ACELERACIÓN HARDWARE CON PROPÓSITOS CRIPTOGRÁFICOS

JAVIER PÉREZ RICO

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería de Computadores

Septiembre de 2013
Nota Final: 9 (SOBRESALIENTE)

Director:

José Luis Imaña Pascual

Autorización de Difusión

JAVIER PÉREZ RICO

Fecha

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Aceleración hardware con propósitos criptográficos", realizado durante el curso académico 2012-2013 bajo la dirección de José Luis Imaña Pascual en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

La criptografía asimétrica está siendo implementada hoy en día en muchas ramas, principalmente en el cifrado de clave pública y en las firmas digitales. Esto es debido principalmente a que no hace falta el intercambio de claves de cifrado entre el emisor y el receptor lo que hace que el sistema sea más seguro. Una variante de este tipo de criptografía es la criptografía de curvas elípticas. Este tipo de criptografía se basa en las matemáticas de curvas elípticas y la aritmética modular. Además, este tipo de criptografía es mucho más interesante que, por ejemplo, RSA ya que utiliza una clave de longitud menor para el mismo nivel de seguridad. En este trabajo se realiza una implementación de una nueva arquitectura de multiplicación modular para las operaciones sobre los cuerpos primos F_p, utilizada en este tipo de criptografía. Para ello se han utilizado algoritmos ya conocidos y revisiones de los mismos. Las revisiones y mejoras implementadas están orientadas al incremento de la velocidad de la operación de multiplicación. La implementación se realiza sobre dispositivos de hardware reconfigurable de la familia Virtex5 de Xilinx. Toda la implementación del sistema se hace mediante codificación en VHDL y el software de desarrollo de Xilinx ISE 14.2. De esta forma se puede estudiar mejor el comportamiento de los algoritmos bajo distintos casos de uso, anchos de bus y familias de FPGA.

Palabras clave

ECC, Criptografía, Curvas, Elípticas, Aceleración, Hardware, Karatsuba, Montgomery, Barret, VHDL, FPGA.

Abstract

Asymmetric cryptography is being implemented today in many branches, mainly in public-key encryption and digital signatures. This is mainly because you do not need the encryption key exchange between sender and receiver which makes the system more secure. A variant of this type of cryptography is elliptic curve cryptography. This type of cryptography is based on the mathematics of elliptic curves and modular arithmetic. Furthermore, this kind of encryption is much more interesting, for example, RSA key since it uses a shorter length for the same level of security. In this paper a new architecture implementing a modular multiplication operations over Fp prime fields used in this type of cryptography. This has been known algorithms used and revisions thereof. Revisions and improvements implemented are aimed at increasing the speed of the multiplication operation. The implementation is done on reconfigurable hardware devices in the Xilinx Virtex5. The whole system implementation is done using VHDL coding and development software Xilinx ISE 14.2. In this way you can study better the behavior of the algorithms under different use cases, bus widths and FPGA families.

Keywords

ECC, Criptography, Curve, Ellíptic, Aceleration, Hardware, Karatsuba, Montgomery, Barret, VHDL, FPGA.

Índice de contenidos

Agradecimientos

A mis padres, por darme esta oportunidad y apoyarme. A Cristina, por estar ahí, ayudarme, animarme a seguir siempre un poco más y no a rendirme nunca. También a mis amigos, en especial a Rubén, que tanto me han ayudado y apoyado.

Capítulo 1 - Introducción

1.1 Objetivos

Este trabajo tiene como objetivo principal el desarrollo completo de un sistema de módulos para la aceleración de la operación de multiplicación modular. Esta operación es clave en diversos algoritmos criptográficos como por ejemplo la criptografía de curvas elípticas sobre cuerpos primos. La multiplicación modular es muy costosa en tiempo, debido a que generalmente después de multiplicar hay que reducir el valor hasta su representante en módulo. Existen contadas excepciones, como el algoritmo de Montgomery (el cual se explica en este trabajo), que calcula la multiplicación modular de una sola vez. Los objetivos secundarios que se han considerado para realizar dicho objetivo principal son los siguientes:

1. Realizar un estado del arte de los diferentes algoritmos involucrados en la multiplicación modular para aprovecharlas en el diseño.
2. Investigar sobre un diseño que sea eficiente en el tiempo con los algoritmos ya expuestos.
3. Revisar y mejorar, en la medida de lo posible, los diseños anteriores para mejorar su velocidad.
4. Implementar dichos diseños usando el lenguaje VHDL y el programa de desarrollo ISE 14.2 de Xilinx.
5. Comparar las diferentes implementaciones (tanto propias como de trabajos de otros autores) del multiplicador modular para poder determinar las versiones que proporcionan un incremento de velocidad.

1.2 Antecedentes

Desde el inicio de la civilización ha existido una necesidad de ocultar información a extraños. En la antigüedad esta necesidad era, sobretodo, para proteger la confidencialidad de operaciones militares o de información política. Con esta necesidad nace la criptografía. Esta nueva disciplina trata de cifrar y codificar información para que solamente las personas autorizadas pueda acceder al contenido de la información.

Según la tecnología fue avanzando fueron más los campos donde se necesitaba cierta seguridad ante posibles filtraciones. Hoy en día y con la aparición de la informática son muchos más los ámbitos donde la confidencialidad de los datos está por encima de otros factores. Las comunicaciones digitales han producido un número creciente de problemas de seguridad. Las transacciones vía web también son susceptibles de ser interceptadas, modificadas, etc... Por ello se crearon algoritmos y protocolos capaces de dotar seguridad en las comunicaciones entre entidades de una red. Por esta razón, actualmente la criptografía es quien se encarga del estudio de dichos sistemas/protocolos/algoritmos teniendo como principales propiedades las siguientes:

- Confidencialidad: Garantiza que la información está accesible únicamente a personal autorizado.
- Integridad: Se garantiza la corrección y completitud de la información.
- No repudio: Proporciona protección ante la situación por la cual alguna de las entidades implicadas en la comunicación pueda negar haber participado en toda o parte de la comunicación.
- Autentificación: Proporciona mecanismos que permiten verificar la identidad del emisor ante el receptor.
- Asíncrono: Soluciones a problemas de la falta de simultaneidad en la tele-firma digital de contratos.

De la misma forma que la comunicación dio lugar a la criptografía, la criptografía dio lugar al criptoanálisis. El criptoanálisis se define como el conjunto de técnicas mediante las cuales los adversarios (aquellos participantes que quieren conocer el contenido cifrado pero no están autorizados para ello) intentan "romper" los sistemas criptográficos. Por "romper" puede entenderse que el adversario sea capaz de cumplir alguno de los siguientes objetivos:

- Saber que existe una comunicación entre dos entidades dadas.
- Poder leer el mensaje cifrado sin tener conocimiento de la clave.
- Obtener el valor de la clave secreta.
- Poder inyectar mensajes apócrifos en la comunicación, sin que estos sean detectados.

De esta forma, para un usuario malintencionado, el objetivo principal es conocer el contenido del mensaje sin tener la clave de cifrado. Además, de no poder lograrse, el sistema se diseña para que este objetivo se logre de manera muy poco eficiente.

Existen varios tipos de criptografía (criptografía de clave secreta, de doble clave, con umbral, basada en identidades, etc...) Estos tipos se dividen principalmente en 2 grupos: criptografía simétrica y asimétrica. La criptografía simétrica es aquella que se basa en el uso de una sola clave para cifrar y descifrar un mensaje. Esto significa que el emisor y el receptor tienen que ponerse de acuerdo primero sobre la clave que van a usar. El principal problema de usar este tipo de criptografía es que todo el peso de la seguridad lo tiene la clave.

Hoy en día, una clave con 64 bits puede ser encontrada usando un ordenador de sobremesa en varios días y usando un ordenador con más capacidad y proceso de cálculo como un "*mainframe"* entre varias horas y un día. Esto es posible ya que la cantidad de claves es relativamente pequeña y, por ejemplo, se pueden ir comprobando. La solución a este problema es usar claves mucho más largas usando 128, 192 ó 256 bits dependiendo del protocolo que se este utilizando. De este modo el número de claves posibles aumenta y el coste de probarlas todas aumenta de manera significativa. Un ejemplo de este tipo de cifrados es el algoritmo AES que cifra los mensajes por bloques de 128 bits.

Un problema mayor es el uso de las claves pues se necesitan $2n$ claves para la comunicación entre n personas. Además estos algoritmos necesitan una comunicación directa previa para concretar qué clave utilizar. De no poder establecerse un canal directo se enviaría en claro por una gran distancia aumentando el riesgo de seguridad. A este problema se le conoce como problema de secreto compartido. En 1976, Witfield Diffie y Martin Hellman [1], proponen una solución a este problema con un nuevo protocolo de secreto compartido usando la teoría fundamental de los números y de los cuerpos primos, explotando lo que se conoce como el problema del logaritmo discreto en cuerpos finitos.

Un cuerpo finito F_n está compuesto por n números enteros sobre los cuales se aplican las operaciones básicas de suma, resta, multiplicación e inversión modulares. Cuando nos referimos

a un cuerpo primo F_p el número n es un número primo. El problema del logaritmo discreto es un conocido problema por el cual es computacionalmente imposible conocer un número x conociendo b y, donde $bx=y$. Se denota como $x=log_b(y)$. Gracias a los protocolos de secreto compartido, como el de Diffie-Hellman, se sentaron las bases de la criptografía clave de pública o asimétrica, abriéndose un nuevo campo de la criptografía.

La criptografía asimétrica consta de dos claves, una privada y otra pública. Aunque las dos claves pertenecen a un mismo dueño, la clave pública la puede conocer todo el mundo mientras que la privada sólo la debería conocer una persona, el dueño. Con este tipo de criptografía no es necesario que emisor y receptor se comuniquen previamente para concretar la clave a utilizar. De este modo se evita el problema del intercambio. Además, en los algoritmos de clave asimétrica, la clave pública la pueden usar todas las personas que quieran iniciar la comunicación. De esta forma sólo se necesitan n pares de claves para n usuarios. Estos algoritmos se basan principalmente en el problema del logaritmo discreto y en la utilización cuerpos primos. Unos ejemplos de criptografía asimétrica son ElGamal [2] o la criptografía de curvas elípticas [3].

1.3 Motivación

Una de las diferencias entre la criptografía simétrica y la asimétrica, es que esta última necesita de una clave más corta para ofrecer el mismo nivel de seguridad que una clave simétrica. Esto es gracias principalmente al uso del problema del logaritmo discreto en cuerpos finitos. Por otra parte los algoritmos de criptografía asimétrica requieren de operaciones mucho más complejas que con la criptografía simétrica, lo que hace que estas últimas sean más rápidas en ejecución. Desde el día de su creación se ha publicado mucho acerca de este tema reduciendo el tiempo de ejecución tanto en software como en hardware.

Con la aparición de nuevos dispositivos de carácter no general como smartphones, tablets, consolas, etc., las implementaciones en hardware están cobrando una mayor relevancia, ya que estos dispositivos no pueden ejecutar los algoritmos en un tiempo razonable. De esta forma, incluyendo una unidad dedicada a la criptografía, por ejemplo, liberaría de carga a otros

recursos como el procesador y al estar especializado terminaría mucho más rápido que un procesador de carácter general.

Dentro de las implementaciones en hardware de las operaciones modulares, en sistemas más complejos como las curvas elípticas sobre cuerpos primos, es necesario implementar todas las operaciones (suma modular, resta modular, multiplicación modular e inversión). La más costosa y lenta es la multiplicación. Si se mejora la multiplicación modular se aumentaría el rendimiento de las demás operaciones de carácter general que se utilizan para la obtención de la clave. Podemos decir entonces que el producto modular es una operación crítica en la ejecución de los criptosistemas de clave pública que trabajan sobre cuerpos primos, curvas elípticas y de emparejamientos bilineales (criptografía basada en identidad). Por ejemplo para calcular estos últimos se requieren miles de productos modulares con números de longitud grande. Por este motivo, es necesario el diseño de una multiplicación modular que realice dicha operación a altas velocidades.

1.4 Estado del arte

En esta sección no se abordan todos los componentes que se utilizan en el trabajo, sino que se centra principalmente en los pilares sobre los que se basa para la construcción de la construcción del módulo de multiplicación modular final. Estos pilares son: las curvas elípticas, el sumador de Kogge-Stone, la multiplicación de Karatsuba-Ofman y la reducción de Barret.

La criptografía de curvas elípticas fue propuesta en el año 1985 por N.Koblitz en [16] y por V. Miller en [3]. Desde esa fecha han evolucionado bastante pero siempre se han impuesto a otros algoritmos de criptografía gracias a que ofrecen una mayor seguridad con una menor longitud de clave. Desde su proposición este tipo de criptografía ha sido investigada intensivamente para obtener una implementación eficiente, tanto en hardware [19,20] como en software [17,18]. Una gran parte de estos trabajos se centra en la operación más importante o compleja de todas las que envuelven a las ECC, la multiplicación escalar. La idea detrás de esto es simple, mejorando la operación más compleja, se mejora el rendimiento general de todo el criptosistema. El uso de este tipo de criptografía se ha extendido mucho, desde el uso más común que es el uso de estos criptosistemas para generar una librería de uso [22] hasta en tarjetas como

en [21] donde además se discute sobre las ventajas y desventajas del uso de las ECC en este tipo de campos.

