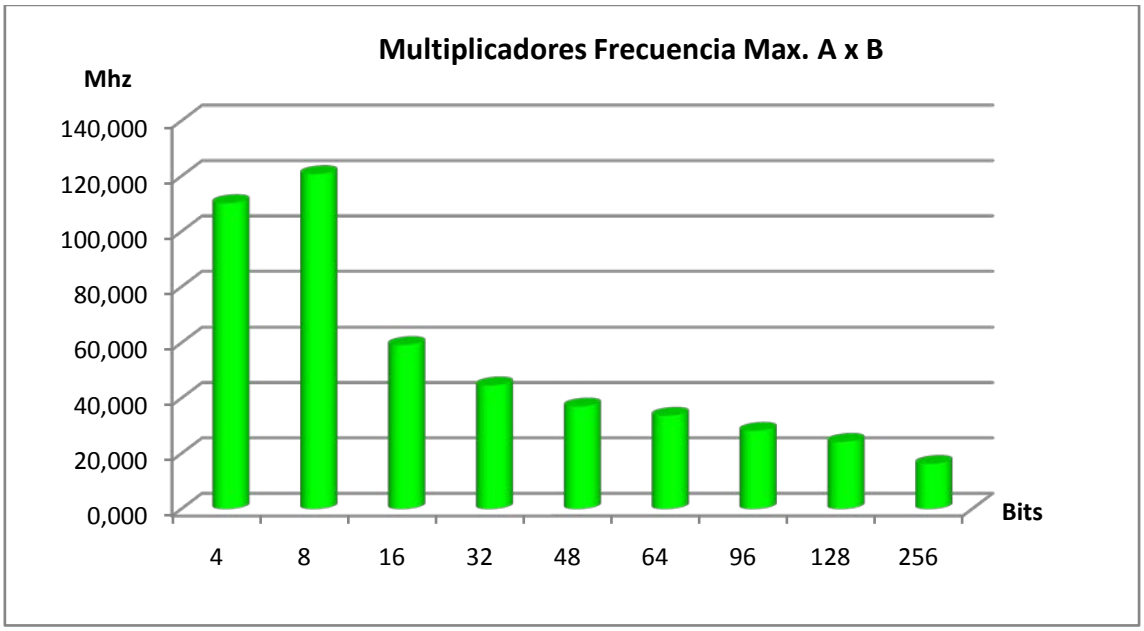

Ilustración 60: Frecuencia máxima "A x B"

8.1.4. Resumen de multiplicadores

En este punto presentamos un resumen de las propiedades más importante de los multiplicadores que hemos realizado. Veremos graficas de comparación de áreas y del factor área x tiempo.

En primer lugar comparamos los datos para los multiplicadores "Add & Shift" y el algoritmo de Booth, ya que estos son comparable en todo el ancho de palabra.

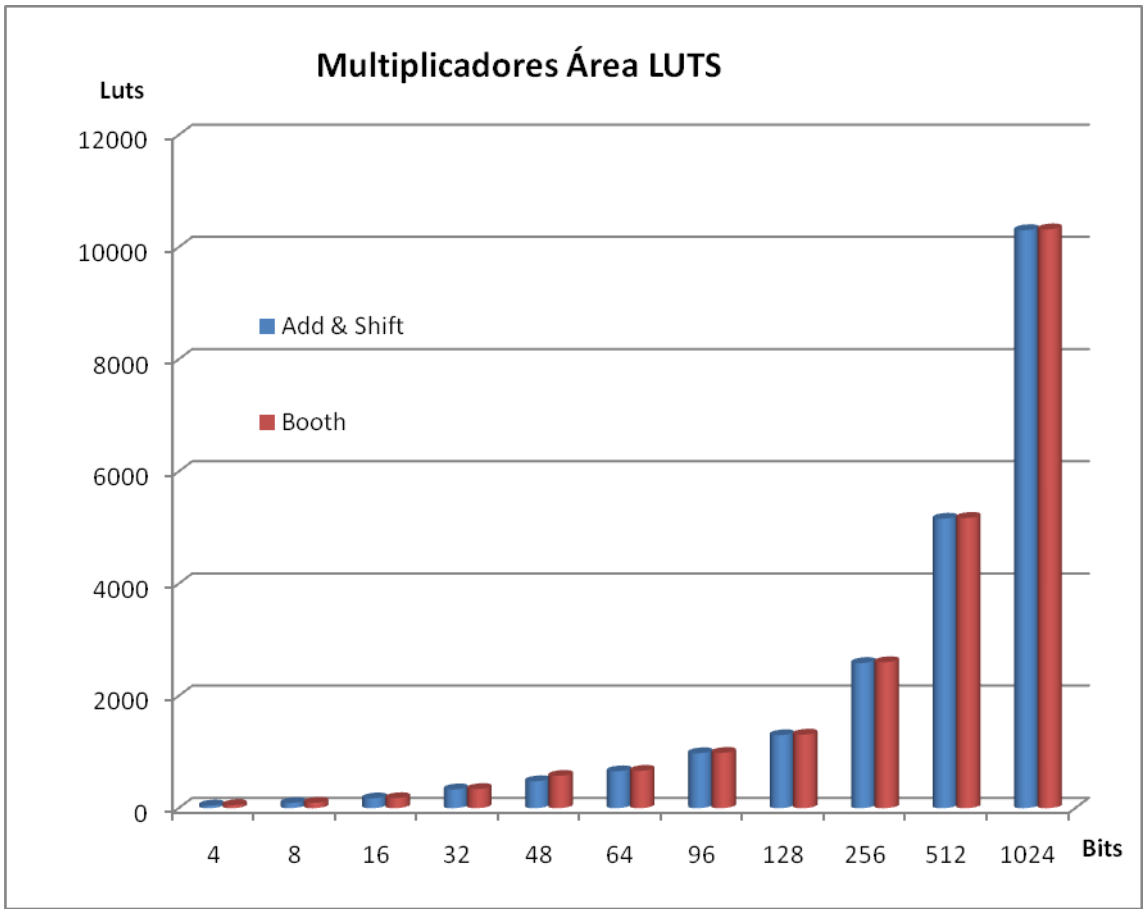

Ilustración 61: Comparación Área multiplicadores I

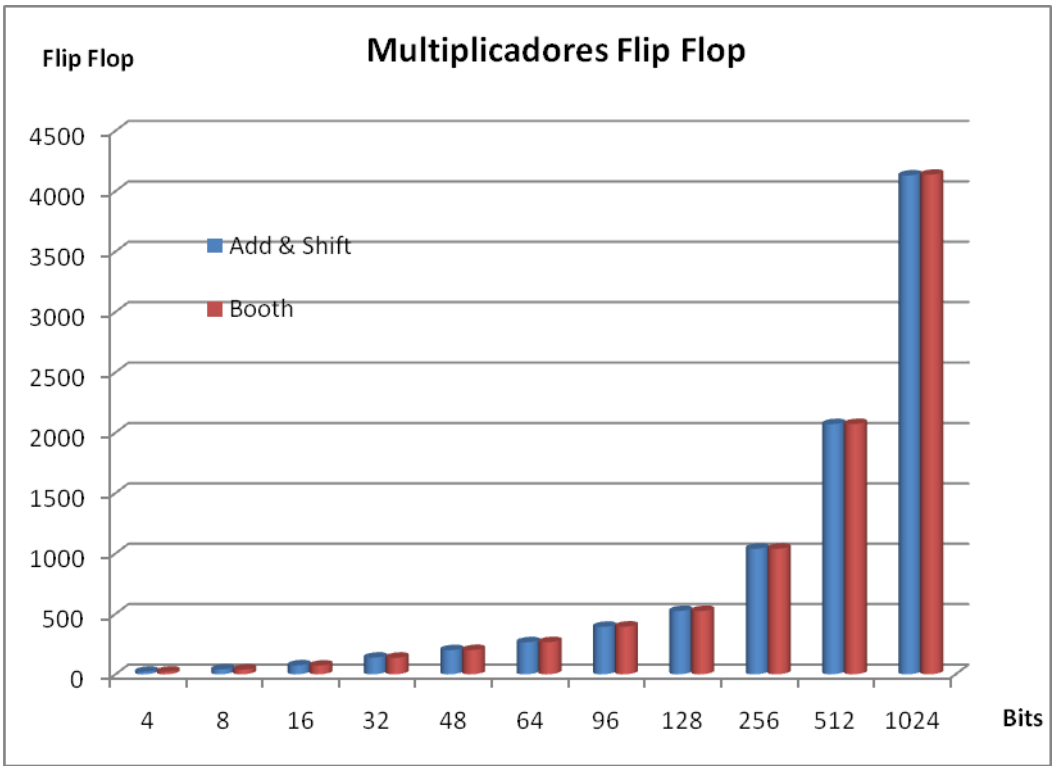

Ilustración 62: Comparación Flip Flop multiplicadores I

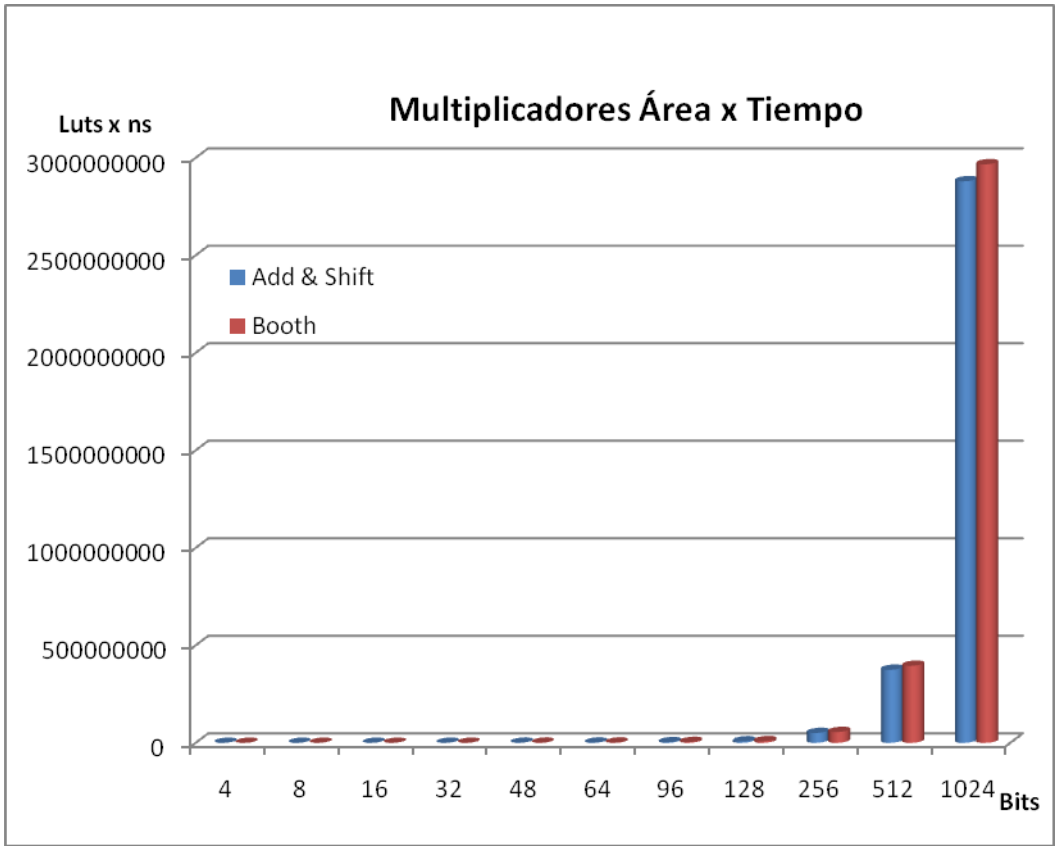

Ilustración 63: Comparación factor Área x Tiempo Multiplicadores I

Por último compararemos los tres algoritmos de multiplicación, tanto en su factor área x tiempo como el numero luts que esta algoritmo ocupa, el sintetizador en esta caso para el algoritmos “A x B”, no ha arrojado valores de Flip Flop.

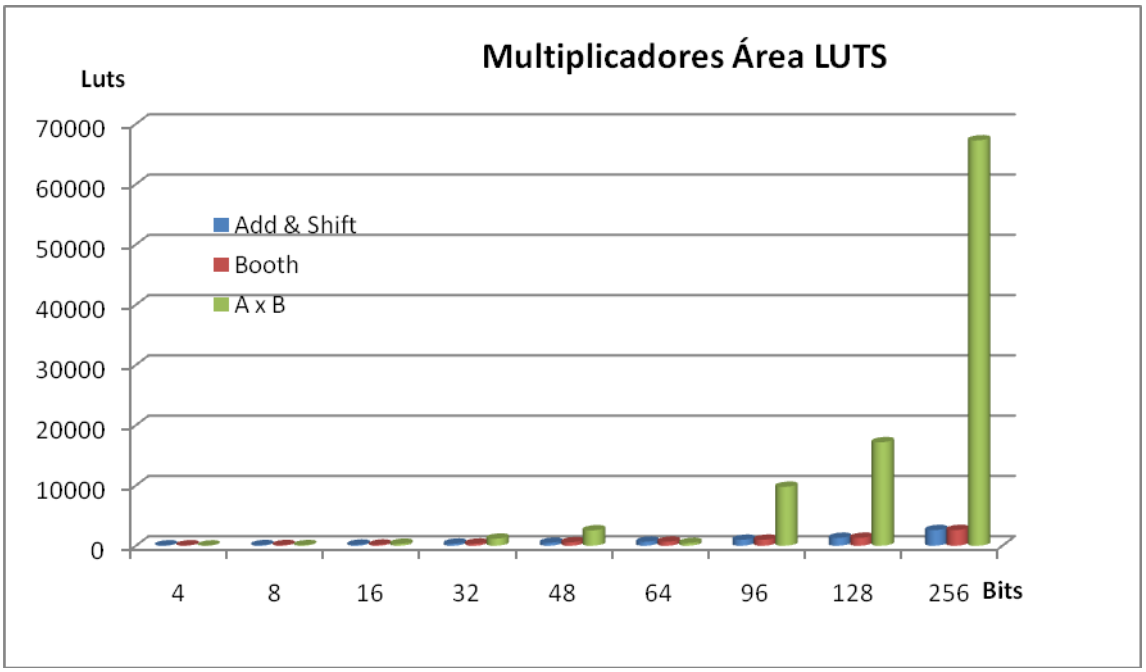

Ilustración 64: Comparación Área multiplicadores II

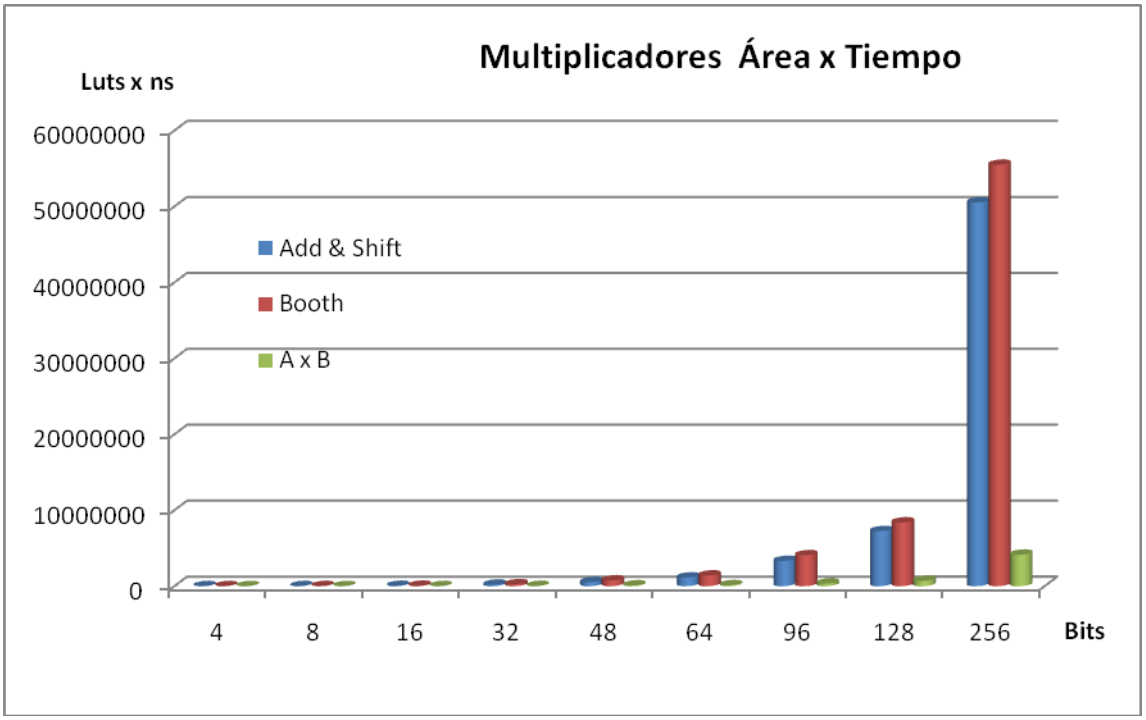

Ilustración 65 Comparación factor Area x Tiempo multiplicadores II

8.2. Resultado Reductores de "Barret"

8.2.1. Resultados Algoritmo de Naive

Una vez realizado el código pertinente, que se puede ver en el anexo de códigos, pasamos a analizar los datos más relevantes que arroja el sintetizador los demás resultados se pueden observar en el anexo de síntesis. Hay que tener en cuenta que este algoritmo de reducción realiza la reducción en función del valor de los operandos de entrada, con lo que para tomar una referencia, se ha decidido para poder realizar una comparación, tomar como tiempo de ejecución "4n" ciclos de reloj, con el fin de ser los mismos ciclos que tarda el otro algoritmo de reducción. Siendo 'n' el número de bits los operandos de entrada.

N=256

Minimum period: 25.472ns (Maximum Frequency: 39.259MHz)
Minimum input arrival time before clock: 25.747ns
Maximum output required time after clock: 11.191ns

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	681	13312	5%
Number of Slice Flip Flops	259	26624	0%
Number of 4 input LUTs	1284	26624	4%
Number of bonded IOBs	772	221	349%
Number of GCLKs	1	8	12%

Ilustración 66: Resultados síntesis Naive de 256 bits

N=512

Minimum period: 41.997ns (Maximum Frequency: 23.811MHz)
Minimum input arrival time before clock: 42.272ns
Maximum output required time after clock: 11.684ns

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1360	13312	10%
Number of Slice Flip Flops	517	26624	1%
Number of 4 input LUTs	2566	26624	9%
Number of bonded IOBs	1540	221	696%
Number of GCLKs	1	8	12%

Ilustración 67: Resultados síntesis Naive de 512 bits

En la siguiente tabla podemos ver un resumen de los datos más relevantes que nos da el sintetizador cuando realizamos la síntesis del algoritmo para los distintos anchos de palabra. Podemos ver un resumen tanto del área Flip Flor y Luts que ocupan el algoritmo así como la frecuencia máxima de funcionamiento como el factor área x tiempo

Naive	4 Bits	8 Bits	16 Bits	32 Bits	48 Bits	64 Bits
LUTS	24	44	84	164	243	324
FLIP FLOP	6	10	18	34	50	66
LUTS x ns	3273	13308	53566	230660	3689096	1081258
Frec. Max. (Mhz)	117,330	105,798	100,361	91,008	79,070	76,711

Naive	96 Bits	128 Bits	256 Bits	512Bits	1024 Bits
LUTS	484	664	1285	2567	5130
FLIP FLOP	98	130	259	517	1032
LUTS x ns	2803452	5824332	33644713	220713699	1574276014
Frec. Max. (Mhz)	66,295	58,370	39,110	23,819	13,347

Ilustración 68: Resumen algoritmo de Naive

Podemos ver un resumen de la frecuencia máxima de funcionamiento contra el ancho de palabra del algoritmo.

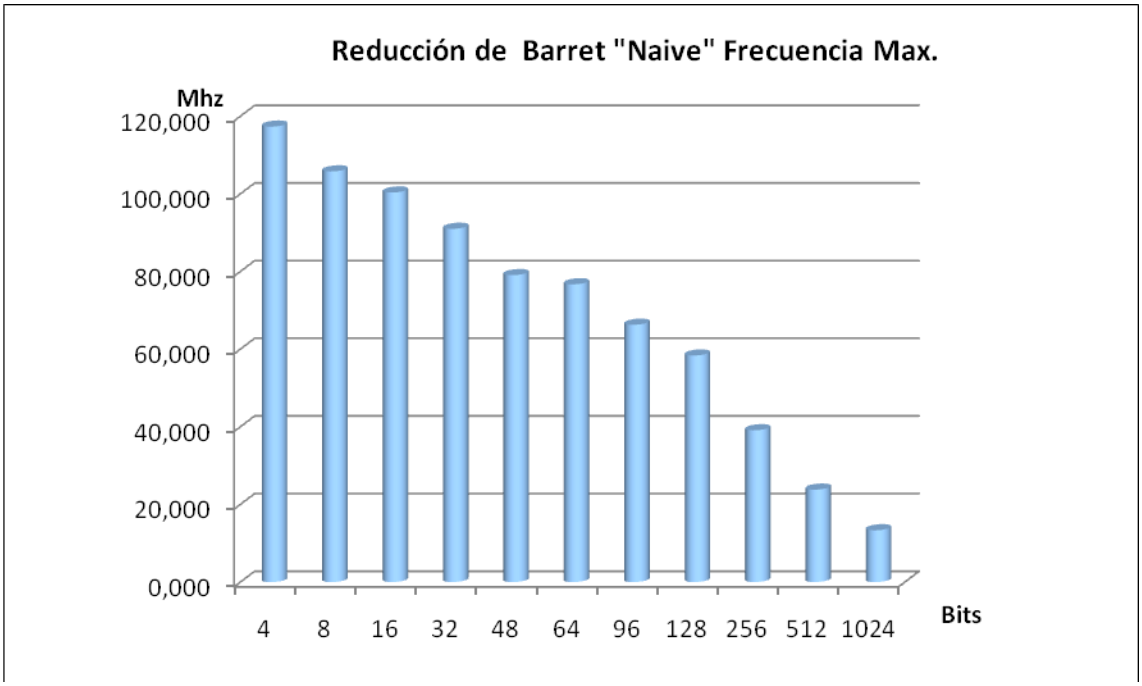

Ilustración 69: Frecuencia máxima Algoritmo de Naive

8.2.2. Resultados algoritmo "No Restore Ring"

Una vez realizado el código pertinente, que se puede ver en el anexo de códigos, pasamos a analizar los datos más relevantes que arroja el sintetizador los demás resultados se pueden observar en el anexo de síntesis. Hay que tener en cuenta que este algoritmo de reducción realiza una reducción en "4n" ciclos de reloj, siendo 'n' el número de bits los operandos de entrada.

N=256

Minimum period: 83.584ns (Maximum Frequency: 11.964MHz)
Minimum input arrival time before clock: 15.900ns
Maximum output required time after clock: 11.289ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	3739	13312		28%
Number of Slice Flip Flops	1549	26624		5%
Number of 4 input LUTs	6938	26624		26%
Number of bonded IOBs	772	221	349%	
Number of GCLKs	1	8		12%

Ilustración 70: Resultados síntesis "No Restore Ring" de 256 bits

N=512

Minimum period: 149.905ns (Maximum Frequency: 6.671MHz)
Minimum input arrival time before clock: 16.776ns
Maximum output required time after clock: 11.266ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	7469	13312		56%
Number of Slice Flip Flops	3089	26624		11%
Number of 4 input LUTs	13860	26624		52%
Number of bonded IOBs	1540	221	696%	
Number of GCLKs	1	8		12%

Ilustración 71: Resultados síntesis "No Restore Ring" de 512 bits

En la siguiente tabla podemos ver un resumen de los datos más relevantes que nos da el sintetizador cuando realizamos la síntesis del algoritmo para los distintos anchos de palabra. Podemos ver un resumen tanto del área Flip Flor y Luts que ocupan el algoritmo así como la frecuencia máxima de funcionamiento como el factor área x tiempo.

NRR	4 Bits	8 Bits	16 Bits	32 Bits	48 Bits	64 Bits
LUTS	119	227	446	880	1313	1747
FLIP FLOP	29	54	103	200	296	393
LUTS x ns	6007	25707	116438	586685	694172	3327294
Frec. Max. (Mhz)	103,082	93,765	76,115	59,308	48,300	45,086

NRR	96 Bits	128 Bits	256 Bits	512Bits	1024 Bits
LUTS	2611	3475	6938	13860	27698
FLIP FLOP	585	779	1549	3089	6165
LUTS x ns	9696377	22167053	148455883	1063773850	7984143051
Frec. Max. (Mhz)	35,624	30,542	19,372	11,750	6,573

Ilustración 72: Resumen Algoritmo “No Restore Ring”

Podemos ver un resumen de la frecuencia máxima de funcionamiento contra el ancho de palabra del algoritmo.