Los sumadores son el elemento más crítico en la mayoría de los circuitos digitales. Tal es la importancia, que la investigación de mejora de estos elementos es continua para garantizar un mejor uso del área, de la potencia consumida o del retardo o de la propagación del acarreo. Después de varios diseños como los *"Ripple Carry Adder"* o los *"Carry Skip Adder"* surgen los denominados *"Parallel Prefix Adders"*. Entre ellos destaca un sumador por su gran velocidad. Los sumadores de Kogge-Stone [34] fueron desarrollados en la década de los 70 por los ingenieros Peter M. Kogge y Harold S. Stone. Hasta el momento, no se han desarrollado algoritmos o dispositivos capaces de igualar su velocidad. Por esa razón es una de las opciones más comunes para las operaciones de suma en dispositivos de alto rendimiento en la industria.

Existen multitud de algoritmos de multiplicación: algoritmo de sumas y desplazamientos, algoritmo de Booth, algoritmo de Fibonacci. Pero dentro de estos algoritmos de multiplicación destaca el algoritmo de A. Karatsuba. En [23] se propone el primer algoritmo que baja de la complejidad cuadrática para valores enteros. Los anteriores algoritmos de multiplicación, como el básico de productos y sumas, tiene una complejidad cuadrática. En [10], y con la colaboración de Y.Ofman, se evoluciona el primer algoritmo mencionado de multiplicación para obtener el algoritmo de Karatsuba-Ofman para la multiplicación de enteros que se usa actualmente. Este algoritmo ha sido modificado [11,41] muchas veces con el fin de mejorar la velocidad aumentando la complejidad y el área utilizada. El aumento de área se puede ver tanto en espacio si la implementación se realiza en hardware [25] como en memoria si se realiza en software [24]. Aunque el uso de este algoritmo está presente casi siempre en trabajos relacionados con la criptografía hay muchas otras aplicaciones donde se usa como en [26] donde se explica el uso para una CPU eficiente. Pero muy pocas son las aplicaciones de este algoritmo que no sea con fines criptográficos.

Hasta 1986 las reducciones modulares se hacían por el método naive de sucesivas restas. En ese año, P.D. Barrett introdujo un nuevo algoritmo de reducción en [27]. Este algoritmo reduce considerablemente el número de multiplicaciones y restas frente al algoritmo naive.

Desde entonces se han propuesto numerosas mejoras al algoritmo anterior. Estas mejoras se basan en la pre-computación de los valores necesarios para ello [30], o sin el cálculo de estos valores [28]. También se han hecho muchos estudios comparando la efectividad de este algoritmo frente a la reducción de Montgomery y frente al método naive [29]. El algoritmo de Montgomery fue introducido por P. Montgomery en [31] y consigue una reducción modular en varios pasos usando valores pre-computados.

1.5 Contribución

El objetivo de en este trabajo es, como se menciona anteriormente, el de mejorar la multiplicación modular para aumentar el rendimiento en velocidad. Para ello se han estudiado distintos métodos de suma, multiplicación y reducción para compararlos y escoger el que mejor se adaptaba a las necesidades de velocidad. Al estudiar la aritmética modular se encontró una propiedad que se verá más adelante que se cumple en la aritmética modular y que por el momento no se ha tenido constancia alguna de su uso. Usando esta nueva propiedad de la aritmética modular y haciendo uso del lenguaje de VHDL, se ha implementado sobre FPGA una nueva arquitectura para la multiplicación modular que proporciona unos buenos resultados en términos de velocidad al realizar comparaciones con otros autores.

1.6 Estructura del documento

Este trabajo está organizado en 10 capítulos donde el primero corresponde a la introducción del mismo. En el capítulo 2 se da una visión de las tecnologías utilizadas para el desarrollo del trabajo. Desde la descripción de los elementos utilizados hasta la metodología de uso. En el capítulo 3 se explica toda la base matemática que hay detrás del trabajo. En los capítulos 4, 5 y 6, se hace un estudio de los siguientes elementos: sumadores, multiplicadores y reductores, respectivamente. En ellos se explica el funcionamiento y las propiedades de cada uno para escoger el que mejor se adecue a las necesidades finales de velocidad. En el capítulo 7 se explica el componente de contribución de este trabajo a la aceleración hardware. Además se detalla un estudio como en los anteriores elementos, de su funcionamiento y propiedades. En el capítulo 8 se implementa el multiplicador modular utilizando para ello los componentes descritos en los capítulos anteriores. En el capítulo 9 se muestran todos los resultados experimentales. En

el capítulo 10 se describen las conclusiones finales y el trabajo futuro. En la parte final del trabajo está la bibliografía utilizada, un pequeño apéndice con enlaces y el archivo de figuras.

Capítulo 2 - Tecnologías Hardware

En este capítulo se detallan y se explican las tecnologías usadas para el desarrollo del proyecto. Dentro de este capítulo se explica el funcionamiento de los módulos de hardware reconfigurables o FPGA en los que se integra el proyecto. También se explican los programas de diseño hardware y de las herramientas usadas para el desarrollo del componente final y el lenguaje utilizado para el desarrollo del trabajo (VHDL). Este lenguaje hace de nexo entre los programas de diseño y el dispositivo final, que en este caso son las FPGA.

2.1 FPGA

En los principios de la historia de la informática el tiempo de diseño de circuitos integrados era sustancialmente mayor que el tiempo que se tardaba en la implementación de dicho diseño y de las pruebas. Esta diferencia de tiempos llegaba a ser un problema cuando el diseño era erróneo o tenia un defecto ya que implicaba una pérdida de tiempo y de recursos nada despreciables.

A finales de los años 60, principios de los años 70 se crearon los PLD's que fueron una de las soluciones a este problema de implementación y test. Los Dispositivos Lógicos Programables o PLD's por sus siglas en inglés no eran más que una matriz de puertas lógicas y componentes programables (memorias) conectadas entre sí mediante fusibles. Hoy en día se han desarrollado numerosas familias de PLD. Muchas de estas son más complejas debido a la conexión entre sí que montan. Las más conocidas son las denominadas FPGA's o *"Field Programmable Gate Array"* ya que ofrecen una gran flexibilidad, capacidad y versatilidad con respecto a otros dispositivos como las PLA's *"Programmable Logic Array"* o las PAL's *"Programmable Array Logic"*.

En general, las FPGA's no son dispositivos finales para el diseño, por el contrario, es una plataforma de pruebas donde el circuito es evaluado y depurado. De esta manera, cuando el diseño está listo para su producción en masa, es implementado en los llamados VLSI o *"Very Large Scale Integration"* , que son circuitos integrados de función específica, los cuales son más baratos y rápidos que una FPGA pero mucho menos flexibles. La compañía Xilinx divide sus

modelos de FPGA en distintas familias con base en su tamaño y características, de esta manera dependiendo del presupuesto y los requerimientos, se elige la familia de FPGA adecuada a las necesidades del proyecto. Hay distintas familias como las Spartan (de uso prácticamente académico), Kintex, Artix(de uso principalmente industrial) y Virtex. Otras empresas vendedoras y fabricantes de FPGA's son Atmel, Altera, AMD o Motorola.

Dentro de las FPGA's se encuentra una estructura jerarquizada de componentes unos dentro de otros. En esta jerarquía se encuentran en primer lugar los bloques configurables CLB's o en ingles *"Configurable Logic Blocks"* los cuales están conectados directamente a una matriz de interconexión donde se realizan las conexiones reconfigurables. Dentro de estos CLB's podemos encontrar varias unidades de componentes llamados *"Slices"*(Figura 1). Los "*Slices"* están formados por multiplexores y elementos reconfigurables llamados LUT's o *"Look Up Tables"* que pueden usarse como bloque de memoria o un elemento de lógica. El número de *"Slices"* requeridos en una implementación es considerado una métrica para definir el tamaño del diseño.

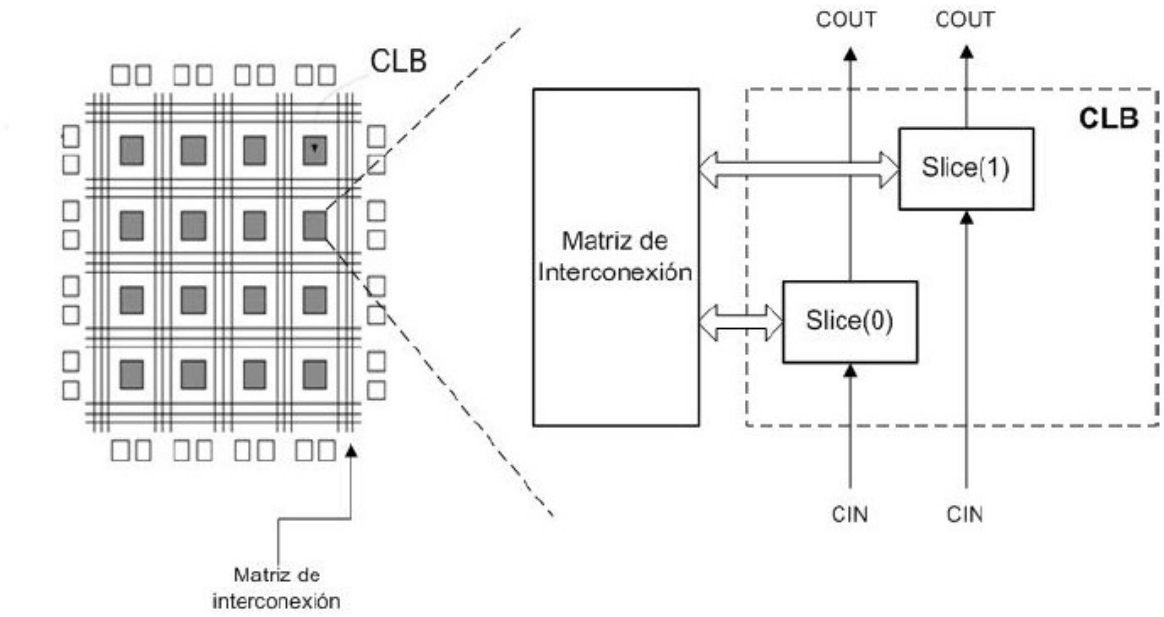


Figura 1: Detalle de un "Slice" en una FPGA

Se ha terminado concluyendo en vista de las características vistas de todas las familias que la familia de uso sea la Virtex5 debido a sus características con respecto a la velocidad y otros recursos que pueden ser utilizados en futuras mejoras para el componente final del trabajo. Además la familia Virtex5 ofrece una gran cantidad de componentes llamados *"DSP48Slices"* que son elementos para procesamiento digital de señales, los cuales son capaces de realizar multiplicaciones asimétricas de 25x18 bits y sumas de 48 bits. Estos componentes trabajan a una frecuencia máxima de 450 Mhz y tienen la capacidad de ser usados como acumuladores, sumadores, multiplicadores y hasta como compuertas lógicas.

2.2 VHDL

Como se mencionó antes los dispositivos como las FPGA's son reconfigurables para adaptarse a varios diseños en un menor tiempo y cantidad de recursos. El puente entre el diseño y el componente es donde hacen su aparición los HDL o *"Hardware Description Language"* que describen el funcionamiento o definen una arquitectura de un componente digital. Hay multitud de lenguajes de descripción, entre los más conocidos Verilog, VHDL y Abel.

El lenguaje de descripción de hardware más utilizado actualmente es VHDL, cuyas siglas provienen de VHSIC *"Very High Speed Integrated Circuit"* y HDL. Fue establecido como estándar por la IEEE *"Institute of Electrical and Electronics Engineers"* en 1987, y hasta el momento lleva dos revisiones, una en 1993 y la otra en el 2002 (Ver [3] del apéndice 1). Para el lenguaje VHDL, cada compañía proporciona un entorno de desarrollo para sus dispositivos, aportando sus propias bibliotecas de componentes y funciones para hacer más sencilla la implementación.

2.3 Metodología de diseño

En el desarrollo de diseño digital hay que tener en cuenta que la metodología es muy diferente a la utilizada en la creación de software, debido a que hay que ser más cuidadoso con los recursos disponibles y tener más claros los resultados esperados. En el desarrollo del software, por ejemplo, la máquina ya tiene una serie de características que no se pueden modificar, por ejemplo el tamaño de palabra. En el desarrollo de hardware generalmente no existen estas restricciones. Siempre que existen tantas libertades, hay que realizar un mejor

análisis acerca de cuales son las mejores opciones. Según lo anterior, el flujo de diseño usado generalmente y definido en [39] es:

- Diseño del circuito y captura en HDL: Codificar el algoritmo de un dispositivo en un lenguaje descriptor.
- Simulación Funcional: Verificar el correcto funcionamiento de un diseño sin llegar a construirlo para ahorrar recursos.
- Síntesis: Transformación del código en el lenguaje descriptor a su equivalente en puertas lógicas y dispositivos lógicos.
- *"Place and route"*: Una síntesis mucho más detallada donde se realiza el análisis de recursos y su velocidad, basándose en diferentes características físicas del dispositivo.
- Análisis del circuito: Primera parte de la integración con la FPGA.
- Programación del dispositivo: Segunda parte de la integración con la FPGA.

2.4 Herramientas

Como se ha mencionado antes, cada compañía fabricante de dispositivos PLD's ofrece un entorno de trabajo y unas herramientas destinadas al desarrollo de componentes. La mayoría ofrecen herramientas para la escritura del código, la síntesis y simulaciones. Esto es debido a que cada compañía junto con sus dispositivos lanza un paquete de desarrollo para los mismos, donde el uso y las herramientas pueden variar dependiendo de la compañía y las disponibilidades de la versión.

Dado que el objetivo principal es construir un dispositivo y sintetizarlo para una FPGA de la familia Virtex5 de Xilinx, es necesario usar la herramienta que ofrece esta misma compañía para un síntesis adecuada. Las herramientas utilizadas son las siguientes:

- ISim: Es el simulador que ofrece esta versión ISE. Es una versión mejorada de los antiguos simuladores (ISE Simulator) pero por el contrario hace que su depuración sea más compleja tanto en desarrollo como en la visualización. Antes de esta herramienta el

entorno usaba otro simulador llamado ModelSim que sino era tan eficiente, era mucho más rápido y mejor en la depuración.

- ISE 14.2: Es la herramienta de desarrollo y entorno de trabajo proporcionado por la empresa Xilinx. Las principales funciones son las de escritura de código, síntesis del mismo, generación de resultados o informes, gráficas, generación de máquinas de estados, etc.

Capítulo 3 - Teoría matemática

En este capítulo se describe la teoría matemática que está detrás de todo el trabajo. El fin de este capítulo es dar un soporte teórico a todo lo que se hace en el proyecto. Los temas que se explican son la aritmética modular, por la cual sin ella no existiría este proyecto, los cuerpos finitos y las curvas elípticas.

3.1 Aritmética modular

La aritmética modular es una aritmética distinta a la aritmética clásica donde los valores que se utilizan están en el rango de [0..*m*-1]. Fue introducida por Carl Friedrich Gauss en [7]. Esto repercute en que todo número N dentro de los Naturales está representado por uno de los números en dicho rango. La operación matemática se suele expresar de la siguiente forma:

$$A \bmod m = B \tag{1}$$

Entonces si tenemos un número *A*, el representante en el rango *M* al que llamaremos *B*, *X* es el resultado de la división entera entre *A* y *M*, se cumple que:

$$A/(m-1) = X \cdot (m-1) + B \tag{2}$$

Una congruencia es un término usado en la teoría de números para designar que dos números enteros tienen el mismo resto al dividirlos por un número natural distinto de 0. Esta definición quiere decir, que dos números *x* e *y* son congruentes si tienen el mismo representante módulo M, es decir, cumplen con (2) obteniendo el mismo valor de B. El símbolo para denotar una congruencia es "≡". Con las congruencias podemos establecer un conjunto de operaciones aritméticas, como:

Siendo *a, b, c, d* $\in$ Z y *m* $\in$ N, tales que *a* ≡ *b (mod (m))* y *c* ≡ *d (mod (m))*.

Entonces:

$$a + c \equiv (b + d) \bmod (m) \tag{3}$$

$$a \cdot c \equiv b \cdot c \,(mod\,(m)) \tag{4}$$

A partir de (2), (3) y (4) se pueden definir las siguientes propiedades aritméticas para las sumas y productos de congruencias:

Asociativa: $a+(b+c)(mod(m))=(a+b)+c(mod(m))$

Elemento neutro: Existe un elemento $0 \in Z_m, tal\ que, a+0(mod(m))=a(mod(m))$

Elemento opuesto: Existe un elemento $b \in Z_m, tal\ que, a+b=0$

Conmutativa: $a+b(mod(m))=b+a(mod(m))$

Cancelativa: $a \cdot c \equiv b \cdot c(mod(m))\ y\ mcd(m, c)=1, entonces, a \equiv b(mod(m))$

Asociativa: $a \cdot (b \cdot c)(mod(m))=(a \cdot b) \cdot c(mod(m))$

Elemento neutro: Existe un elemento $1 \in Z_m, tal\ que\ a \cdot 1(mod(m))=a(mod(m))$

Elemento inverso: Existe un elemento

$$a-1 \in Z_m\ para\ todo\ a \in Z_m\ con\ mcd(a, m)=1, tal\ que\ a \cdot a-1=1$$

La aritmética modular está presente en muchas áreas de las matemáticas: En la teoría de números [8], en el álgebra abstracto, teoría de grupos. Por tanto también está presente en las áreas donde se utilizan dichas teorías y álgebras como por ejemplo la Física y la Química pero también podemos verla utilizada en otras áreas menos convencionales como la música o el arte visual de forma sutil o directamente.

3.2 Cuerpos finitos

Un cuerpo finito, es un cuerpo que tiene un número finito de números. Un cuerpo se asemeja a un sistema algebraico con dos leyes de composición interna, llamadas tradicionalmente: adición y multiplicación. De modo que respecto de la adición es un grupo abeliano (la operación de adición entre dos números da un tercero dentro del conjunto y es conmutativo) con elemento neutro llamado *cero*. Los demás elementos, sin el cero, también forman un grupo multiplicativo con elemento unitario, denotado con 1, respecto de la segunda operación. Los cuerpos finitos que interesan en este trabajo son los cuerpos primos. Los cuerpos primos F_p son aquellos donde los números y las operaciones dentro del cuerpo son módulo p siendo p un primo.

3.3 Criptografía de curvas elípticas

La criptografía de curva elípticas fue propuesta en 1985 por Neal Koblitz y Victor Miller. Es un enfoque nuevo de la criptografía de clave pública basado en la estructura de las curvas elípticas sobre cuerpos primos. Esto quiere decir que utiliza una clave pública y otra privada relacionada directamente con puntos de una curva elíptica. La criptografía de clave pública se basa principalmente en ese hecho, dos claves estrechamente ligadas pero que conociendo una clave es matemáticamente muy complejo obtener la otra. Este tipo de criptografía se beneficia del problema del logaritmo discreto del que hablaremos después. Una curva elíptica es una curva plana que satisface este sistema de ecuaciones para todo valor del cuerpo donde este la curva (incluido el infinito):

$$y^2 = x^3 + Ax + B \tag{5}$$

$$4A^3 + 27B^2 \neq 0 \tag{6}$$

Si se cambian los valores de las constantes de A y B obtendremos una curva elíptica distinta cada vez. Los valores de A y B pueden ser enteros, reales, racionales, complejos. etc... Por ejemplo, en la Figura 1 se muestran dos curvas en el cuerpo de los enteros que responden a las ecuaciones $y^2 = x^3 - 2x + 3$ y $y^2 = x^3 - 3x$

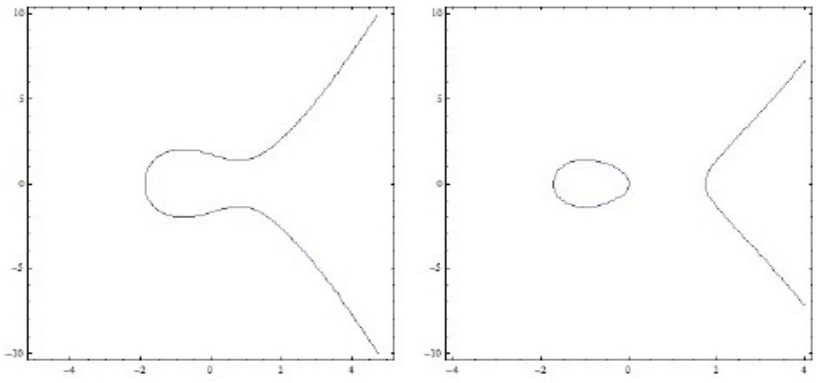


Figura 2: Curvas elípticas

La clave pública es un punto de la curva y la clave privada es un número aleatorio. La clave pública se obtiene por la multiplicación de este número aleatorio (la clave privada) por un generador de puntos G de la curva. Este generador es una ecuación que recibiendo como entradas dos valores cualesquiera y conociendo los parámetros de la curva (A y B) genera un punto de la curva.

Una de las propiedades que hay que destacar de estas curvas es el punto negativo. El punto negativo de un punto $P=(x, y)$ en este tipo de curvas responde a la reflexión de la misma en el eje de las X, es decir:

$$-P=(x, -y) \tag{7}$$

Hay muchas implementaciones tanto en software [1] como en hardware [2] de este tipo de curvas para obtener mejores resultados a la hora de ocultar información. Los beneficios de usar una implementación software es que son muy polivalentes y pueden ser adaptadas rápidamente para múltiples situaciones o curvas. En cambio pueden ser muy lentas. Por otra parte, las implementaciones en hardware pueden ser extremadamente rápidas, pero las puede hacer más rígidas frente a cambios como veremos más adelante.

Son varios los beneficios que otorga usar la criptografía de curvas elípticas, entre ellas quizá la más destacada sea la reducción en anchura de la clave. Si se elige bien la curva, el mejor método que se conoce para romper el código es exponencial mientras que los sistemas criptográficos usan algoritmos sub-exponenciales. También y gracias a esto último claves más pequeñas pueden ser igual de seguras que claves más grandes en otro criptosistema. Por ejemplo, una clave RSA de 1024 bits es igual de segura que una clave de 160 bits usando la criptografía de curvas elípticas. Hay muchos estudios, como por ejemplo [36], donde se verifica que la criptografía basada en curvas elípticas es más eficiente en muchos aspectos que RSA (su inmediato competidor). Se concluye de dichos estudios que RSA se queda atrás en comparación.

Se puede ver en la siguiente tabla que la criptografía de curvas elípticas utiliza mucho menos el canal para transmitir la información necesaria frente a RSA. Esta mejora incluye el

paso de parámetros, por parte de la ECC ya que el algoritmo RSA no necesita, lo cual hace que sea más eficiente en otros ámbitos como el uso del canal.

	ECC sobre primos	**RSA 1024-bit con módulo=216+1**
Parámetros (en bits)	(4 * 160) + 1 = 641	0
Clave pública (en bits)	160 + 1 = 161	1024 + 17 = 1041
Clave privada (en bits)	160 (801 con los parámetros)	2048 (2560 con la información CRT)

Tabla 1: Comparación ECC vs RSA

3.2.1 Operaciones

- *"Point addition"*

Obtenido un punto P en una curva elíptica, es posible sumarle otro punto Q (también en la curva) que de como resultado un punto L que sigue estando definido en la misma curva. Esta operación es posible siempre que $Q \neq -P$ ya que si fuera así tendríamos P + *(-P)* y el resultado de esta operación es un punto en el infinito. Las ecuaciones que definen la suma por componentes dados un punto $P=(x_p, y_p)$ y $Q=(x_q, y_q)$ podemos definir $L = P + Q$, donde $L=(x_l, y_l)$ como:

$$x_l = s^2 - x_p - x_q \quad (8)$$

$$y_l = -y_p + s(x_p - x_l) \quad (9)$$

$$s = (y_p - y_q)/(x_p - x_q) \quad (10)$$

Es posible ver que si $Q = P$ la suma da como resultado *2P* que es lo que se denominará más adelante como *"Point doubling"*. Por último este tipo de suma cumple la propiedad conmutativa de tal forma que: $P + Q = Q + P$

- *"Point doubling"*

Obtenido un punto P en una curva elíptica, es posible sumarle de nuevo P que de como resultado otro punto en la misma curva. Esto es posible si la coordenada de las y en el punto P no

es cero. De ser así el doble tendría la solución en el infinito. Las ecuaciones que definen el doble del punto por componentes dados un punto $P=(x_p, y_p)$ donde $y_p \neq 0$ podemos definir $L=2P$, donde $L=(x_l, y_l)$ y A es un parámetro de la curva como:

$$x_l = s^2 - 2x_p \quad (11)$$

$$y_l = -y_p + s(x_p - x_l) \quad (12)$$

$$s = (sx_p^2 + A)/(2y_p) \quad (13)$$

- *"Point multiplication"*

Obtenido un punto P en una curva elíptica, es posible multiplicar este punto por un entero k para obtener otro punto Q en la misma curva , es decir, $Q = kP$. Esta operación a su vez está formada por dos operaciones más (*"Point addition"* y *"Point doubling"*). Por ejemplo, si se tiene un punto P y se quiere multiplicar multiplicar 11 veces ($k = 11$) se define la multiplicación escalar como:

$$Q = kP = 11P = 2(2(2P) + P) + P \quad (14)$$

El método descrito en (14) se denomina en ingles *"Double and Add"*. Hay muchas más formas de conseguir la multiplicación escalar como la NAF (Non Adjacent Form) o la wNAF (windowed NAF) pero que no se explicarán aquí.

3.2.2 Problema del logaritmo discreto

La mejora en la anchura de la clave viene dada principalmente por esta característica o problema. La seguridad de una clave usando la criptografía de curvas elípticas pasa principalmente por este problema. Si se tiene un punto P cualquiera de la curva y un punto Q, tal que $Q = k \cdot P$, donde k es un entero. Es computacionalmente intratable obtener k (si es suficientemente grande). Se dice que k es el logaritmo discreto de Q en base P. Esta dificultad es posible ya que la principal operación de las curvas elípticas es la multiplicación escalar.