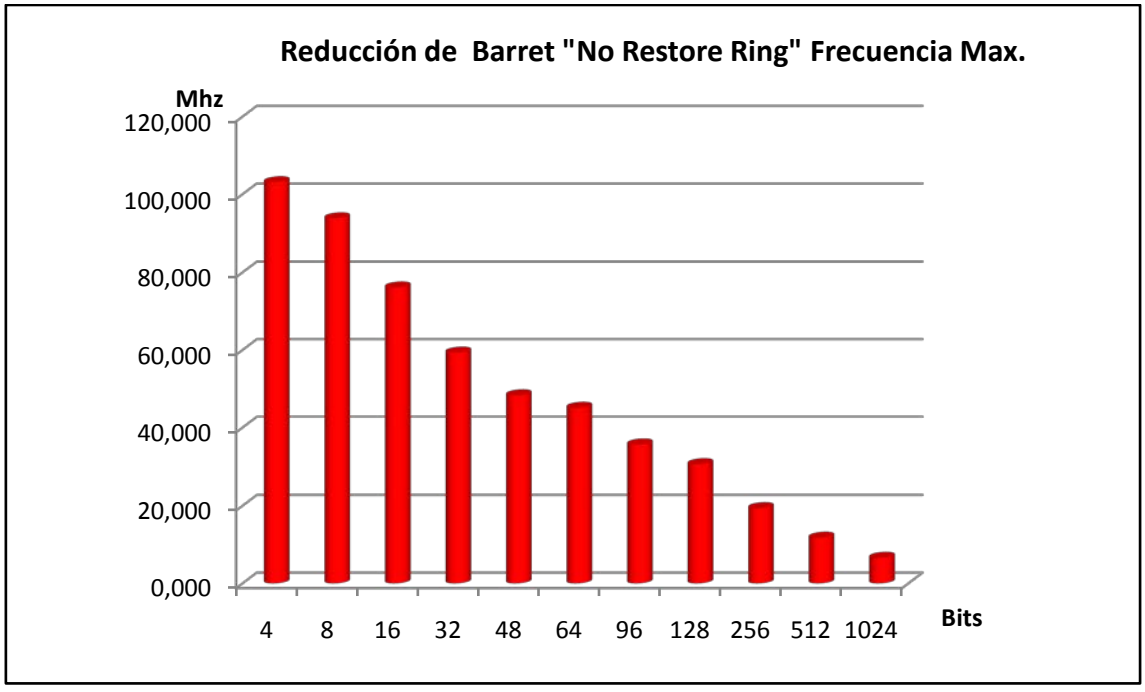

Ilustración 73: Frecuencia máxima Algoritmo “No Restore Ring”

8.2.3. Resumen de Reductores

En este punto presentamos un resumen de las propiedades más importante de los reductores que hemos realizado. Veremos graficas de comparación de áreas y del factor área x tiempo.

En primer lugar comparamos los datos de área para los reductores por el algoritmo de "Naive" y el algoritmo "No restore Ring".

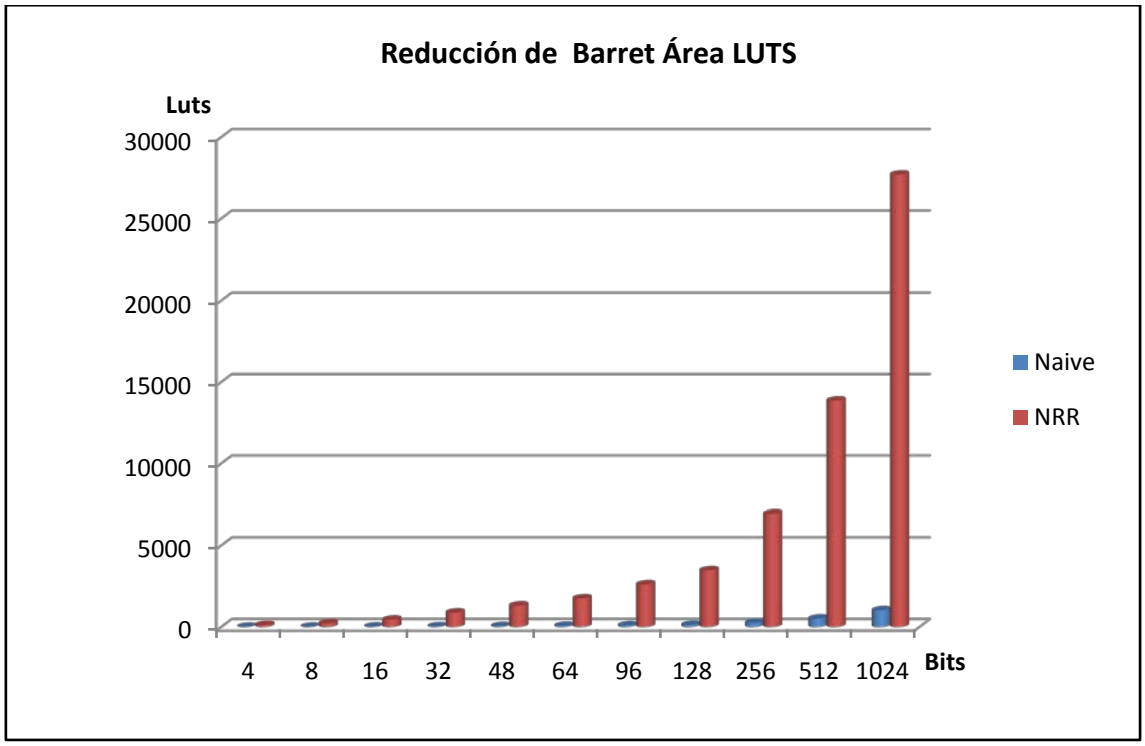

Ilustración 74: Comparación Área Reductores

Podemos ver en el siguiente grafico las comparaciones los datos para el factor Área x Tiempo de los reductores por el algoritmo de "Naive" y el algoritmo "No Restore Ring".

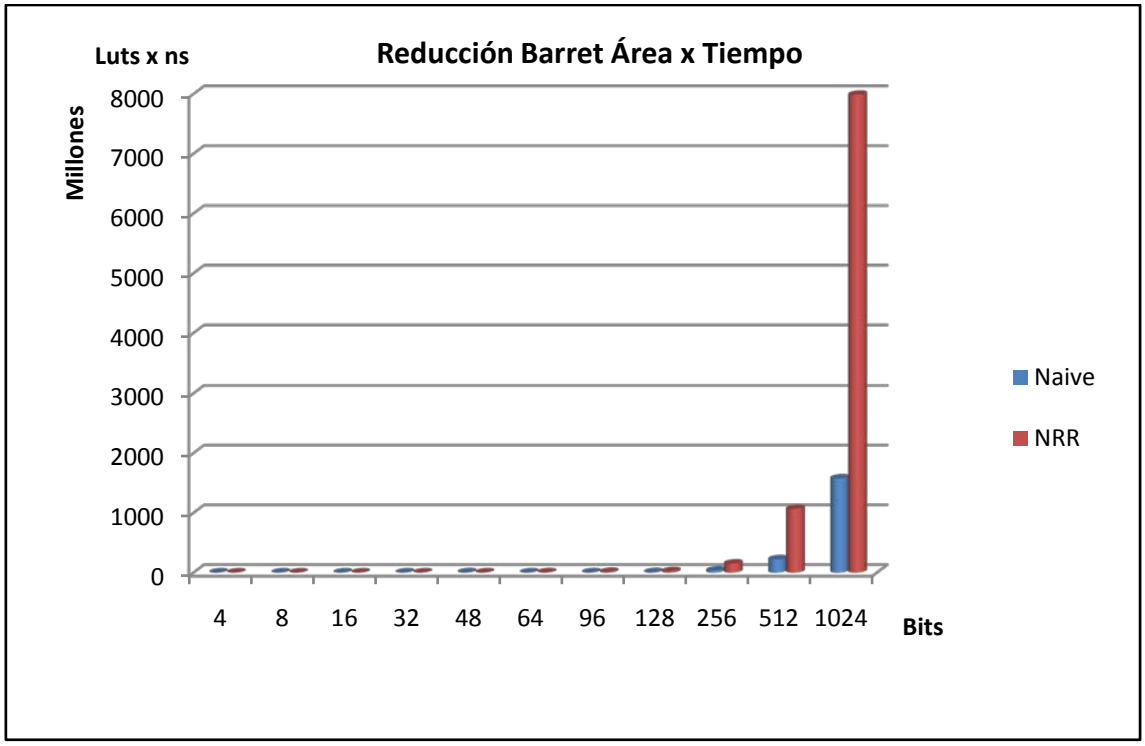

Ilustración 75: Comparación factor Área x Tiempo Reductores

Por último los datos de Flip Flop para los reductores por el algoritmo de "Naive" y el algoritmo "No Restore Ring".

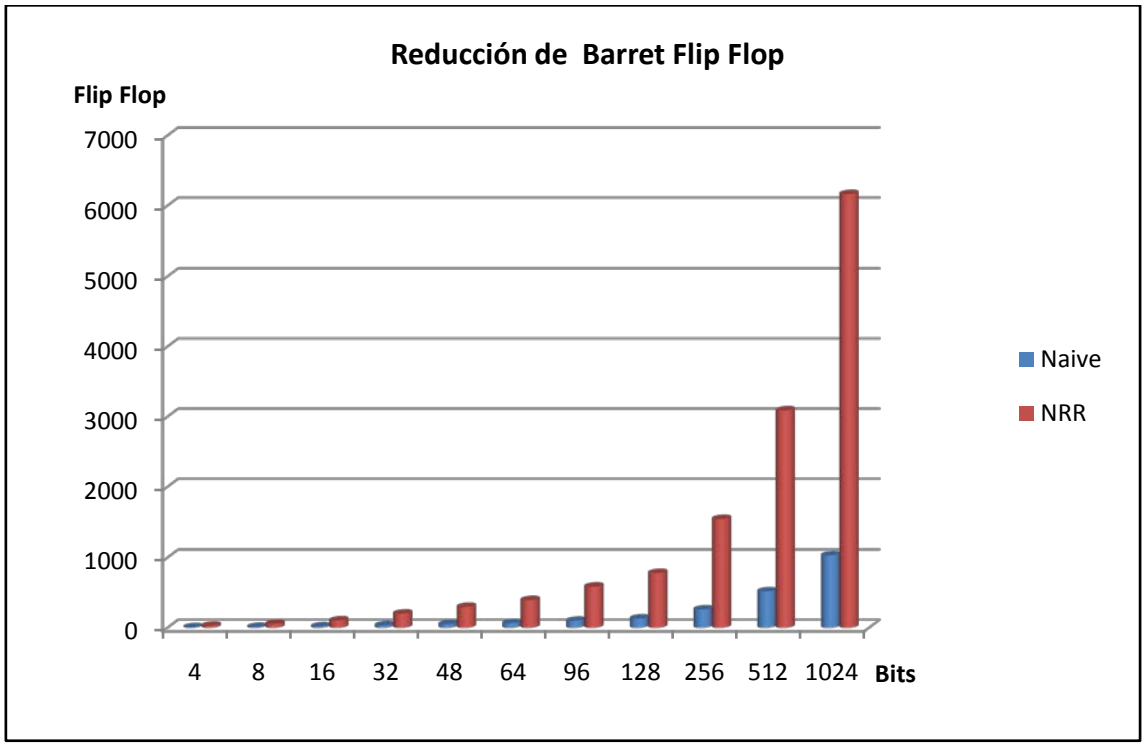

Ilustración 76: Comparación Flip Flop reductores

8.3. Resultados Multiplicación Modular

8.3.1. Resultados método indirecto Multiplicador "Add & Shfit" con reductor "No Restore Ring"

Una vez realizado el código pertinente, que se puede ver en el anexo de códigos, pasamos a analizar los datos más relevantes que arroja el sintetizador los demás resultados se pueden observar en el anexo de síntesis. Hay que tener en cuenta que este algoritmo de multiplicación modular indirecto realiza una multiplicación modular en "19n" ciclos de reloj, siendo 'n' el número de bits los operandos de entrada.

N=256

Minimum period: 150.002ns (Maximum Frequency: 6.667MHz)
Minimum input arrival time before clock: 12.759ns
Maximum output required time after clock: 12.411ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	8937	13312		67%
Number of Slice Flip Flops	4137	26624		15%
Number of 4 input LUTs	16575	26624		62%
Number of bonded IOBs	1540	221	696%	
Number of GCLKs	1	8		12%

Ilustración 77: Resultados síntesis A&S y NRR de 256 bits

N=512

Minimum period: 281.845ns (Maximum Frequency: 3.548MHz)
Minimum input arrival time before clock: 13.624ns
Maximum output required time after clock: 12.537ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	18343	13312	137%	
Number of Slice Flip Flops	8257	26624		31%
Number of 4 input LUTs	33608	26624	126%	
Number of bonded IOBs	3076	221	1391%	
Number of GCLKs	1	8		12%

Ilustración 78: Resultados síntesis de 512 bits

En la siguiente tabla podemos ver un resumen de los datos más relevantes que nos da el sintetizador cuando realizamos la síntesis del algoritmo para los distintos anchos de palabra. Podemos ver un resumen tanto del área Flip Flor y Luts que ocupan el algoritmo así como la frecuencia máxima de funcionamiento como el factor área x tiempo.

A&S + NRR	4 Bits	8 Bits	16 Bits	32 Bits	48 Bits	64 Bits
LUTS	221	541	1029	2098	3036	1484
FLIP FLOP	75	141	271	592	787	1045
LUTS x ns	40633	222730	1050169	6056135	17027269	14271910
Frec. Max. (Mhz)	70,706	60,724	48,012	33,603	25,854	20,068
	221	541	1029	2098	3036	1484
A&S + NRR	**4 Bits**	**8 Bits**	**16 Bits**	**32 Bits**	**48 Bits**	
LUTS	6042	8303	16575	33608	23155	
FLIP FLOP	1560	2077	4137	8257	16497	
LUTS x ns	118180366	267189212	1911951742	14558843270	38631643620	
Frec. Max. (Mhz)	14,775	11,964	6,667	3,548	1,842	

Ilustración 79: Resumen A&S y NRR

Podemos ver un resumen de la frecuencia máxima de funcionamiento contra el ancho de palabra del algoritmo.

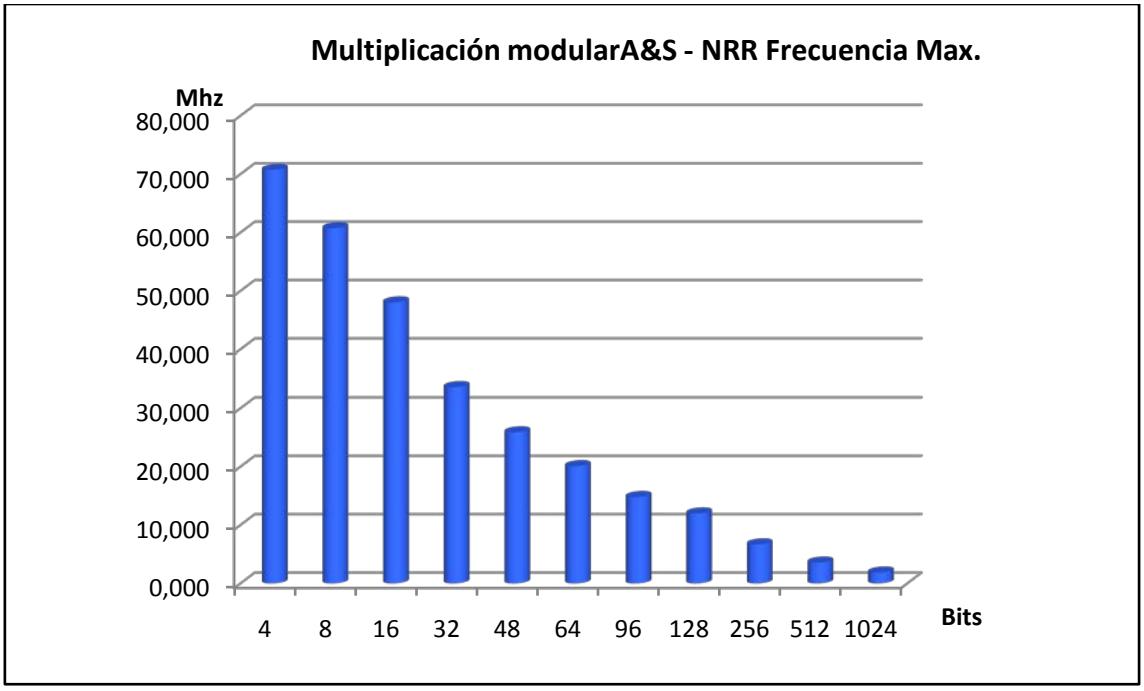

Ilustración 80: Frecuencia máxima A&S y NRR

8.3.2. Resultados método indirecto Multiplicador "AxB" con reductor "No Restore Ring"

Una vez realizado el código pertinente, que se puede ver en el anexo de códigos, pasamos a analizar los datos más relevantes que arroja el sintetizador los demás resultados se pueden observar en el anexo de síntesis. Hay que tener en cuenta que este algoritmo de multiplicación modular indirecto realiza una multiplicación modular en "16n" ciclos de reloj, siendo 'n' el número de bits los operandos de entrada. Teniendo en cuenta que este algoritmo solo es sintetizable hasta un ancho de palabra de 256 Bits. Vemos los siguientes resultados.

N=128

Minimum period: 83.584ns (Maximum Frequency: 11.964MHz)
Minimum input arrival time before clock: 39.715ns
Maximum output required time after clock: 12.589ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	12335	13312		92%
Number of Slice Flip Flops	1810	26624		6%
Number of 4 input LUTs	24107	26624		90%
Number of bonded IOBs	772	221	349%	
Number of GCLKs	1	8		12%

Ilustración 81: Resultados síntesis AxB y NRR de 128 bits

N=256

Minimum period: 149.926ns (Maximum Frequency: 6.670MHz)
Minimum input arrival time before clock: 59.919ns
Maximum output required time after clock: 12.446ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	41308	13312	310%	
Number of Slice Flip Flops	3619	26624		13%
Number of 4 input LUTs	81644	26624	306%	
Number of bonded IOBs	1540	221	696%	
Number of GCLKs	1	8		12%

Ilustración 82: Resultados síntesis AxB y NRR de 256 bits

En la siguiente tabla podemos ver un resumen de los datos más relevantes que nos da el sintetizador cuando realizamos la síntesis del algoritmo para los distintos anchos de palabra. Podemos ver un resumen tanto del área Flip Flor y Luts que ocupan el algoritmo así como la frecuencia máxima de funcionamiento como el factor área x tiempo.

AxB + NRR	4 Bits	8 Bits	16 Bits	32 Bits	48 Bits	64 Bits
LUTS	241	546	1227	2919	5254	8047
FLIP FLOP	63	120	233	458	648	911
LUTS x ns	34063	161848	873199	5733190	19968027	52298177
Frec. Max. (Mhz)	70,751	60,724	47,776	33,603	25,786	20,003

AxB + NRR	96 Bits	128 Bits	256 Bits	512Bits	1024 Bits	
LUTS	15167	24107	81644			
FLIP FLOP	1361	1810	3691			
LUTS x ns	199444564	519859548	6291772884			
Frec. Max. (Mhz)	14,753	11,964	6,670			

Ilustración 83: Resumen AxB y NRR”

Podemos ver un resumen de la frecuencia máxima de funcionamiento contra el ancho de palabra del algoritmo.

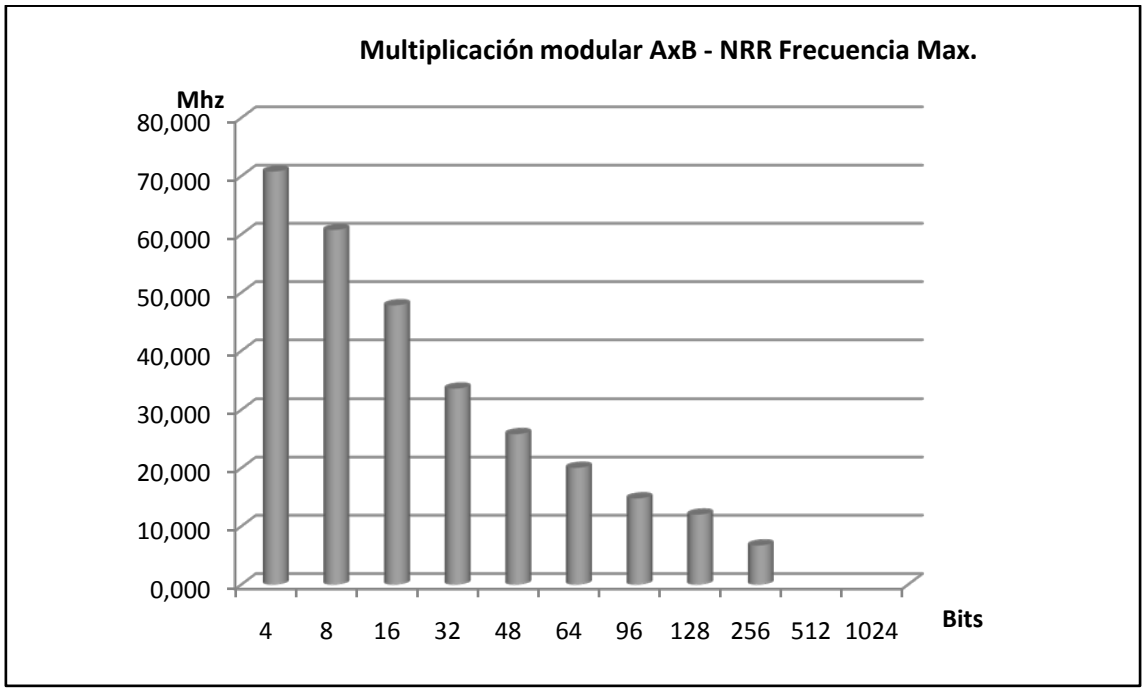

Ilustración 84: Frecuencia máxima AxB y NRR

8.3.3. Resultados implementación "Montgomery"

Una vez realizado el código pertinente, que se puede ver en el anexo de códigos, pasamos a analizar los datos más relevantes que arroja el sintetizador los demás resultados se pueden observar en el anexo de síntesis. Hay que tener en cuenta que este algoritmo de multiplicación modular indirecto realiza una multiplicación modular en "4n" ciclos de reloj, siendo 'n' el número de bits los operandos de entrada.

N=256

Minimum period: 33.730ns (Maximum Frequency: 29.647MHz)
Minimum input arrival time before clock: 29.531ns
Maximum output required time after clock: 12.318ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	1609	13312		12%
Number of Slice Flip Flops	526	26624		1%
Number of 4 input LUTs	3099	26624		11%
Number of bonded IOBs	1028	221	465%	
Number of GCLKs	1	8		12%

Ilustración 85: Resultados síntesis Montgomery 256 bits

N=512

Minimum period: 54.513ns (Maximum Frequency: 18.344MHz)
Minimum input arrival time before clock: 50.679ns
Maximum output required time after clock: 12.318ns

Device Utilization Summary (estimated values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	3212	13312		24%
Number of Slice Flip Flops	1045	26624		3%
Number of 4 input LUTs	6187	26624		23%
Number of bonded IOBs	2052	221	928%	
Number of GCLKs	1	8		12%

Ilustración 86: Resultados síntesis Montgomery 512 bits

En la siguiente tabla podemos ver un resumen de los datos más relevantes que nos da el sintetizador cuando realizamos la síntesis del algoritmo para los distintos anchos de palabra. Podemos ver un resumen tanto del área Flip Flor y Luts que ocupan el algoritmo así como la frecuencia máxima de funcionamiento como el factor área x tiempo.

Montgomery	4 Bits	8 Bits	16 Bits	32 Bits	48 Bits	64 Bits
LUTS	56	104	221	398	595	787
FLIP FLOP	12	21	46	71	104	136
LUTS x ns	5071	20512	76406	365090	972754	1879935
Frec. Max. (Mhz)	79,239	70,641	61,286	47,998	39,559	33,603

Montgomery	96 Bits	128 Bits	256 Bits	512Bits	1024 Bits
LUTS	1175	1558	3099	6187	12350
FLIP FLOP	201	267	526	1045	2081
LUTS x ns	5140522	9759412	53518986	345480496	2244508365
Frec. Max. (Mhz)	25,850	20,066	11,964	6,671	3,552

Ilustración 87: Resumen Montgomery

Podemos ver un resumen de la frecuencia máxima de funcionamiento contra el ancho de palabra del algoritmo.