3.3 Curvas elípticas sobre cuerpos primos

Para un sistema automático y digital, la multiplicación escalar puede ser un problema ya que estos sistemas digitales pueden ser inexactos en la representación de sistemas continuos y por el rendondeo pueden inducir a error, además de ser lentos. Para un sistema de criptografía la velocidad y la eficiencia son aspectos importantes a tener en cuenta. Por eso se implementaron sobre dos cuerpos finitos donde operar con estas curvas. Estos cuerpos son el cuerpo de los primos F_p del que se hablo anteriormente y el cuerpo algebraico o binario F_m. De esta forma la curva quedaría definida de diferente forma a (5) y (6). Ahora se tiene como ecuaciones de la curva:

$$y^2 \bmod m = (x^3 + Ax + B) \bmod m \quad (15)$$

$$(A^3 + 27\mathrm{B}^2) \bmod m \neq 0 \quad (16)$$

Ahora la curva está definida entre [0 – m-1]. El primo m debería ser un número lo suficientemente grande para que el sistema sea seguro. Varios estudios indican que los primos para que el sistema sea lo suficientemente seguro tienen una longitud de entre 112 y 521 bits [40]. Dado que no se está usando números reales la curva no está definida en todo R, y por tanto, hay que re-definir ligeramente las operaciones "básicas" de multiplicación, suma y doble, para que funcionen correctamente para módulo m.

- Suma

$$x_l = (s^2 - x_p - x_q) \bmod m \quad (17)$$

$$y_l = (-y_p + s(x_p - x_l)) \bmod m \quad (18)$$

$$s = ((y_p - y_q)/(x_p - x_q)) \bmod m \quad (19)$$

- Doble

$$x_l = (s^2 - 2\mathrm{x}_p) \bmod m \quad (20)$$

$$y_l = (-y_p + s(x_p - x_l)) \bmod m \quad (21)$$

$$s = ((sx_p^2 + A)/(2\mathrm{y}_p)) \bmod m \quad (22)$$

Capítulo 4 - Sumadores

En este capítulo se detallan los componentes más básicos del proyecto, los sumadores. Dentro de las diferentes versiones y algoritmos de un sumador se han escogido 3 candidatos para hacer un estudio de las diferentes funcionalidades y características para ver cual se adecua a nuestro objetivo final de aumento de velocidad. Los 3 algoritmos de suma candidatos son: El tradicional o sumador paralelo con acarreo en serie *"Ripple-Carry Adder"*, el denominado como *"Carry LookAhead"* o de aceleración del acarreo y el sumador de Kogge-Stone.

4.1 Elementos

El sumador paralelo con acarreo en serie (RCA) es el primer sumador que se aprende cuando se estudia diseño digital. Conceptualmente es el más sencillo y requiere de un elemento o celda adicional denominado como sumador completo. Esta celda se encarga de sumar un bit de cada entrada con un posible acarreo para dar como resultado un bit de salida y otro de acarreo para la siguiente celda de suma. Los sumadores completos se construyen en secuencia una detrás de la otra conectando el acarreo de salida con el acarreo de entrada de la siguiente. El problema que presentan es que para conocer la ultima suma de bits han tenido previamente que calcularse las anteriores sumas para saber el acarreo de entrada de esta última.

El sumador de *"Carry LookAhead"* (CLA) o de aceleración del acarreo soluciona el problema que tienen los anteriores sumadores. Estos sumadores ofrecen una serie de operaciones adicionales que generan el acarreo para cada etapa sin tener que esperar a que llegue de etapas anteriores. Esto lo consigue con bloques que implementan funciones que determinan la generación y la propagación de cada etapa y otro bloque de anticipación que las calcula en función del primer acarreo.

Los sumadores de KoggeStone [34,35] son una extensión de los anteriores. Esta extensión se denominan inglés como PPA o *"Parallel Prefix Adders"*. Se trata de sumadores donde el acarreo se obtiene mediante una operación asociativa que permite utilizar técnicas similares a las de una computación en paralelo. Los acarreos se van resolviendo por niveles y dan lugar a un árbol de computación de acarreos.

4.2 Algoritmos

Cada uno de los sumadores, aunque similares, tienen distintos algoritmos a la hora de calcular el acarreo para la suma del siguiente dígito. También lo tienen a la hora de calcular la suma en función del acarreo y de las entradas. A continuación se pueden ver las diferentes formas de calcularlo para cada sumador y el algoritmo del sumador completo.

4.2.1 Sumador completo

El sumador completo es el componente principal del sumador sumador paralelo con acarreo en serie. Esta celda implementa la tabla de verdad donde se tienen como entradas los bits *A, B* y el acarreo de entrada y como salidas el resultado de la suma y el acarreo de salida. El sumador completo se puede implementar mediante unas pocas operaciones lógicas de la siguiente forma:

```
AcarreoSalida = ((A · B) OR ((A ⊕ B) · Acarreo Entrada));
Suma = ((A ⊕ B) ⊕ Acarreo Entrada);
```

4.2.2 Sumador paralelo con acarreo en serie

El sumador paralelo con acarreo en serie hace uso del componente anterior, el sumador completo. Hace una disposición de estos sumadores de tal forma que se conecta el acarreo de entrada de uno con el acarreo de salida del anterior, creándose una fila de estos componentes. Al utilizarse sumadores completos el primer componente al no tener a quien conectarse por su acarreo de entrada se le coloca el valor 0 o se deja para que otro componente le coloque el valor como entrada. El algoritmo de un sumador paralelo con acarreo en serie que permite el cálculo de los *k* bits del resultado de la suma y el acarreo de salida C_{out} en función de las entradas *A(k-1,k-2,..,0)* y *B(k-1,k-2,..,0)* y el acarreo de entrada C_{in} es el siguiente:

ENTRADAS:

A(k-1,k-2,..,0),B(k-1,k-2,..,0), C_{in}

SALIDAS:

Suma(k-1,k-2,..,0), C_{out}

CUERPO:

```
Suma(0),Acarreo(0) = SumadorCompleto( A(0) , B(0) , Cin );
for i = 1 to k-2 do
      Suma(i),Acarreo(i) = SumadorCompleto( A(i) , B(i) , Acarreo(i-1) );
end for;
Suma(k-1), Cout = SumadorCompleto( A(k-1) , B(k-1) , Acarreo(k-1) );
```

Aunque la carga de las entradas se haga en paralelo, el cálculo del último bit de la suma y el acarreo de salida, ha de esperar hasta que no se calcule el acarreo de salida del componente anterior.

4.2.3 Sumador con aceleración del acarreo (CLA)

El sumador con aceleración del acarreo es más complejo que el anterior. Para construir uno se necesita de un componente adicional que es el denominado CLA o *"Carry LookAhead"*. Este componente se encarga de calcular mediante 2 nuevas funciones los acarreos. Los sumadores completos se reemplazan por bloques con dos nuevas funciones aparte de la suma. Estas dos nuevas funciones son P y G (*"Propagate"* y *"Generate"*).

$$P_i = (A_i \oplus B_i)$$

$$G_i = (A_i \cdot B_i)$$

$$Suma_i = (C_i \oplus P_i)$$

Cada función calcula si el acarreo se propaga o se genera en función de las entradas. El algoritmo de un sumador con aceleración del acarreo que permite el cálculo de los *k* bits del resultado de la suma y el acarreo de salida C_{out} en función de las entradas *A(k-1,k-2,..,0)* y *B(k-1,k-2,..,0)* y el acarreo de entrada C_{in} es el siguiente:

ENTRADAS:

$A(k-1,k-2,..,0),B(k-1,k-2,..,0), C_{in}$

SALIDAS:

$Suma(k-1,k-2,..,0), C_{out}$

```
CUERPO:
for i = 0 to k-1 do
      G(i) = A(i)  XOR  B(i);
end for;
for i = 0 to k-1 do
      P(i) = A(i) AND B(i);
end for;
Acarreo(0) = Cin;
for i = 1 to k+1 do
      Acarreo(i) <= G(i-1) or (P(i-1) and Acarreo(i-1));
end for;
Cout = Acarreo(k+1);
for i = 1 to k-1 do
      Suma(i) = = ( P(i) XOR Acarreo(i) );
end for;
Suma(k) = ( P(k) XOR Acarreo(k) );
```

Este componente se puede escalar de tal forma que cuando el ancho de la entrada es grande se puedan ir acelerando los acarreos por varios niveles. Esto se consigue acelerando la aceleración con otro bloque CLA. Para 4 bit el cálculo de los acarreo es el siguiente.

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1$$

$$C_3 = G_2 + G_1 \cdot P_2 + G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \cdot P_2$$

$$C_4 = G_3 + G_2 \cdot P_3 + G_1 + G_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3$$

4.2.4 Sumador de Kogge-Stone

El sumador de Kogge-Stone como ya se menciono antes es un sumador que pertenece a la categoría de los *"Carry LookAhead Adders"* o sumadores con aceleración del acarreo, que reduce el tiempo requerido para calcular los bits de acarreo. Para ello se hace por medio de varios niveles, concretamente *log(n-1)* donde *n* es le número de bits de la entrada. Se van calculando los sucesivos acarreos y además de valores intermedios de estos. En la primera etapa, siendo *A* y *B* dos entradas de n-bits se calculan las siguientes señales al igual que en el sumador anterior:

$$P_i = A_i \oplus B_i$$

$$G_i = A_i \cdot B_i$$

$$Suma_i = Pi \oplus C_i$$

Para las siguientes etapas, siendo (P', G') y (P'', G'') dos pares de señales de (propagación,generación), con X' más significativa que X'', se calcula la siguiente operación:

$$(P, G) = (P' \cdot P'', G' + G'' \cdot P')$$

Después de calcular los acarreos, la suma no es más que una concatenación de operaciones XOR con A y B y los acarreos calculados o con *Pi*. El algoritmo de un sumador de Kogge-Stone que permite el cálculo de los *k* bits del resultado de la suma y el acarreo de salida C_{out} en función de las entradas *A(k-1,k-2,..,0)* y *B(k-1,k-2,..,0)* y el acarreo de entrada C_{in} es el siguiente:

ENTRADAS:

A(k-1,k-2,..,0),B(k-1,k-2,..,0), C_{in}

SALIDAS:

Suma(k-1,k-2,..,0), C_{out}

CUERPO:

```
// Se generan el primer elemento del primer nivel de acarreos tanto generadores como
propagadores para generar los demás del nivel
matrizProp(0)(0) = (A(0) XOR B(0));
matrizGen(0)(0) = (A(0) AND B(0)) OR (A(0) AND Cin) OR (B(0) AND Cin);
Suma(0) = matrizProp(0)(0) XOR Cin;
// Se inicializan los bits de propagación y de generación del acarreo del nivel 0
for i = 1 to k do
      matrizProp(0)(i) = A(i) XOR B(i);
      matrizGen(0)(i) = A(i) AND B(i);
end for;
LOG = Log2(k) // Se calcula el número de niveles
for i = 1 to LOG do
      for j = 0 to LOG do
            if (j < 2*(i-1))  // se dejan los acarreos como están
```

```
                matrizProp(i)(j) = matrizProp(i-1)(j);
                matrizGen(i)(j) = matrizGen(i-1)(j);
            end if;

            if (j >= 2*(i-1))//Se generan nuevos acarreos en función del nivel i-1
                matrizProp(i)(j) =  matrizProp(i-1)(j) AND
                                    matrizProp(i-1)(j-2*(i-1));
                matrizGen(i)(j) = matrizGen(i-1)(j) OR
                                  ( matrizGen(i-1)(j-2*(i-1)) AND
                                    matrizProp(i-1)(j) );
            end if;
        end for;
end for;
// Se consiguen los verdaderos acarreos
for i = 0 to k do
        Acarreos(i) = matrizGen(LOG)(i);
end for;

Cout <= Acarreos(k);
// Se realiza la suma
for i = 1 to k do
        Suma(i) = A(i) XOR B(i) XOR Acarreos(i-1);
end for;
```

Este componente es el más complejo de todos. Hay varios niveles que van generando sucesivas veces el valor del acarreo. En el nivel 0 se calcula igual que en algoritmo anterior. En los demás niveles se mira si hay que copiar lo anterior, es decir, si *j < 2*(i-1)*, y si no hay que generar uno nuevo para que en el siguiente nivel se pueda utilizar para calcular los demás. Para 4 bits el cálculo de la suma es el siguiente:

$$Suma_0 = A_0 \oplus B_0 \oplus C_{in}$$

$$Suma_1 = (A_1 \oplus B_1) \oplus (A_0 \cdot B_0)$$

$$Suma_2 = (A_2 \oplus B_2) \oplus ((A_1 \oplus B_1) \cdot (A_0 \cdot B_0) + (A_1 \cdot B_1))$$

$$Suma_3 = (A_3 \oplus B_3) \oplus (((A_2 \oplus B_2) \oplus (A_1 \oplus B_1) \cdot (A_0 \cdot B_0)) + ((A_2 \oplus B_2) \cdot (A_1 \cdot B_1) + (A_2 \cdot B_2))$$

4.3 Arquitecturas

Como se ha visto anteriormente, los algoritmos de cada sumador y del sumador completo o la celda de la suma se han implementado en hardware para más tarde ver el rendimiento de cada componente a distintos anchos de entrada. En ningún caso se ha cambiado el algoritmo inicial y se han implementado de la misma forma que se vieron anteriormente.

4.3.1 Celda sumador completo

El sumador completo se ha implementado mediante puertas lógicas AND, OR, XOR y NOT. El retardo total del sumador completo o de la celda es de 2 puertas XOR. La implementación es sencilla y carece de complejidad alguna. Se puede hacer un diseño distinto anidando dos semisumadores, de manera que, la salida "Suma" del primer semisumador se conecte a una de las entradas del segundo semisumador, la entrada C_{in} se conecte con la otra entrada del semisumador, las salidas de acarreo se conectan a una puerta OR para proporcionar la salida del acarreo total de la suma (C_{out}) y la señal Suma del segundo semisumador se queda como resultado total de la operación. Se denomina como semisumador a un sumador como el anterior pero sin la variable de C_{in}.