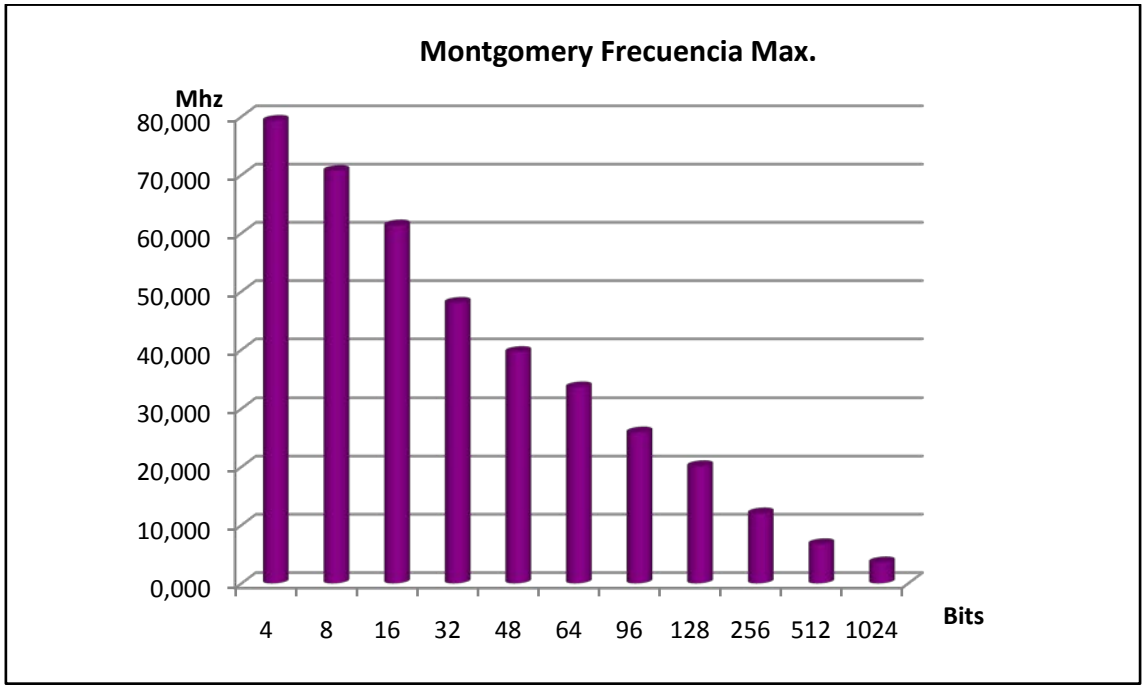

Ilustración 88: Frecuencia máxima Montgomery

8.3.4. Resumen de multiplicadores modulares

En este punto presentamos un resumen de las propiedades más importante de los multiplicadores modulares que hemos implementado. Compararemos los dos métodos indirectos vistos como el método directo. Veremos graficas de comparación de áreas y del factor área x tiempo y de los biestables (Flip Flop) que estos algoritmos ocupan.

En primer lugar comparamos los datos de área para los distintos multiplicadores modulares.

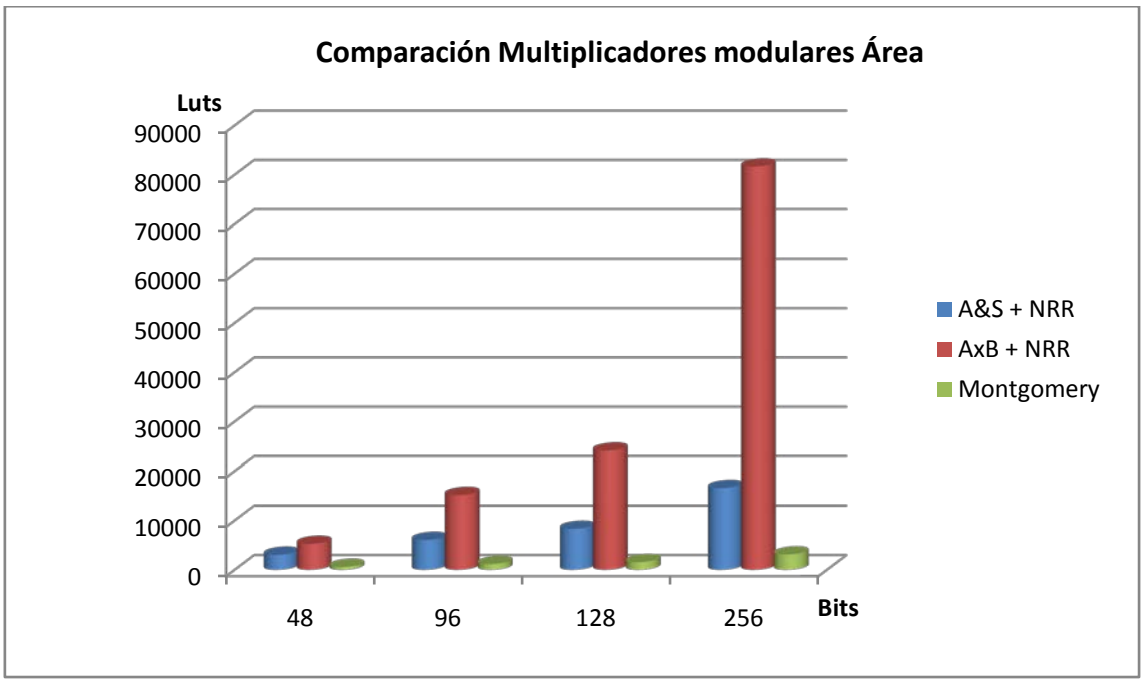

Ilustración 89: Comparación Área multiplicadores modulares

Podemos ver en el siguiente grafico las comparaciones los datos para el factor Área x Tiempo de los multiplicadores modulares

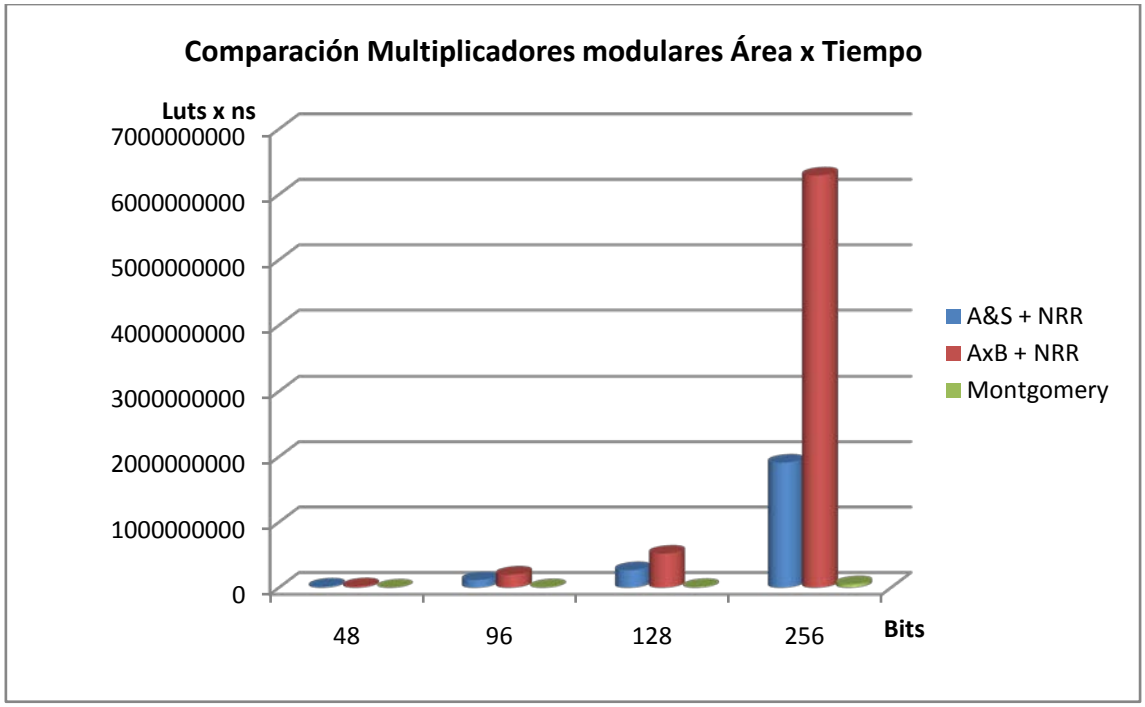

Ilustración 90: Comparación factor Área x Tiempo Multiplicadores modulares

Por último los datos de Flip Flop para los multiplicadores modulares.

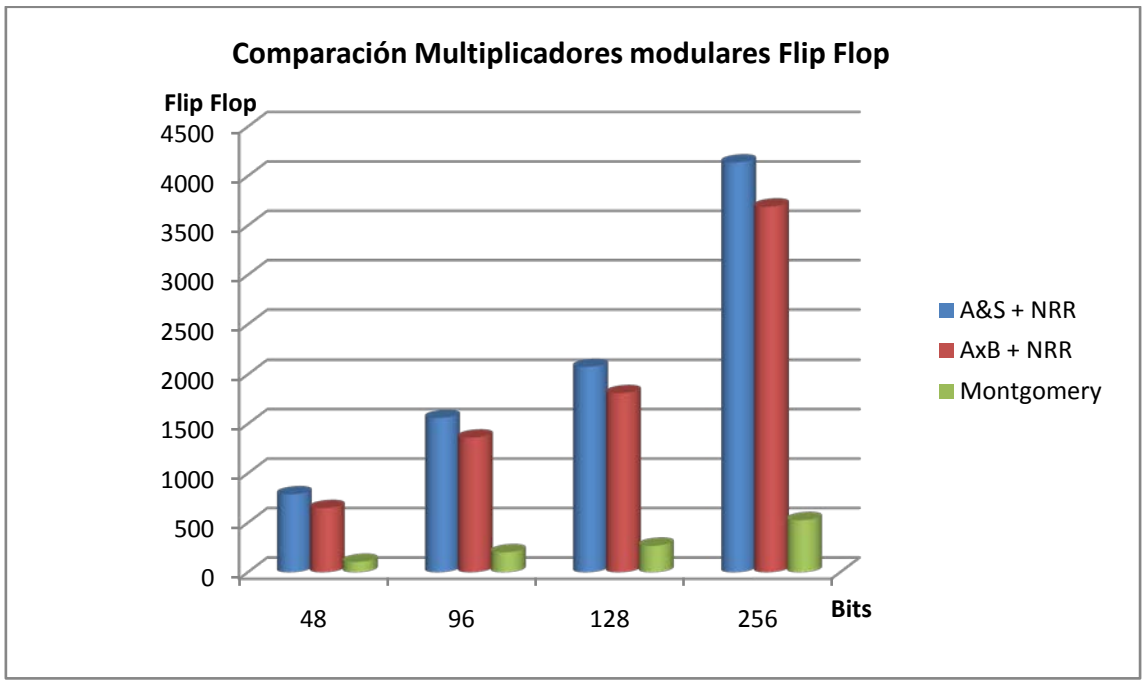

Ilustración 91: Comparación Flip Flop multiplicadores modulares

8.4. Resultado Exponenciación Modular

8.4.1. Resultado Exponenciación modular mediante el algoritmo "Left to Right"

Una vez realizado el código pertinente, que se puede ver en el anexo de códigos, pasamos a analizar los datos más relevantes que arroja el sintetizador los demás resultados se pueden observar en el anexo de síntesis. Hay que tener en cuenta que este algoritmo de exponenciación modular realiza una exponenciación modular en "48n" ciclos de reloj, siendo 'n' el número de bits los operandos de entrada.