4.3.2 Sumador paralelo con acarreo en serie

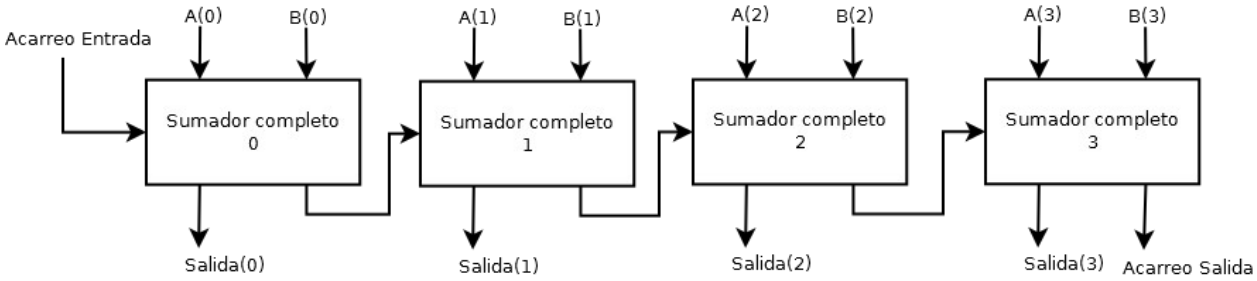


Figura 3: Detalle de un sumador paralelo con acarreo en serie

En la figura 3 se puede apreciar la arquitectura interna de un sumador de 4 bits. Se puede apreciar, como, sin que termine uno de los sumadores el siguiente no puede dar un resultado válido. Este comportamiento hace que el sistema, aun simple, tenga que esperar una cantidad *n* veces el tiempo de respuesta de un sumador para dar un resultado válido. Una mejora implementada al sistema consistió en cambiar el primero de los sumadores completos por un

semisumador. Esto eliminaría la posibilidad de insertar un acarreo de entrada pero agiliza el primer acarreo.

4.3.3 Sumador con aceleración del acarreo

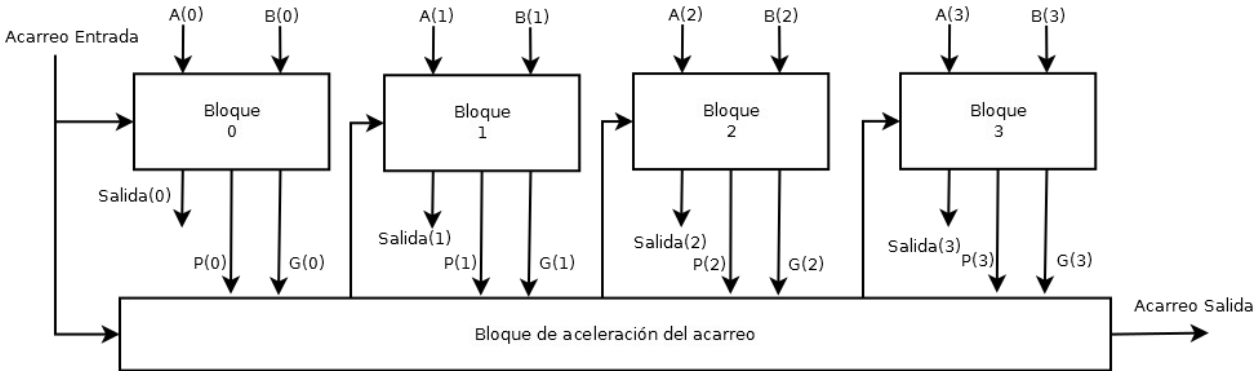


Figura 4: Detalle de un sumador CLA

El sumador con aceleración del acarreo es un componente que consta de tres partes, la primera que es la generación de los bits de propagación y generación (P_i y G_i respectivamente) que están dentro del $Bloque_i$. La segunda parte es el cálculo de los acarreos Cin_i dentro del bloque de aceleración del acarreo y por último el cálculo de la salida Si con el valor de Cin_i nuevo.

Aunque no se puede apreciar tan bien, el retardo de todo el sistema viene dado principalmente por la segunda parte. Para un sumador de 4 bits solo es necesario un nivel de aceleración pero es posible anidar bloques de aceleración si calculamos la función P y G del bloque acelerador. Para un CLA de 4 bits el cálculo de P y G viene dado por:

$$P = P_3 \cdot P_2 \cdot P_1 \cdot P_0$$

$$G = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

De esta forma se puede conectar a otro nivel más grande de aceleración bloques de 4 bits para aumentar la velocidad del cálculo del último acarreo.

4.3.4 Sumador de Kogge-Stone

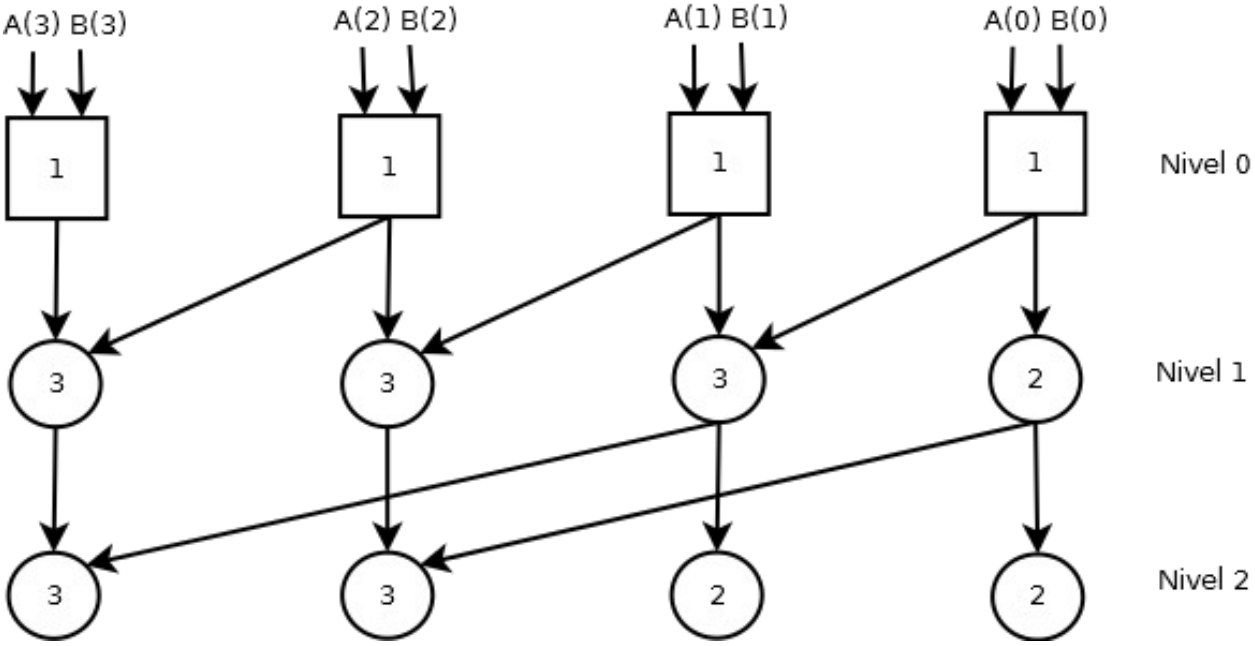


Figura 5: Detalle de la ruta del acarreo en un sumador de Kogge-Stone de 4 bits

El acarreo en el sumador de Kogge-Stone tiene que calcularse en varios niveles, en concreto $log_2(n)$, donde n es el número de bits de las entradas. Los acarreos se propagan de la forma que se ve en la figura 5 a 3 módulos distintos. Cada uno de ellos genera un nuevo valor de P y de G para pasarlo al siguiente nivel. El problema del algoritmo de Kogge-Stone es la cantidad de cableado que se necesita para todas las interconexiones entre nodos de cada nivel. Esto que por una parte hace el diseño más rápido, hace que necesite más área que otros algoritmos. Otras versiones como la de Brent–Kung [37] hacen el sistema más lento pero utilizando mucha menos área. Este sumador de Kogge-Stone es el que tiene menos latencia de todos (ver resultados en el Capítulo 9) y por esa razón es el escogido para usarse en el proyecto como sumador.

Capítulo 5 - Multiplicadores

En este capítulo se detallan los multiplicadores candidatos a utilizar en el proyecto. Los multiplicadores están presentes en las reducciones ya que contienen en sus algoritmos diversas multiplicaciones para calcular de forma rápida el valor representante. En este proyecto tienen gran importancia pero también lo tienen en otras aplicaciones como cálculos vectoriales o en transformaciones de polígonos en tiempo real. Dentro de las diferentes versiones y algoritmos de un multiplicador se han escogido 3 para hacer un estudio de las diferentes funcionalidades y características para ver cual se adecua a nuestro objetivo final de aumento de velocidad. Los 3 algoritmos de multiplicación candidatos son: El tradicional a base de sumas y desplazamientos, el algoritmo de Karatsuba-Ofman y el algoritmo de Booth.

5.1 Elementos

El multiplicador a base de sumas y desplazamientos es uno de los primeros algoritmos que se aprenden en diseño digital. Este algoritmo, mejora en la velocidad de las operaciones al algoritmo básico de la escuela que consiste en multiplicaciones parciales y una suma de *n* elementos. El multiplicador a base de sumas y desplazamientos consiste en ir mirando los bits del multiplicador y dependiendo del valor realizar una acción u otra. Este algoritmo reduce la cantidad de operaciones a sumas (ya implementadas) y desplazamientos de un bit a la izquierda.

La multiplicación de Karatsuba-Ofman [10] permite multiplicar números con un coste mucho menor que por el algoritmo anterior. Está basado principalmente en la idea del "Divide y vencerás" muchas veces utilizada en informática. Hay varias versiones de este algoritmo desarrolladas en software [12] y en hardware [13] que permiten ver mejor el uso de la estrategia de "Divide y vencerás" y generalizaciones para implementaciones [14] que pueden servir como apoyo para futuros trabajos.

La multiplicación de Booth [38] es uno de los algoritmos más utilizados ya que está presente prácticamente en la gran mayoría de las calculadoras de mesa. Este algoritmo aprovecha la velocidad a la hora de desplazar frente a la velocidad de una suma o resta. Una particularidad de este algoritmo es que trabaja con las entradas en complemento a 2 y que utiliza un bit extra.

Este bit se utiliza como auxiliar para ir comparando cadenas de dos bits en las posición menos significativa del multiplicador y en la anterior (bit extra). Dependiendo del valor de esos bits se realiza una acción u otra. En este aspecto es muy parecido al primer algoritmo a base de sumas y desplazamientos.

5.2 Algoritmos

Cada uno de los multiplicadores afronta la operación de diversas formas. Mientras unos utilizan bits de un operando y según su valor actúan de una manera u otra, otros dividen el problema para hacerlo más sencillo. A continuación se pueden ver las diferentes formas de calcularlo para cada multiplicador.

5.2.1 Algoritmo de sumas y desplazamientos

El algoritmo básico realiza *n* multiplicaciones de un bit y una suma de *n* elementos para realizarse, con lo que resulta demasiado costoso. El sumador de sumas y desplazamientos se basa principalmente en el ahorro de operaciones de una multiplicación con el algoritmo básico. Además necesita de la creación de un registro auxiliar el doble de grande que las entradas donde ir sumando en los bits menos significativos el valor del multiplicador. La idea es iterar en los bits del multiplicador y realizar una acción dependiendo del valor del bit *i* en el registro auxiliar antes mencionado. Cuando el bit *i* de la iteración del multiplicador es 1 el contenido de este registro se le suma el valor del multiplicador. Si el bit es 0 no se realiza ninguna operación. Después se desplaza un bit a la izquierda dicho registro y se vuelve a iterar con $i = i+1$. El algoritmo termina de iterar cuando el valor de *i* alcanza un valor superior al número de bits del multiplicando. Este algoritmo es el más sencillo de los 3. El algoritmo de un multiplicador a base de sumas y desplazamientos que permite el cálculo de los 2k bits del resultado del producto en función de las entradas A(k-1,k-2,..,0) y B(k-1,k-2,..,0) es el siguiente:

```
ENTRADAS:
      A(k-1,k-2,..,0),B(k-1,k-2,..,0)
SALIDAS:
      Producto(2k-1,(2k-2,..,0)
```

```
CUERPO:
sumaAuxiliar = 0;
for i = 0 to k do
      if (B(i) == '1')
            sumaAuxiliar = A + sumaAuxiliar;
      end if;
      sumaAuxuliar = desplazamientoIzquierda(sumaAuxiliar);
end for;
Producto = SumaAuxiliar;
```

Este algoritmo es increíblemente sencillo y fácil de implementar. Además a primera vista ocupa muy poco espacio en área. Sin embargo necesita de una cantidad de ciclos igual al número de bits de la entrada.