N=256

Minimum period: 34.521ns (Maximum Frequency: 28.967MHz)
Minimum input arrival time before clock: 23.812ns
Maximum output required time after clock: 9.225ns

Device Utilization Summary (estimated values)				
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	2528	13312		18%
Number of Slice Flip Flops	1073	26624		4%
Number of 4 input LUTs	4702	26624		17%
Number of bonded IOBs	775	221	350%	
Number of GCLKs	1	8		12%

Ilustración 92: Resultados síntesis Exponenciación Modular de 256 bits

N=512

Minimum period: 55.341ns (Maximum Frequency: 18.070MHz)
Minimum input arrival time before clock: 40.196ns
Maximum output required time after clock: 9.292ns

Device Utilization Summary (estimated values)				
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of Slices	5017	13312		37%
Number of Slice Flip Flops	2106	26624		7%
Number of 4 input LUTs	9332	26624		35%
Number of bonded IOBs	1543	221	698%	
Number of GCLKs	1	8		12%

Ilustración 93: Resultados síntesis Exponenciación Modular de 512 bits

En la siguiente tabla podemos ver un resumen de los datos más relevantes que nos da el sintetizador cuando realizamos la síntesis del algoritmo para los distintos anchos de palabra. Podemos ver un resumen tanto del área Flip Flor y Luts que ocupan el algoritmo así como la frecuencia máxima de funcionamiento como el factor área x tiempo

Exp Mod	4 Bits	8 Bits	16 Bits	32 Bits	48 Bits	64 Bits
LUTS	141	217	364	655	948	1236
FLIP FLOP	55	72	105	170	235	299
LUTS x ns	86183	275768	938810	3693807	9325994	17731767
Frec. Max. (Mhz)	88,347	80,263	76,770	69,156	59,161	53,952

Exp Mod	96 Bits	128 Bits	256 Bits	512Bits	1024 Bits
LUTS	1816	2391	4702	9332	18577
FLIP FLOP	428	558	1073	2106	4168
LUTS x ns	47612393	90683534	499483755	3176119604	20287267838
Frec. Max. (Mhz)	44,168	40,657	28,975	18,070	11,258

Ilustración 94: Resumen Exponenciación Modular

Podemos ver un resumen de la frecuencia máxima de funcionamiento contra el ancho de palabra del algoritmo.

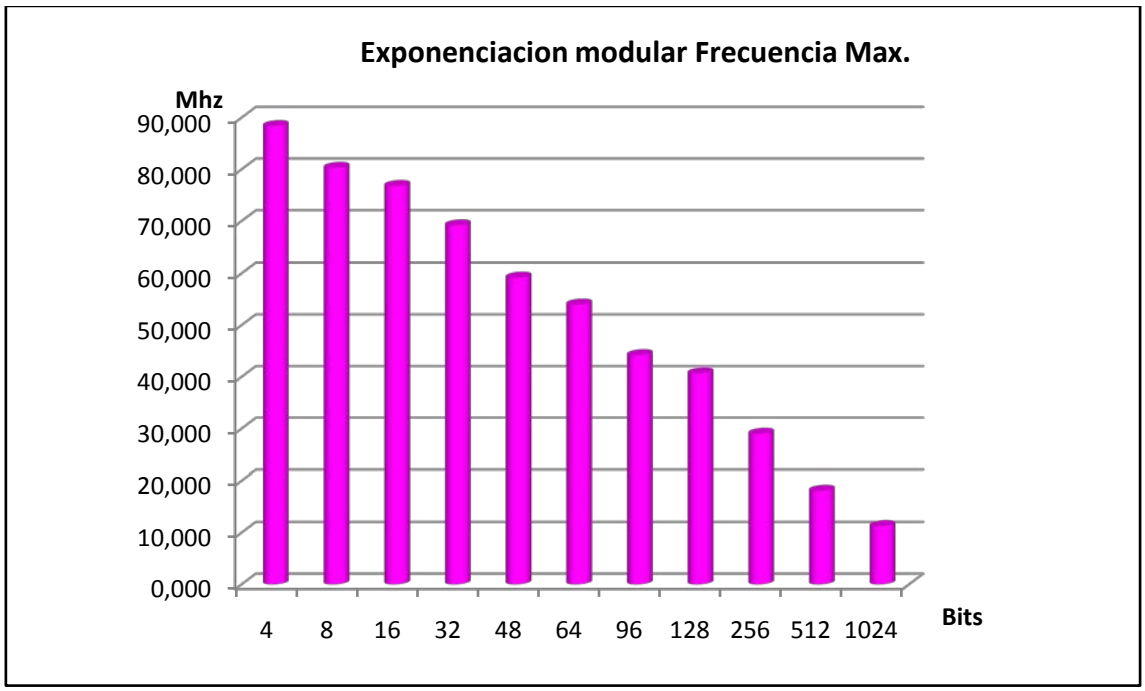

Ilustración 95: Frecuencia máxima Exponenciación Modular

8.5. Comparación de algoritmos

En este punto presentamos un resumen de las propiedades más importante de los distintos algoritmos que hemos implementado. Compararemos el algoritmo de multiplicación "Add and Shift", el multiplicador modular de "Montgomery" y el algoritmo de exponenciación modular. En los siguientes puntos del proyecto veremos que estos algoritmos son los que mejores propiedades tienen, con lo que son los mejores para poder ver comparaciones más ilustrativas.

Veremos graficas de comparación de áreas y del factor área x tiempo y de los biestables (Flip Flop) que estos algoritmos ocupan.

En primer lugar comparamos los datos de área para los tres algoritmos seleccionados para el estudio.

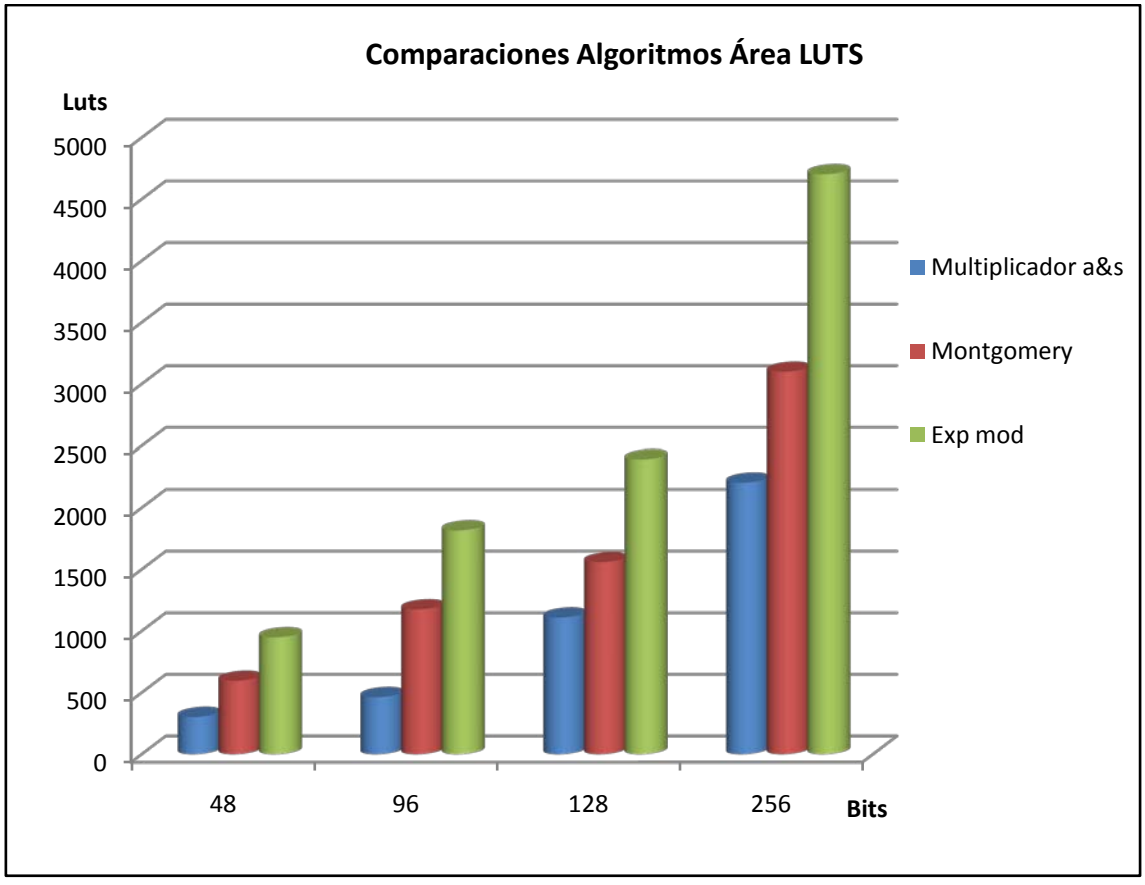

Ilustración 96: Comparación Área algoritmos

Podemos ver en el siguiente grafico las comparaciones los datos para el factor Área x Tiempo.

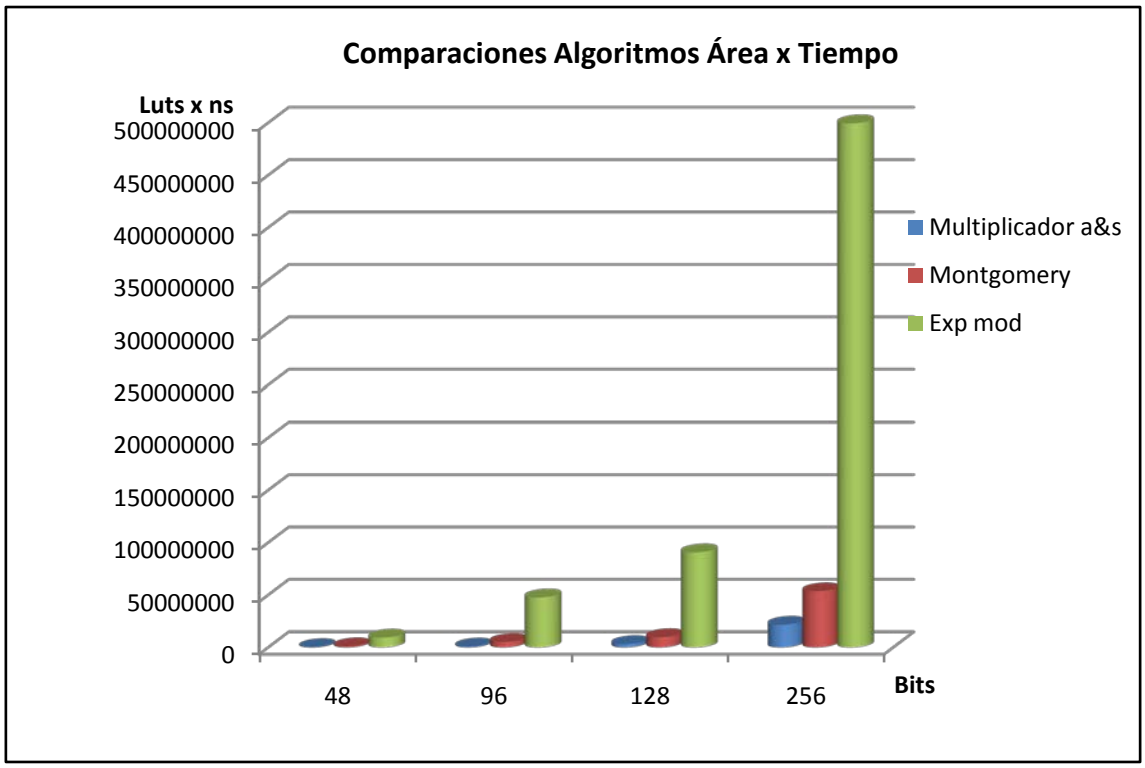

Ilustración 97: Comparación factor Área x Tiempo Algoritmos

En esta última gráfica podemos ver los datos de Flip Flop para los algoritmos.