5.2.2 Algoritmo de Karatsuba-Offman

El algoritmo de Karatusuba-Ofman se basa principalmente en la división de los elementos de la multiplicación en 2 partes más pequeñas y de igual dimensión. En este algoritmo, para entradas de n bits, se realizan cuatro multiplicaciones de $n/2$ dígitos, 2 operaciones de desplazamiento a izquierda, 1 suma de n dígitos y dos sumas de $2n$ dígitos. A continuación se muestra el procedimiento de división de una entrada A en el algoritmo de Karatsuba-Ofman.

$$A = A_1 \cdot r_m + A_0 \tag{23}$$

Siendo r la base de representación del número A y m el número de dígitos de la parte más significativa o A_1. El desarrollo de la multiplicación siendo A y B entradas de n bits siendo $n=2m$ es el siguiente:

$$A \cdot B = (A_1 \cdot r^m + A_0) \cdot (B_1 \cdot r^m + B_0) \tag{24}$$

$$A \cdot B = D_2 \cdot r^{(2 \cdot m)} + D_1 \cdot r^m + D_0$$

$$D_2 = A_1 \cdot B_1$$

$$D_0 = A_0 \cdot B_0$$

$$D_1 = (A_1 \cdot B_0) + (B_1 \cdot A_0)$$

Con ello conseguimos una multiplicación con un coste no mayor a 3 $log_2\ n$. Este algoritmo ha recibido algunas revisiones como el conocido algoritmo de Toom-cook [11] ó Toom-3 para multiplicar enteros grandes de manera que divide las entradas en *k* partes, y no en dos como el original, o como el algoritmo de Schönhage-Strassen [41] pero todos con la misma premisa de división de las entradas. El algoritmo de un multiplicador según el algoritmo de Karatsuba-Ofman que permite el cálculo de los *2k* bits del resultado del producto en función de las entradas *A(k-1,k-2,..,0)* y *B(k-1,k-2,..,0)* es el siguiente:

```
ENTRADAS:
      A(k-1,k-2,..,0),B(k-1,k-2,..,0)
SALIDAS:
      Producto(2k-1,(2k-2,..,0)
CUERPO:
Aalto,Abajo = split(A) // divide el número A en dos partes de igual longitud
Balto,Bbajo = split(B) // divide el número B en dos partes de igual longitud
D2 = Aalto * Balto;
D0 = Abajo * Bbajo;
D1 = ((Balto*Abajo) + (Aalto*Bbajo));
Producto = D0 + D1*exp(10,k/2) + D2*exp(10,k);
```

Una de las características que hace de este algoritmo interesante en cuanto al tiempo es que no es iterativo sino que divide el problema en dos más pequeños para después sumarlos. Por tanto se puede generar una versión del mismo combinacional. A continuación un ejemplo en binario con los valores del anterior algoritmo para su mejor compresión ($A = 5\ (0101)_2$ y $B = 7\ (0111)_2$).

1) Obtenemos las partes correspondientes de los datos

$a_1 = 01\ \ a_0 = 01\ \ b_1 = 01\ \ b_0 = 11$

2) Obtenemos el producto de las partes

$x_0 = a_0 \cdot b_0 = 01 \cdot 11 = 0011$

$x_1 = a_0 \cdot b_1 = 01 \cdot 01 = 0001$

$x_2 = a_1 \cdot b_0 = 01 \cdot 11 = 0011$

$x_3 = a_1 \cdot b_1 = 01 \cdot 01 = 0001$

3) Obtenemos los subproductos de Karatsuba multiplicando por la base

$x_4 = x_3 \cdot 2^{2 \cdot 2} = 0001 \cdot 10000 = 10000$

$x_5 = (x_2 + x_1) \cdot 2^2 = 0011 + 0001 \cdot 00100 = 10000$

4) Sumamos los subproductos y obtenemos el resultado final

$P = x_4 + x_5 + x_0 = 10000+10000+00011 = 100011$

Comprobamos que $5 \cdot 7 = 35 = (0101)_2 \cdot (0111)_2 = (100011)_2$. Podemos ver que en este algoritmo se utilizan multiplicadores pero de pocos bits y además se realizan en paralelo. Con esto se reduce el tiempo para obtener el resultado pero a costa de usar mucha más área.

5.2.3 Algoritmo de Booth

El algoritmo de Booth examina los pares adyacentes de los bits menos significativos del multiplicador. Se añade un bit extra que se coloca en la posición menos significativa del bit menos significativo del multiplicador. Se van observando los bits del multiplicador de la posición *i* y de la posición *i*-1 siendo la posición extra en la primera iteración un 0. Dependiendo del valor de este par se hace una operación u otra. La tabla de codificación de las operaciones a realizar es la siguiente:

B(i)	B(i-1)	Operación
0	0	No se hace nada
0	1	Producto = A * 2^i + Producto
1	0	Producto = A * 2^i - Producto
1	1	No se hace nada

Tabla 2: Códigos de operación de Booth

En este sentido es muy similar al algoritmo de sumas y desplazamientos pero con distintas operaciones. No hay un sentido pre-establecido en el cual ir iterando los bits. Se puede iterar desde el más significativo al menos o viceversa. Generalmente se itera de la forma habitual

que es desde el menor al mayor. De esta forma la multiplicación por 2^i se reemplaza por un registro desplazador de *i* posiciones a la izquierda. Los bits bajos pueden ser desplazados hacia fuera, y las sumas y restas siguientes pueden ser hechas justo en los *n* bits más altos del producto. El algoritmo de un multiplicador según el algoritmo de Booth que permite el cálculo de los 2k bits del resultado del producto en función de las entradas A(k-1,k-2,..,0) y B(k-1,k-2,..,0) es el siguiente:

```
ENTRADAS:
      A(k-1,k-2,..,0),B(k-1,k-2,..,0)
SALIDAS:
      Producto(2k-1,(2k-2,..,0)
CUERPO:
sumaAuxiliar = A;
for i = 0 to k do
      ultimoB = bin(P(0)); // consigue el último bit del  número
      penultimoB = bin(P(0)); // consigue el penúltimo bit del número
      if (ultimoB != penultimoB))
            if ( ultimoB == '0')
                  sumaAuxiliar = sumaAuxiliar+A;
            else
                  sumaAuxiliar = sumaAuxiliar-A;
            end if;
      end if;
Producto = desplazamientoCircularIzquierda(sumaAuxiliar);
end for;
sumaAuxiliar(k) = '0';
Salida = sumaAuxiliar;
```

El algoritmo es sencillo, como complejidad tiene el mantener los dos bits y las comparaciones con el código. Concretamente este algoritmo es muy eficiente cuando las entradas tienen largas cadenas de un mismo valor tanto 0 como 1. Sin embargo, si tienen valores alternantes entre 0 y 1 el algoritmo hace demasiadas operaciones. Siguiendo con el ejemplo anterior ($A = 5\ (0101)_2$ y $B = 7\ (0111)_2$) vemos como se reducen las sumas y multiplicaciones.

A = 010100000 ; -A=101100000 ; P = 000001110

1) i = 0; últimos bits = '10'; P = P-A = 101101110;

Desplazamiento circular P = 010110111;

2) i = 1; últimos bits = '11'; Nada
Desplazamiento circular P = 101011011;

3) i = 2; últimos bits = '11'; Nada
Desplazamiento circular P = 110101101;

4) i = 1; últimos bits = '01'; P = P+A = 001001101;
Desplazamiento circular P = 100100110;
Se desprecia el último bit extra y cambiamos el signo (+*+=+) 00010011

Comprobamos que $5 \cdot 7 = 35 = (0101)_2 \cdot (0111)_2 = (100011)_2$. Podemos ver que en este algoritmo se utilizan mucho los registros con desplazamiento además de utilizar un sumador y restador de forma iterativa. Al ser iterativo ocupara muy poco en área, sin embargo necesita de una cantidad de ciclos igual al número de bits de la entrada al igual que el algoritmo de sumas y desplazamientos.

5.3 Arquitecturas

Como se ha visto anteriormente, los algoritmos de cada multiplicador se han implementado en hardware para más tarde ver el rendimiento de cada componente a distintos anchos de entrada. En varios casos se ha modificado ligeramente el algoritmo para proporcionar algo de paralelismo y aprovechar al máximo el tiempo de reloj para realizar operaciones conjuntas.

5.3.1 Algoritmo de sumas y desplazamientos

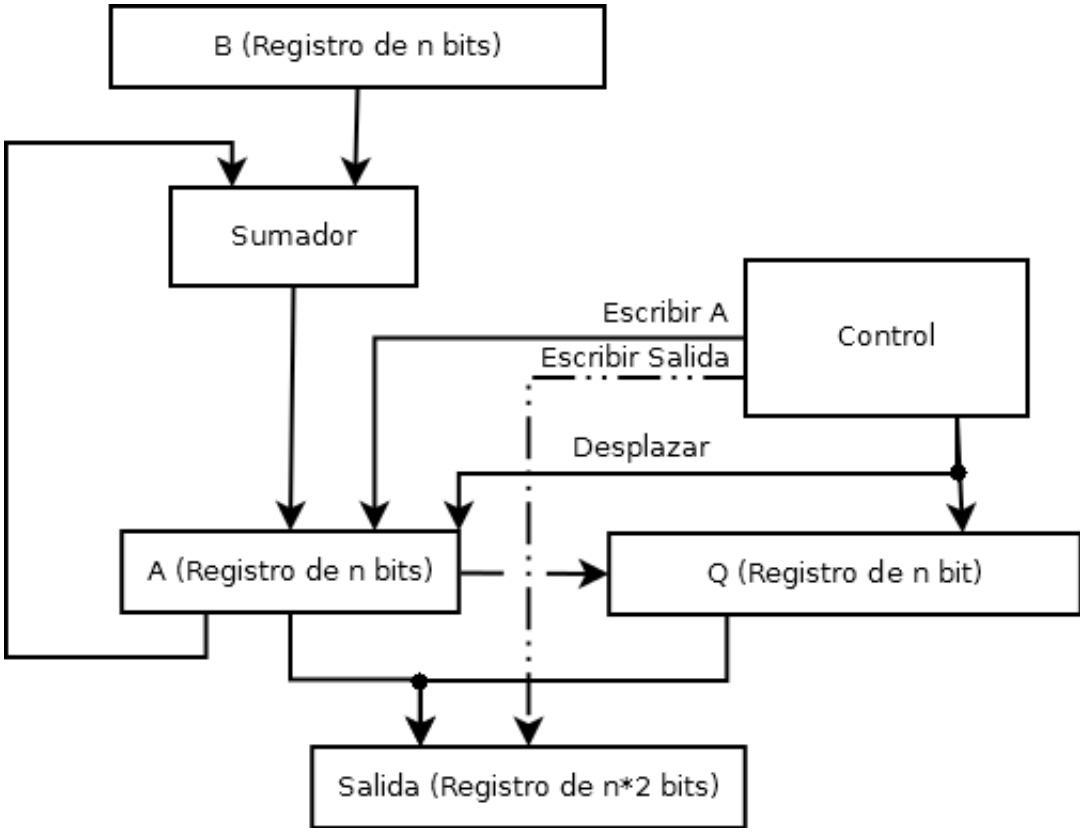


Figura 6: Detalle de un multiplicador a base de sumas y desplazamientos

El multiplicador a base de sumas y desplazamientos consta de varias partes. El módulo de control, el componente de suma y los registros con desplazamiento. El módulo de control es fácilmente implementable con una máquina de estados o un contador de modo que en cada estado se activan las señales de control correspondientes. El ciclo es muy sencillo, se suma *B* o no dependiendo del valor del bit *i* del registro *B* que está en el bloque de control al *registroA* y después se desplaza 1 bit. Esta rutina se ejecuta *k* veces, lo que hace un total de $2x+k$ ciclos de reloj siendo *x* el número de 1's que hay en el multiplicador. Se le ha añadido aparte una señal de "reset" y de "done" para completar el diseño del multiplicador. La mejora realizada dentro del módulo es la suma y el desplazamiento que se hace en el mismo ciclo. De esta forma tenemos solamente *k* ciclos por multiplicación.