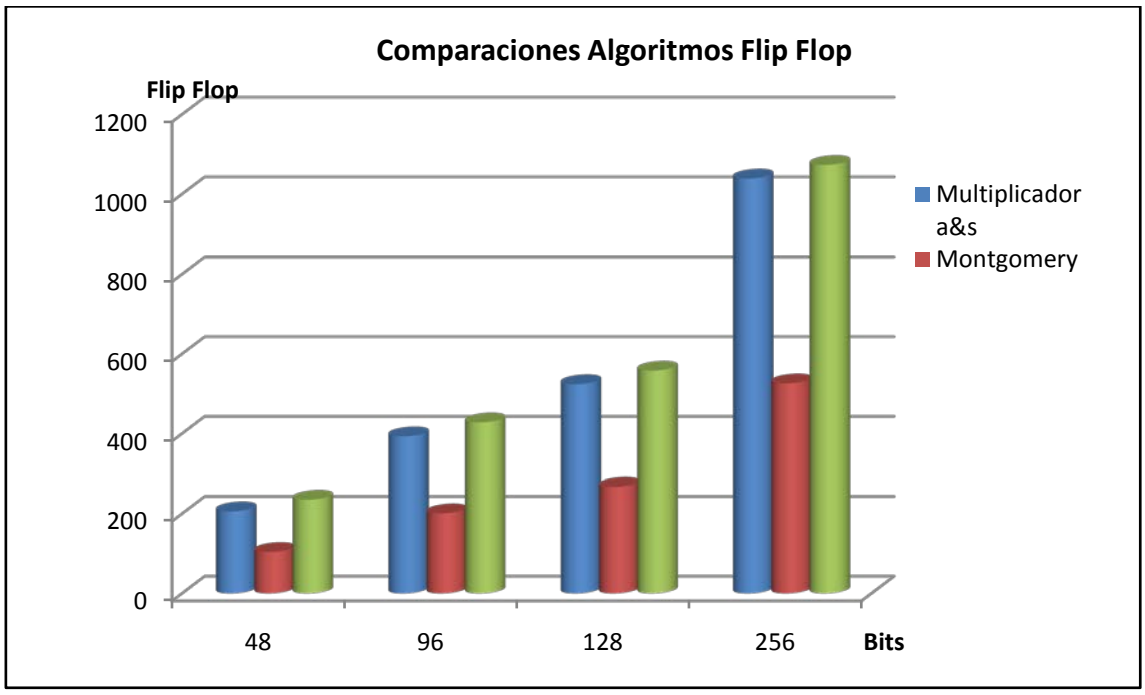

Ilustración 98: Comparación Flip Flop Algoritmos

Por último vemos un resumen de los ciclos de reloj necesarios para que cada algoritmo realice su función.

Algoritmo	Número de ciclos de Reloj	Ciclos de Reloj n=256	Freq. máxima con n= 256	Luts n = 256	Luts x tiempo n= 256
Add & Shift	n	256	26,102	2203	21606178
Booth	n	256	19,372	2345	30988518
A x B	1	1	16,260	67274	4137351
Naive	Xn	¿X?	39,110	1285	33644713
NRR	n	256	19,372	6938	148455883
Montgomery	2n	512	11,964	3099	53518986
A&S + NRR	3n+1	769	6,667	16575	1911951742
AxB + NRR	2n+2	514	6,670	81644	6291772884
Exp Mod	12n +6	3078	28,975	4702	499483755

Ilustración 99: Resumen ciclos de reloj algoritmos

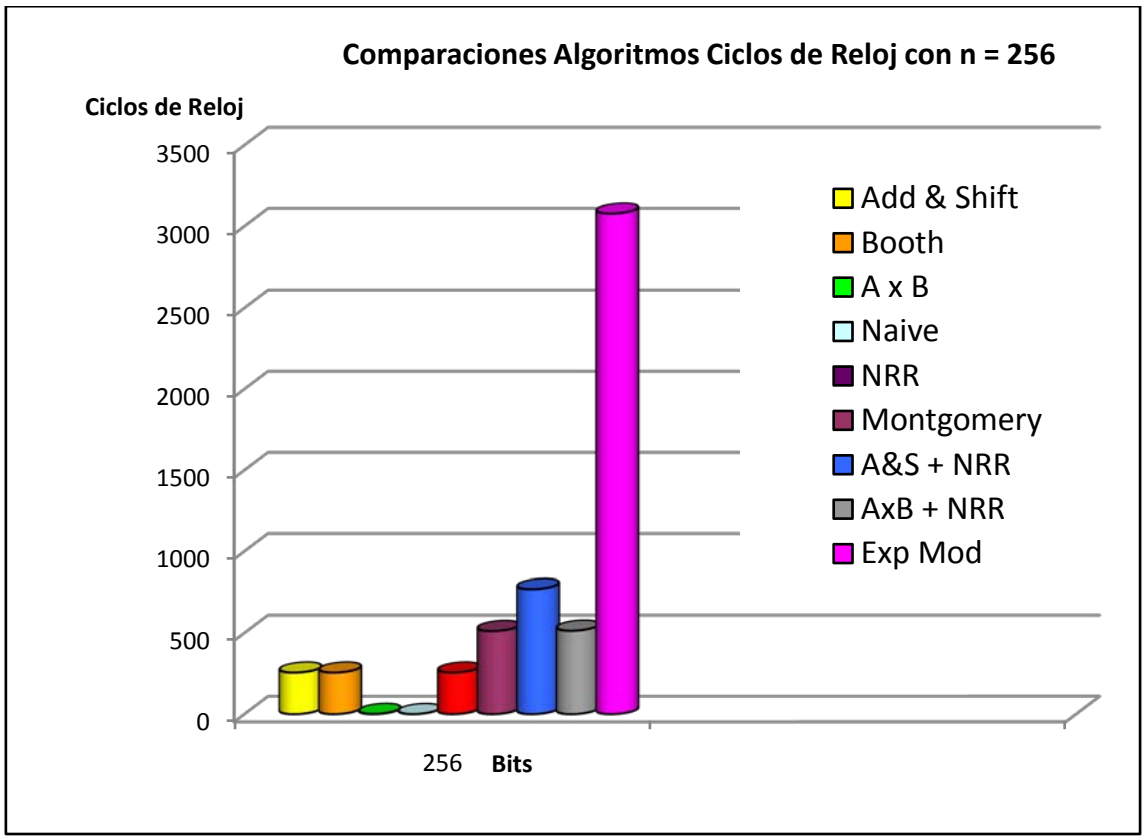

Ilustración 100: Resumen algoritmos ciclos de reloj n=256

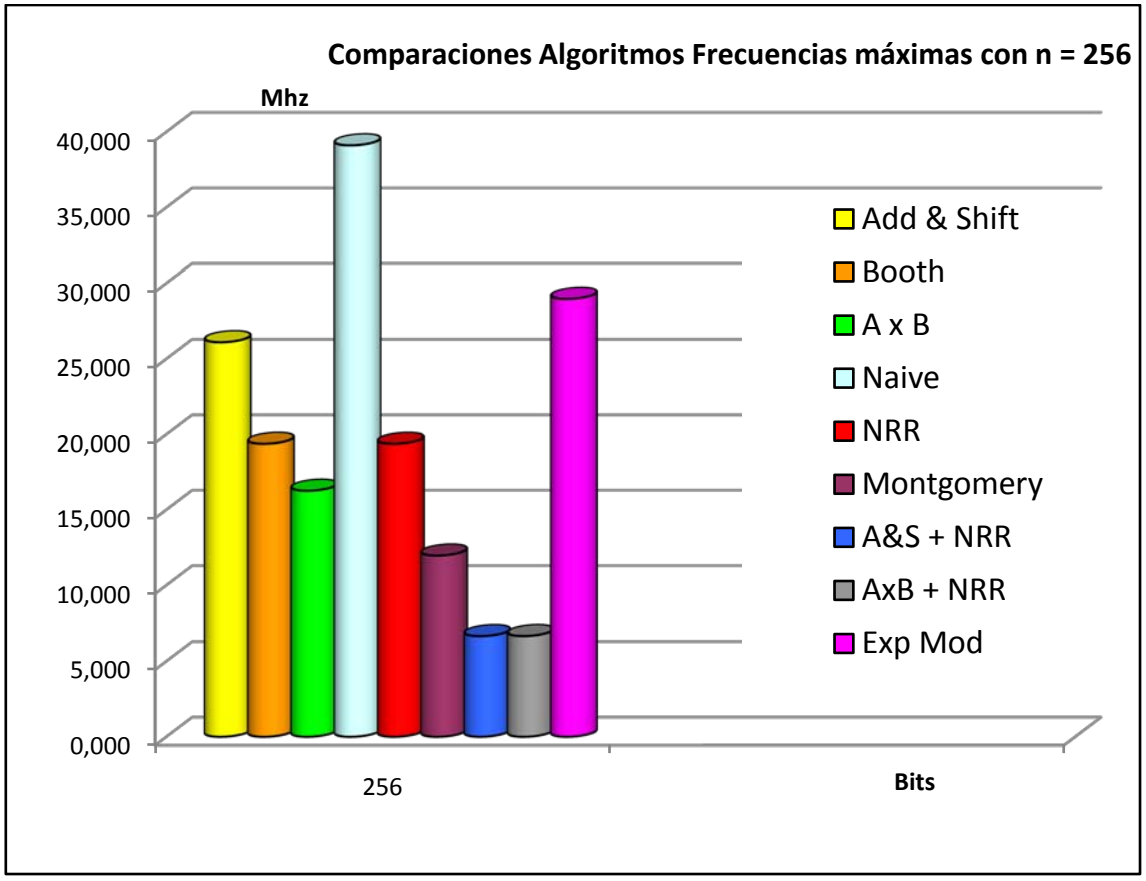

Ilustración 101: Resumen algoritmos frecuencias máx n= 256

CAPÍTULO V - CONCLUSIONES

9. Conclusiones

Una vez realizado los distintos algoritmos, realizado las pruebas pertinentes, tanto en el simulador como en el sintetizador y comparando los diferentes resultados obtenidos, podemos llegar a las siguientes conclusiones.

Si repasamos los objetivos de nuestro proyecto, estos son los objetivos principales que se van a intentar desarrollar a lo largo de este proyecto:

1. Selección de distintos algoritmos matemáticos.
2. Implementación de algoritmos seleccionados.
3. Sintetización de los algoritmos matemáticos.
4. Comparación de los distintos algoritmos.
5. Elección de algoritmo adecuado según área y tiempo de ejecución.

Según los objetivos marcados en los primeros puntos del proyecto, podemos ver como en primer lugar hemos realizado una selección de algoritmos. Hemos seleccionado los algoritmos matemáticos, más característicos que nos pueden servir a la hora de realizar un protocolo de seguridad para un sistema de etiquetas RFID. Con lo que podemos decir que hemos cumplido con el primer objetivo de nuestro proyecto

Seguidamente podemos decir que se han realizado la implementación de los diferentes algoritmos matemáticos propuestos en el lenguaje de programación VHDL, con lo que el segundo objetivo de nuestro proyecto se ha realizado con éxito, los códigos implementados se pueden ver el anexo de códigos incluidos en este proyecto.

Una vez implementados los diferentes algoritmos, estos se probaron en el simulador para ver su correcto funcionamiento, una vez realizada las adecuadas modificaciones para su correcto funcionamiento, estos algoritmos han pasado por la fase de sintetización, y podemos ver en el anexo de síntesis los diferentes resultados que hemos obtenido en este proceso. Con lo que damos por realizado el tercer objetivo de nuestro proyecto.

Para completar los dos últimos objetivos de nuestro proyecto, pasamos a ver las diferentes conclusiones que podemos sacar, una vez tenemos a nuestra disposición toda la información necesaria para completar nuestros objetivos.

En primer lugar hablaremos de las conclusiones que podemos sacar una vez realizado la implementación de los multiplicadores que hemos utilizado en nuestro proyecto.

Como podemos ver en los gráficos y en los datos obtenidos en el sintetizador, podemos decir que el circuito más conveniente para nuestro cometido, intentar tener el menos área posible, es el algoritmo “Add & Shift” este es el algoritmo que menos área requiere para realizar la operación de la multiplicación.

Como podemos ver el algoritmo de “Booth”, ocupa unos cuantos “luts” más que el algoritmo “Add & Shift”, pero tiene peor factor área x tiempo, con lo que este algoritmo queda descartado para las distintas pruebas que se tengan que hacer utilizando multiplicadores.

En cuanto al algoritmo “A*B” vemos que supera clarísimamente los “luts” utilizados, salvo para los casos en que se utilicen números de 4 o 8 bits, cosa que es poco probable, ya que lo más normal es que estas aplicaciones utilicen anchos de palabra mayores de 128 bits.

Por otro lado vemos que el algoritmo “A*B”, posee el menor factor área x tiempo, es debido a que las multiplicaciones las realiza en un solo ciclo de reloj, aunque el tiempo no es un factor decisivo en nuestro estudio este multiplicador también podría quedar descartado, debido a que en los anteriores resultados vemos que es el que más área ocupa. Si el tiempo de ejecución es el parámetro mas critico este sería el mejor multiplicador.

Como observamos en estas primeras conclusiones, en caso de que tuviéramos que implementar algún tipo de algoritmo que realice tareas de encriptación para las etiquetas RFID, basándose en algoritmos de multiplicación, utilizaríamos la multiplicación por el algoritmo de “Add & Shift”, ya que es el que presenta las mejores características en tamaño.