5.3.2 Algoritmo de Karatsuba-Ofman

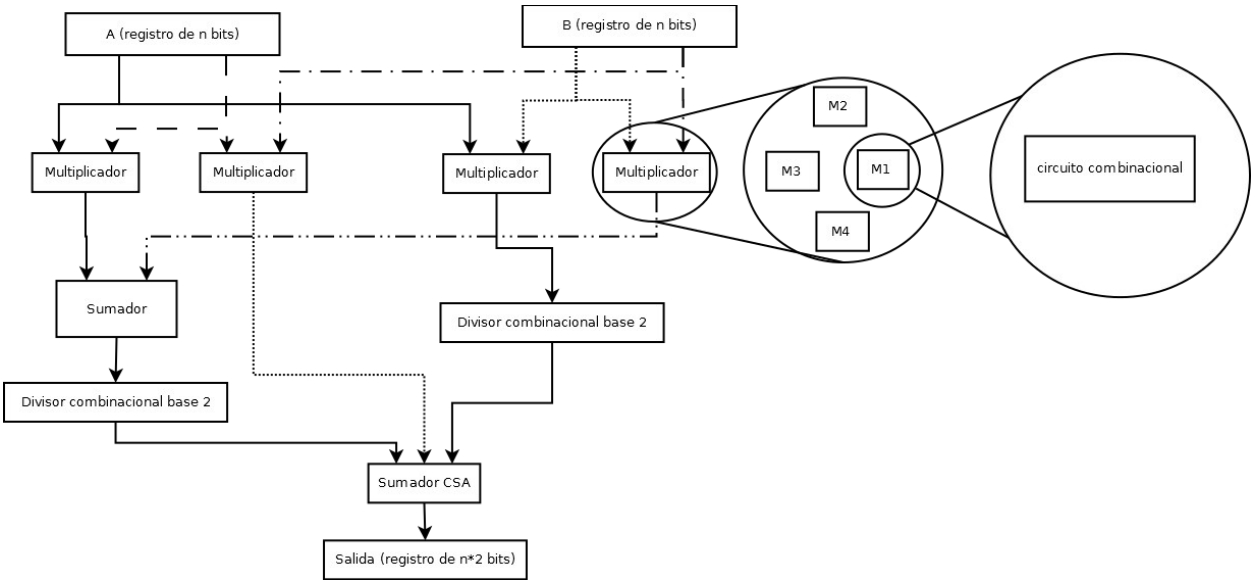


Figura 7: Detalle de un multiplicador de Karatsuba-Ofman

Como se ha visto anteriormente podemos dividir el algoritmo en unas 4 partes bien diferenciadas. Se ha implementado en sólo 3 partes incluyendo un sumador de 3 elementos o CSA *"Carry save adder"*. En la primera parte se realizan las 4 multiplicaciones en paralelo para obtener los valores de los productos de las partes de las entradas. Esta es la parte que recibe la recursión, pues estas multiplicaciones se realizan en una versiones con el ancho de las entradas más pequeñas del mismo algoritmo. La recursión de la multiplicación sigue hasta un caso base realiza una multiplicación de varios elementos mediante una tabla de verdad, como se puede observar en la figura 7. Esta idea hace que el diseño sea recursivo y se pueda replicar y aumentar de tamaño sin complicaciones. La segunda parte se hacen los desplazamientos a la izquierda simulando las multiplicaciones de 2^k y 2^{2k}. Y por último en la tercera parte se calcula la suma de las 3 partes con el sumador CSA y el valor de la señal de "done" se pone a 1 para denotar que el resultado de la multiplicación se ha terminado.

El método de multiplicación de Karatsuba-Ofman es principalmente combinacional. No requiere de ningún reloj para realizar la multiplicación. Como mejora se ha decidido implementarlo de forma segmentada de tal forma que cada paso corresponda a un ciclo de reloj. Con este cambio se puede conseguir un periodo de reloj mucho más pequeño y así mejorar el

tiempo total del sistema. El módulo de control es fácilmente implementable con una máquina de estados o un contador de modo que en cada estado se activan las señales de control correspondientes.

Debido a la recursividad cada componente realizará la multiplicación en 4 pasos más los pasos que se tarde en calcular la multiplicación del nivel inferior. Lo que deja al caso base (en el que la multiplicación se hace mediante una tabla de verdad combinacional) con 5 pasos y las demás con (4**n*)+5 pasos donde *n*-2 es el número de niveles que se emplean. Es decir, el logaritmo en base 2 del ancho del bus menos 2 siendo el ancho del bus mayor o igual que 4.

Como se ve en los resultados (Capítulo 9) este módulo de multiplicación será escogido por su velocidad. Por desgracia es también el que más área ocupa, aunque nuestro objetivo principal es la velocidad. Esto es debido a que el componente es recursivo y por tanto tiene que implementar muchos más módulos que un algoritmo iterativo.

5.3.3 Algoritmo de Booth

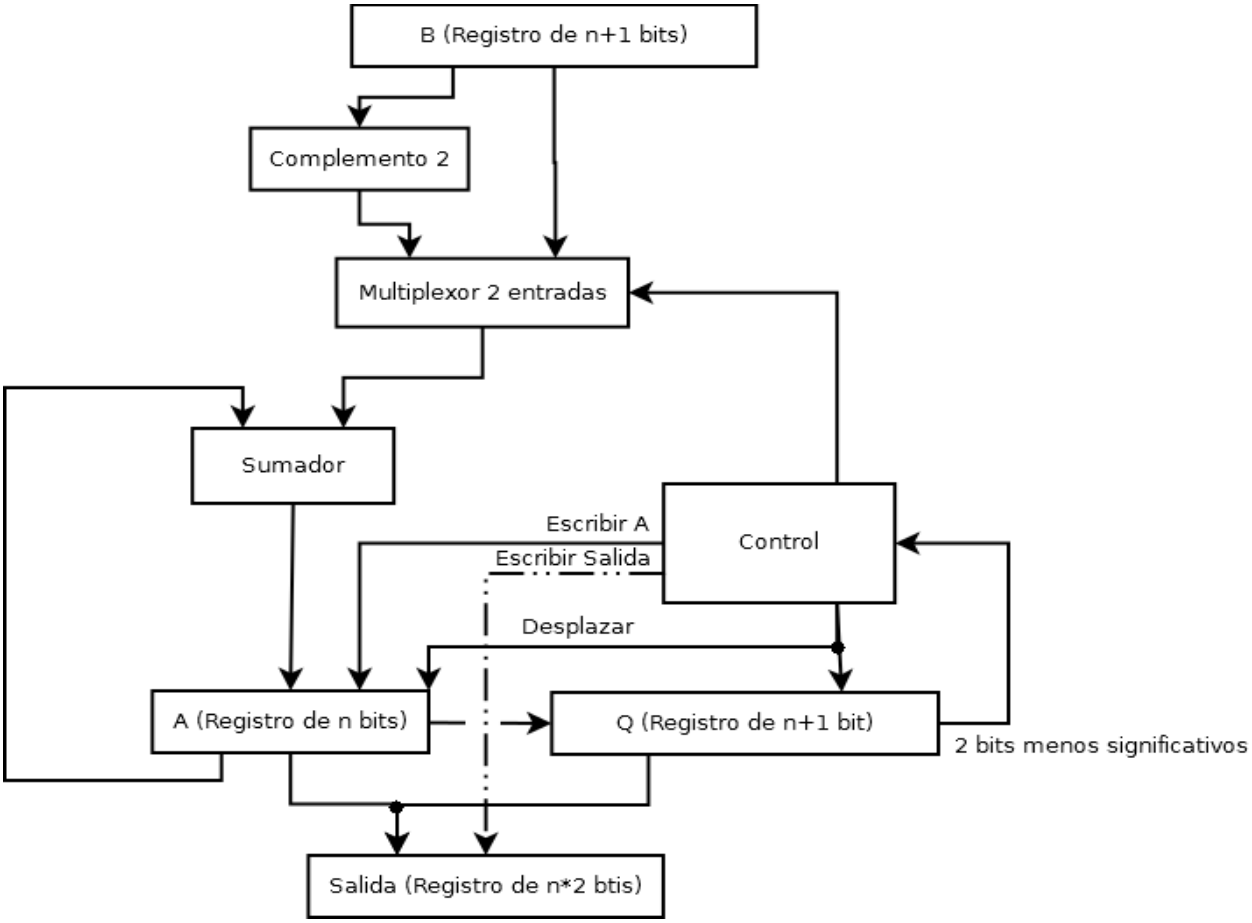


Figura 8: Detalle de un multiplicador de Booth

El multiplicador según el método de Booth es junto al de sumas y desplazamientos un algoritmo de multiplicación que ocupa poco en área. Esto es debido a que es iterativo y no recursivo como Karatsuba-Ofman. El módulo de control es fácilmente implementable con una máquina de estados o un contador de modo que en cada estado se activan las señales de control correspondientes. En este caso sólo tiene 5 estados para cargar, terminar y los cuatro estados restantes coinciden con los códigos de operación. Dado que dos hacen lo mismo, comparten el mismo código. Igual que el algoritmo de sumas y desplazamientos, el número de ciclos está definido por el multiplicador. En este caso el número de ciclos sera igual a $2+x+2y$ donde x es el número de pares consecutivos de cifras iguales ('00' y '11') e y el número de pares consecutivos de cifras distintos ('10' y '01'). Se han implementado el sumador de Kogge-Stone y un restador basándose en el sumador de Kogge-Stone para realizar las sumas y las restas de forma rápida.

Capítulo 6 - Reducciones

En este capítulo se detallan los componentes más importantes del proyecto, los módulos de reducción modular. Dentro de las diferentes versiones y algoritmos de un reductor modular se han escogido 3 candidatos para hacer un estudio de las diferentes funcionalidades y características orientadas objetivo final de incremento de velocidad. Los 3 algoritmos de reducción candidatos son: El algoritmo naive a base de restas, el algoritmo de Barret y el algoritmo de Montgomery.

6.1 Elementos

La reducción naive a base de restas, es un algoritmo simple e intuitivo. Consiste en ir restando el valor del módulo a un resultado parcial en cada ciclo de reloj hasta que este tenga un valor inferior al módulo. Requiere muy pocos elementos y un módulo de control muy pequeño. Por otro lado es posible que requiera bastante tiempo en obtener el resultado pero obtiene muy buenos resultados cuando el módulo y la entrada tienen una diferencia muy pequeña. Este reductor tiene como coste A/m restas, $(A/m)+1$ comparaciones donde A es el valor de la entrada y m el valor del módulo.

La reducción de Barret es el método que puede calcular un módulo de una forma más rápida que el método tradicional a base de restas y fue introducido por P.D.Barrett en [15]. La reducción de Barret requiere de un dato adicional denominado constante μ. Esta constante se obtiene haciendo una división entera entre el módulo y un valor que depende de k (el ancho del valor del módulo). De media, el número de ciclos necesarios (incluyendo la división entera) es mucho menor que con la reducción naive. El número de ciclos depende sobre todo de los ciclos de las multiplicaciones y de la división entera.

La reducción de Montgomery tiene la característica básica de permitir el cálculo eficiente de multiplicaciones modulares sin necesidad de hacer divisiones. El no necesitar divisiones enteras es de gran interés para reducir el coste computacional de los algoritmos de cálculo de reducciones modulares con números grandes. En varios de los algoritmos de reducción (algoritmo de Barret, algoritmo de reducción a base de divisiones), son especialmente las

divisiones (necesarias para el cálculo de los restos modulares), las operaciones de mayor coste. Por otra parte necesita de más valores de entrada que los demás.

6.2 Algoritmos

Cada uno de los reductores afronta la operación de diversas formas. Mientras unos se basan en una constante además de multiplicaciones, otros cambian de representación. A continuación se pueden ver las diferentes formas de calcularlo para cada reductor.

6.2.1 Algoritmo naive

El método naive para el cálculo del módulo de un número se basa principalmente en la idea primitiva de la división. El algoritmo resta sucesivas veces el valor del módulo a un resultado parcial, que toma el valor de la entrada al principio, hasta que tiene un valor inferior al del módulo. El algoritmo de un reductor según el algoritmo naive que permite el cálculo de los k bits del resultado de la reducción en función de las entradas *A(k-1,k-2,..,0)* y *m(k-1,k-2,..,0)* es el siguiente:

```
ENTRADAS:
      A(k-1,k-2,..,0),m(k-1,k-2,..,0)
SALIDAS:
      Reducción(k-1,k-2,..,0)
CUERPO:
restaAuxiliar = A;
while restaAuxiliar > m do
      restaAuxiliar = restaAuxiliar – m;
end while;
Reducción = restaAuxiliar;
```

Este algoritmo es increíblemente sencillo y fácil de implementar y que a primera vista ocupa muy poco espacio en área. Por otra parte depende mucho del valor de la entrada para hacer pocas iteraciones.

6.2.2 Algoritmo de Barret

El algoritmo de reducción de Barret se basa principalmente en la pre-computación de una constante μ que conociendo la entrada del módulo puede ser guardado en un registro o bien, mediante señales fijas. La constante μ, siendo *m* el valor del módulo y *k* el número de bits del módulo, se calcula de la siguiente forma:

$$\mu = \left(b^{(2k)} / m \right) \qquad (25)$$

Gracias a esta constante, se pueden calcular las demás variables de forma secuencial para obtener la reducción. Una de las restricciones de este algoritmo es que A, el valor de entrada, tiene que ser necesariamente el doble de ancho que el valor de *m*. El algoritmo de un reductor según el algoritmo de Barret que permite el cálculo de los *k* bits del resultado de la reducción en función de las entradas *A(2k-1,2k-2,..,0)* y *m(k-1,k-2,..,0)* y siendo *b* la base de representación de los valores de *A* y *m* es el siguiente:

```
ENTRADAS:
      A(2k-1,2k-2,..,0),m(k-1,k-2,..,0)
SALIDAS:
      Reducción(k-1,k-2,..,0)
CUERPO:
μ = floor(exp(2*k,b)/m);
q1 = floor(x / b^(k-1));
q2 = q1 * μ;
q3 = floor(q2 / exp(b,k+1));
r1 = x mod b^(k+1);
r2 = (q3 * m) mod exp(b,k+1);
r  = r1 - r2;
if (r < 0)
  r = r + b^(k+1);
end if;
while (r >= m) do
  r = r – m;
end while;
Salida = r;
```

Como muestra el algoritmo, μ no está pre-computado. Esto resulta en un algoritmo más general que con la constante fija ya que puede hacer multiplicaciones módulo *m* para cualquier valor que cumpla con la anterior restricción (*m* tiene que ocupar la mitad en anchura de *A*). Si la constante fuera fija el cálculo del módulo sería mucho más rápido al saltarse la división para conocer su valor. Para dar un versión más general al reductor, se optará no tener la constante μ pre-calculada. A continuación se expondrá un ejemplo del algoritmo en el que μ no está pre-calculado. Los valores de *A* y *m* son 109 $(01101101)_2$ y 13 $(1011)_2$ respectivamente y donde la base de representación $b = 2$.