El algoritmo que mejor cumple nuestras especificaciones de tener una menor área y un buen factor área x tiempo es el algoritmo “Add & Shift”, con lo que este será el utilizado para realizar la multiplicación modular de forma indirecta, es decir, un multiplicador y reductor. Viendo que el algoritmo “AxB”, tiene el menor tiempo de ejecución también se utilizara para realizar la multiplicación modular de forma indirecta, ya que nos servirá para comparar la utilidad de este primero frente a un algoritmo que da unas características algo diferentes.

El algoritmo de “Booth”, presenta características similares al “Add & Shift”, con lo que no vemos conveniente su uso.

Una vez visto las conclusiones de los algoritmos de multiplicación en este punto del proyecto expondremos las conclusiones que hemos podido obtener de los según los datos obtenidos del sintetizador y de las distintas graficas obtenidas de los algoritmos que realizan reducciones.

De los dos reductores estudiados podemos decir que el que mejor se ajusta a nuestro cometido es el algoritmo “No Restore Ring”, aunque este ocupe un mayor número de LUTS y Flip Flop, la razón por la cual elegimos este algoritmo se debe al factor tiempo, ya que este es siempre constate independientemente del valor de los operandos, con lo que siempre de antemano se va a conocer el tiempo que se va requerir para realizar la reducción.

El principal problema del algoritmo de “Naive” es que según sean los operandos que entran a este algoritmo, el tiempo que tarda en dar respuesta depende de estos operandos, ya que como este algoritmo realiza restas sucesivas, cuanto mayor sea el operando de entrada y menor sea el modulo el algoritmo tardara mas con lo que se pueden dar casos en los cuales la salida del proceso sea muy larga, con lo que este aspectos puede entrañar dificultades para el funcionamiento del sistema de encriptación en las etiquetas RFID

En cambio el algoritmo “No Restore Ring” tarda lo mismo independientemente de cuáles sean los operandos.

En cuanto el factor área x tiempo, tendremos el mismo problema. Aunque observando la grafica de factor área x tiempo podemos ver que los valores son bastante iguales cuando el ancho de palabra no supera los 128 bits.

Con lo que por lo cual hemos elegido el algoritmo “No Restore Ring” como el mejor algoritmo para realizar la reducción de un número. Ya que para anchos de palabra de un tamaño relevante se comporta mejor, ya que hay que tener en cuenta que este reductor se coloca detrás de un multiplicador, con lo que normalmente si el sistema funciona con un ancho de palabra ‘n’, el multiplicador devuelva un resultado de ‘2n’, y como hemos visto para anchos de palabra grandes este reductor se comporta de forma más estable y definida.

Acabamos de ver las conclusiones de las operaciones matemáticas que realizan solamente reducciones, y seguidamente analizaremos las multiplicaciones modulares, ya sean con el método directo, algoritmo de Montgomery, o con los métodos indirectos de multiplicación que hemos implementado.

Según los resultados del sintetizador, el mejor algoritmo para realizar la multiplicación modular es el algoritmo de “Montgomery”, ya que este es el que mejor cumple las necesidades de nuestro proyecto para etiquetas RFID, este algoritmo es el que mejor características ofrece, ya que tiene un menor tamaño, y una mejor relación área x tiempo, que los otras dos combinaciones de multiplicador y reductor, en este punto es donde realmente podemos ver por qué debemos utilizar el algoritmo de “Montgomery”, ya que su relación área x tiempo destaca sobre las demás.

Como podemos ver en las graficas, los otros dos algoritmos, tanto el que utiliza un multiplicador “Add & Shift” como la otra propuesta de multiplicador, es decir el que hemos llamado “AxB, tiene unas peores propiedades para el proyecto que nos ocupa. En primer lugar los dos algoritmos ocupan un mayor número de LUTS y Flip Flop, especialmente el algoritmo “AxB” que doblo o incluso triplica el espacio requerido. Si comparamos el factor área x tiempo, esperamos que el algoritmo “AxB” mejore, pero en cambio vemos que sigue siendo el peor de los tres, aunque este solo tarde un ciclo de reloj en realizar la multiplicación, debido a su gran tamaño este sigue siendo el que da peores resultados. En este punto también el algoritmo de “add & Shift” presenta un peor factor área x tiempo.

Como hemos visto en las partes anteriores del proyecto, en caso de que tuviéramos que implementar algún tipo de algoritmo que realice tareas de encriptación para las etiquetas RFID, basándose en multiplicación modular, utilizaríamos la multiplicación modular por el algoritmo de “Montgomery”, ya que es el que presenta las mejores características tanto en tamaño como en tiempo de respuesta

Como conclusión final, podemos decir que para realizar la siguiente parte del proyecto, es decir realizar un exponenciador modular, utilizaremos el multiplicador modular con el algoritmo de “Montgomery”, este será el elegido para realizar el siguiente paso del proyecto, realizar la exponenciación modular.

Como hemos comentado, una de las mejores formas de realizar una multiplicación modular es mediante el algoritmo de “Montgomery”, nuestro exponenciador modular se basara en este algoritmo por lo cual se ha decidido elegir el algoritmo “Left to Right” para realizar esta operación matemática.

Para completar el cuarto y quinto objetivo de nuestro proyecto podemos comparar los distintos algoritmos, para poder ver qué ventajas y desventajas supone utilizar un método u otro para realizar una implementación de un sistema de encriptación para la tecnología RFID.

Hay que tener en cuenta que cuanto más compleja sea la operación, la seguridad del algoritmo de encriptación es mejor, es decir, es más difícil romper la seguridad si la encriptación se realiza con un exponenciador modular que si se realiza con un algoritmo de multiplicación modular o si se realiza la implementación del sistema de seguridad con un simple multiplicador.

Hay algoritmos de encriptación que utilizan los algoritmos matemáticas que hemos implementado, pero nuestro cometido no es ver cuál de ellos es mejor, sabiendo que cuanto mayor sea la complejidad de la operación mayor es la seguridad de los datos que transfieren las etiquetas, intentamos ver el compromiso que se da entre seguridad y los factores principales que hemos considerado en nuestro diseño de hardware, es decir entre la seguridad que puede aportar los distintos algoritmos matemáticos y el tamaño y tiempo que dan las distintas simulaciones y síntesis de los algoritmos.

Según el último objetivo del proyecto, encontrar un algoritmo adecuado para poder proporcionar, en un futuro, a la tecnología RFID un protocolo de seguridad adecuado, es decir, un protocolo que se ajuste tanto en tamaño que pueda ocupar en una tarjeta RFID como en velocidad de ejecución necesaria para que este de respuesta y como no que sea difícil de franquear por medios externos, concluimos que:

Una vez realizadas las pruebas pertinentes a todos los algoritmos, y comparando los resultados de las distintas síntesis y simulaciones aportadas en el proyecto. Una de las conclusiones a la que hemos llegado es que la multiplicación simple cumple los requisitos de velocidad de ejecución, ya que es el método más rápido de los tres métodos estudiados.

En cuanto al área, ya que este algoritmo es el más sencillo, es el que menos área ocupa. En contra partida es un método demasiado frágil en cuanto a la seguridad se refiere, en casi ningún protocolo de encriptación moderno, pensado para la tecnología RFID, se utiliza como elemento principal de encriptación, con lo que no se aconseja utilizar para futuros estudios como en un posible elemento principal de un protocolo de encriptación para la tecnología RFID.

Otro de los algoritmos estudiados, es la exponenciación modular, este tipo de algoritmos es utilizado en muchos sistemas criptográficos modernos. La exponenciación modular proporciona un sistema de encriptación en las comunicaciones lo bastante robusto como para proporcionar una gran seguridad ante el ataque al sistema. Con lo que si buscáramos extremar la seguridad de las comunicaciones entre distintos componentes de un entramado de etiquetas, lectores y demás componentes que integran una red RFID este sería el algoritmo seleccionado.

Teniendo en cuenta que esta tecnología está destinada al uso, comercial, logístico, controles de acceso… creemos que por las desventajas que ahora comentaremos, este algoritmo presenta más inconvenientes que ventajas.

Los principales problemas de la exponenciación modular son que no cumple ninguna de nuestras dos condiciones de diseño, en cuanto a área ocupada, este algoritmo destaca considerablemente, ya que como podemos ver en las diferentes síntesis este algoritmo tiene una mayor ocupación de área que los demás algoritmos seleccionados.

Pero realmente el principal problema de este algoritmo es su factor área x tiempo, que como podemos ver en las diferentes graficas, es 20 veces mayor a los demás algoritmos, esto significa que el tiempo de ejecución del exponenciador modular es mucho más alto que el de los otros algoritmos.

Con lo que por esto dos principales problemas, desaconsejamos la utilización de la exponenciación modular como elemento principal para realiza un algoritmo de encriptación para la tecnología RFID.

Con lo que pensamos que el mejor método para realizar un algoritmo criptográfico es el multiplicador modular, ya que con este algoritmo somos capaces de mantener un equilibrio entre los dos factores fundamentales de nuestro estudio, área, tiempo de ejecución y por otro lado factores de seguridad.

Con este tipo de algoritmos, se pueden realizar protocolos de seguridad lo suficientemente robustos, para mantener la seguridad de las comunicaciones entre los distintos componentes que se encuentran en una red RFID. Los datos pueden ser encriptados de forma que agentes externos al sistema les sea realmente complicado realizar alguna modificación en ellos, o realizar algún ataque con el fin de falsificar la comunicación entre los diferentes componentes en la red RFID.

Este tipo de algoritmo a su vez mantiene un compromiso adecuado en lo que se refiere a ocupación de área, ya que se mantiene en unos niveles similares a lo que pueda ocupar un simple multiplicador. Es cierto que la ocupación es mayor que en el algoritmo de multiplicación, pero actualmente existen tarjetas RFID que podrían asumir esta ocupación de área sin ningún problema, y dejando el suficiente espacio para los diferentes módulos de Hardware necesarios para el correcto funcionamiento de la tarjeta.

En cuanto al factor área x tiempo, este algoritmo no es un algoritmo relativamente lento, con lo que podemos decir que aunque tarde más que la multiplicación, con la actual tecnología de las etiqueta RFID, somos capaces de proporcionar frecuencias de reloj adecuadas para que la utilización de este algoritmo de unos tiempo de respuesta aceptables.

Con lo que para definir el último objetivo de nuestro proyecto podemos decir que el mejor algoritmo que cumple con nuestras necesidades es la multiplicación modular por el método de "Montgomery".

CAPÍTULO VI - BIBLIOGRAFÍA

10. Bibliografía

1. "The Designer's Guide to VHDL", published by Morgan Kaufmann Publishers, ISBN 1-5860-270-4. The new book covers VHDL-93

2. Bhattacharya, Shaoni; 2005. "Electronic tags for eggs, sperm and embryos" en New cientist.com, 2 de abril de 2005 Roger Smith: RFID: A Brief Technology Analysis, CTO Network Library, 2005.

3. A Review of Modular Multiplication Methods and Respective Hardware Implementations Nadia Nedjah Department of Electronics Engineering and Telecommunications, Engineering Faculty, State University of Rio de Janeiro, Rio de Janeiro, Brazil nadia@eng.uerj.br, Luiza de Macedo Mourelle Department of System Engineering and Computation, Engineering Faculty, State University of Rio de Janeiro, Rio de Janeiro, Brazil

4. Implementación en VHDL de un módulo de multiplicación basado en algoritmos de Booth Sánchez, E. Baz, I. Gutiérrez, R. Nava, S. García Instituto Tecnológico de Orizaba División de Estudios de posgrado, Orizaba, Veracruz, México

5. Implementación de la multiplicación modular de Montgomery en hardware reprogramable Escuela de Ingeniería Eléctrica y Electrónica, Universidad del Valle, Cali, Colombia. R. Palacios; R. Nieto y A. Bernal

6. SLAP – a Secure but Light Authentication Protocol for RFID based on Modular Exponentiation Batbold Toiruul1, KyungOh Lee1, and JinMook Kim1 Computer Science Department, Sunmoon University, ChungNam 336-708, Korea

7. Implementación de la función potencial modular en hardware reprogramable. Freddy Bolaños, Rubén Nieto, Álvaro Bernal. Escuela de ingeniería eléctrica y electrónica. Grupo de arquitecturas digitales y microelectrónica. Universidad del valle Colombia

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

9. RFID : la tecnología de identificación por radiofrecuencia Muñoz Frías, José Daniel

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

11. 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.

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

13. 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

14. 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 Prs, pp. 226-235, 2002

15. 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

CAPÍTULO VII - ANEXOS