1) Obtenemos el valor de la constante μ:
$\mu = 2^8/13 = 19$; 100000000 / 1011 = 10011

2) Calculamos el valor de las variables *q*
$q_1 = 109 / 2^3 = 13$; 01101101 / 100 = 1101
$q_2 = 13 \cdot 19 = 247$; 1101 · 10011 = 11110111
$q_3 = 247 / 2^5 = 7$; 11110111 / 100000 = 00000111

3) Calculamos el valor de las variables *r*
$r_1 = 109 \bmod 32 = 13$; 01101101 mod 100000 = 00001011
$r_2 = (7 \cdot 13) \bmod 32 = 27$; 0111 · 1011 = 1011011 mod 100000 = 0011011
r = 13 - 27 = - 14

4) Comprobamos el valor de *r* y sumamos un offset de $2^{(k+1)}$ si fuera necesario
r = -14 + 32 = 18

5) Comprobamos de nuevo el valor de *r* y hacemos el módulo de la forma clásica
resultado = 18 - 13 = 5 ; 00010010 – 00001011 = 00000101

Comprobamos que $109 \bmod 13 = 5 = (01101101)_2 \cdot (1011)_2 = (101)_2$. Se puede ver que en este algoritmo se puede paralelizar el cómputo de ciertos valores para ganar tiempo. Además, la ventaja de usar una base binaria y anchos potencias de 2 hace que muchas de las operaciones de

división y las reducciones modulares que se necesitan sean triviales. Las divisiones se reducen a simples desplazamientos a la derecha (menos en el cálculo de la constante μ)y las reducciones a sesgar el resultado. Por otra parte si se evita el cálculo de la constante μ el algoritmo que resulta es extremadamente rápido.

6.2.1 Algoritmo de Montgomery

El algoritmo de Montgomery es un algoritmo que antes de poderse ejecutar necesita de ciertas verificaciones. Para que funcione correctamente se necesita que teniendo *m, R* y *A,* donde *A* es la entrada, *m* el módulo y *R* valor auxiliar.

$$0 \leq A \leq m \cdot R \tag{26}$$

Otra de las restricciones es que *m* y *R* tienen que ser primos entre sí, es decir, *mcd(m,R) = 1.* Esto se consigue muy fácilmente dejando *m* siendo impar y *R* siendo par o viceversa. De esta forma se tiene que la reducción de Montgomery o *A* módulo *m* con respecto a *R* como el valor T_r que cumple la siguiente congruencia.

$$T_r \equiv A\,R^{(-1)} \bmod m \tag{27}$$

Bajo estas condiciones se define el cálculo de T_r, como una función nueva Mont() en la que siendo $R = 2k$ el uso de multiplicaciones y divisiones es trivial como se verá a continuación. El algoritmo de un reductor según el algoritmo de Montgomery que permite el cálculo de los *k* bits del resultado de la reducción en función de las entradas *A(k-1,k-2,..,0), m(k-1,k-2,..,0)* y *R(k-1,k-2,..,0),* es el siguiente:

```
ENTRADAS:
      A(k-1,k-2,..,0),m(k-1,k-2,..,0),R(k-1,k-2,..,0)
SALIDAS:
      Reducción(k-1,k-2,..,0)
CUERPO:
c = -m mod R;
U = A · c mod R;
auxValor = ( A + U · m ) / R;
```

```
if (auxValor >= m)
       auxValor = auxValor – m;
end if;
Reducción = auxValor;
```

Dado que las únicas divisiones, al igual que los módulos, son por *R* y este es primo con *m* y potencia de 2, el coste computacional se reduce de manera apreciable ya que las divisiones se convierten en simples desplazamientos a la derecha y los módulos en simples sesgos del valor a un ancho determinado. A continuación se expone un ejemplo del algoritmo con los valores de *A* = 62578, *R* = 1024 y *m* = 451;

1) Calculamos el valor de *c*

   c = (-451) mod 1204 = 277

2) Calculamos el valor de U

   U = (62578·277) mod 1024 = 858

3) Calculamos el valor de auxValor

   auxValor = (62578+(858·451)) / 1024 = 439

4) Como auxValor < m no se hace nada

Comprobamos que 62578·1024 mod 451 = 439. Se puede ver que en este algoritmo no se puede paralelizar dado que cada paso depende del anterior para hacer sus cálculos. Pero la ventaja de usar una base binaria y una variable *R* potencia de 2 hace que el algoritmo se termine de manera muy rápida.

6.3 Arquitecturas

Como se ha visto anteriormente, los algoritmos de cada reductor se han implementado en hardware para más tarde ver el rendimiento de cada componente a distintos anchos de entrada. En varios casos se ha modificado ligeramente el algoritmo para proporcionar algo de paralelismo como se ha mencionado en el apartado de los algoritmos y así aprovechar al máximo el tiempo de reloj para realizar operaciones conjuntas.

6.3.1 Algoritmo de reducción naive

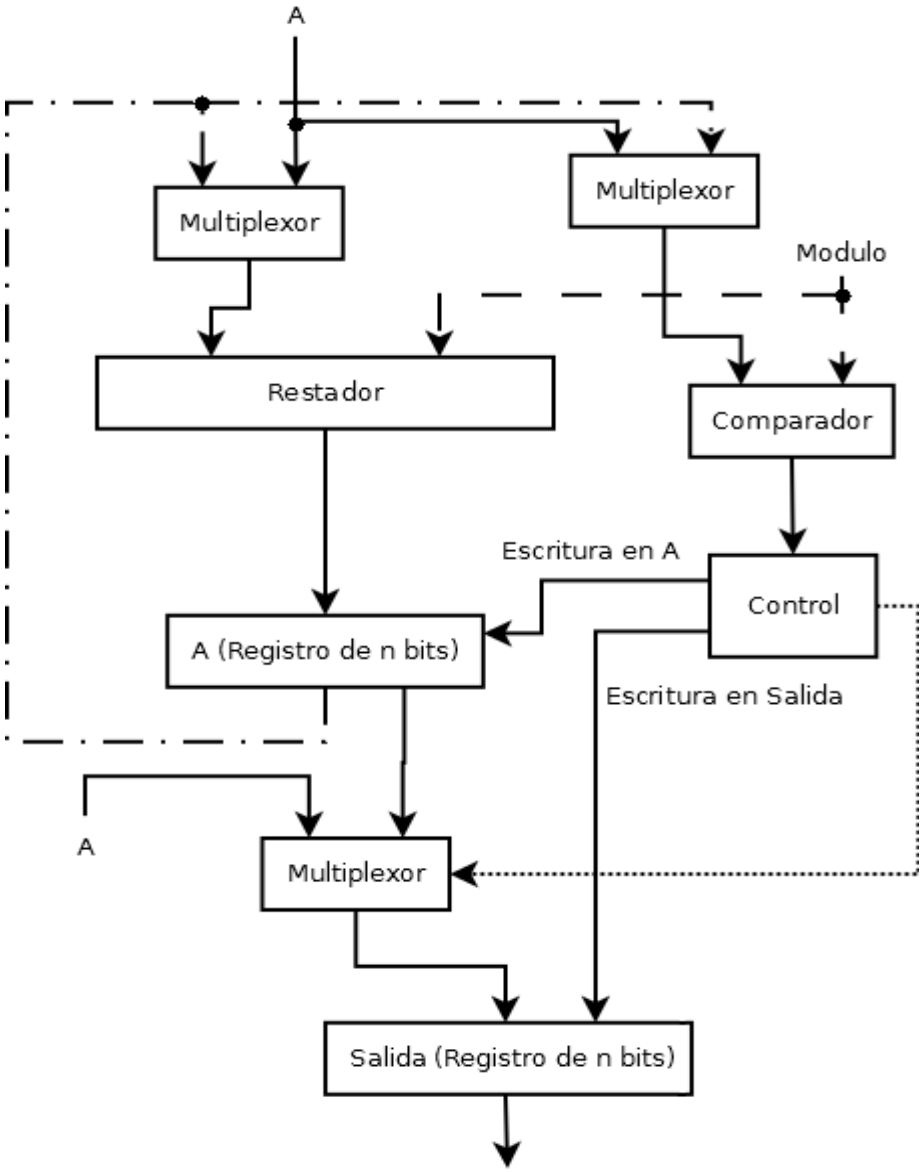


Figura 9: Detalle de un reductor naive

El reductor naive es el reductor más simple que se conoce. Gracias a que tiene muy pocos componentes, (2 registros, el módulo de control, el restador, el comparador y varios multiplexores). Este reductor ocupa muy poco área. El módulo de control es fácilmente implementable con una máquina de estados o un contador de modo que en cada estado se activan las señales de control correspondientes. Este reductor actúa en base al resultado del comparador. Por lo tanto existen 4 estados teniendo en cuenta el estado de inicio. Para mejorar la efectividad del sistema, el restador se ha construido a base del sumador de Kogge-Stone. En principio no es

una buena opción ya que puede necesitar muchos ciclos de reloj, concretamente *A/m*. Pero es interesante ver que si el módulo toma un valor próximo de la entrada consume muy pocos ciclos.

6.3.2 Algoritmo de reducción de Barret

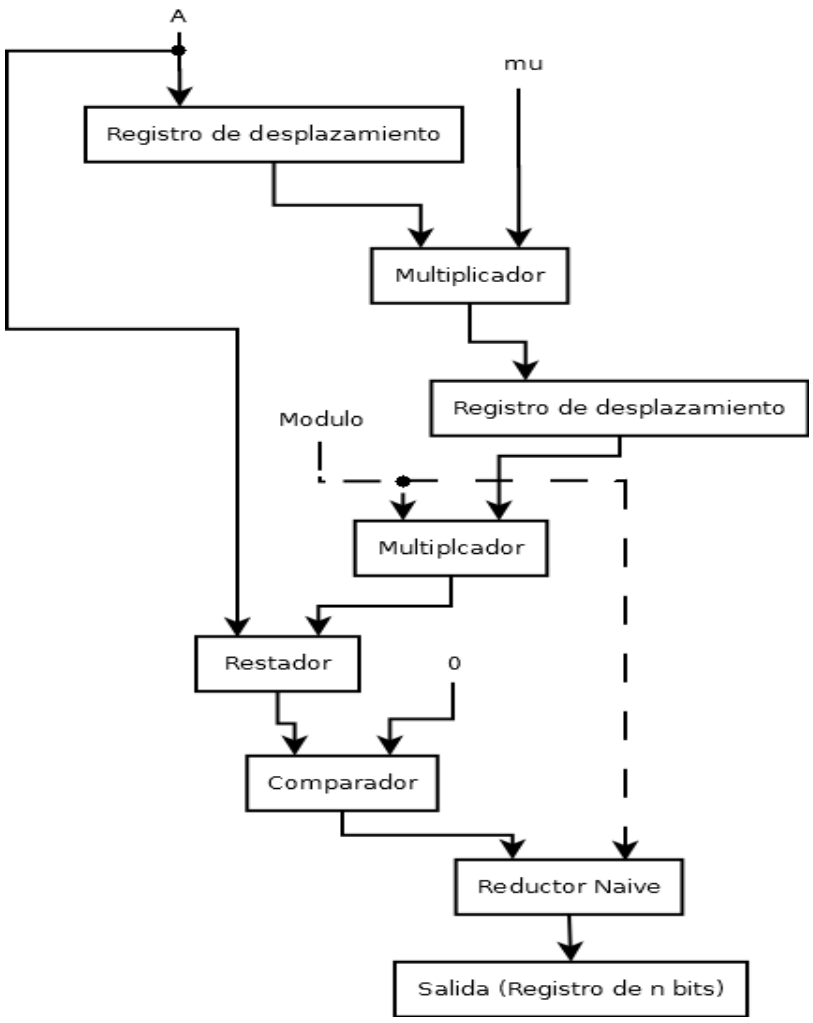


Figura 10: Detalle de un reductor de Barret

Como se mencionó anteriormente, se puede hacer alguna mejora con respecto al algoritmo para la obtención de variables auxiliares o secundarias como el cálculo simultaneo de q_1 y r_1 . El módulo de control es fácilmente implementable con una máquina de estados o un contador de modo que en cada estado se activan las señales de control correspondientes. Dado que el último paso del algoritmo es un bucle *"while"* similar al reductor naive, se ha utilizado el módulo de reducción naive para ese paso del algoritmo.

Para las 2 multiplicaciones que se encuentran en el algoritmo se dispone de los módulos de multiplicación de Karatsuba-Ofman que se vieron en el anterior capítulo para mejorar los tiempos. Con sólo 10 ciclos de reloj podemos obtener el resultado de la reducción. Estos ciclos están calculados sin tomar en cuentan los pasos que se necesitan para las dos multiplicaciones ni para el cálculo de μ.

Para proveer al reductor una forma para la cual no tenga que calcular la constante μ se ha usado una señal dentro del mismo que indica si μ debe o no ser calculada y un registro para alojar este valor. Esta señal evita la división de grandes proporciones que se necesita para el cálculo de dicha constante. Como se muestra en los resultados de implementación (Capítulo 9) este módulo es elegido como reductor para el diseño final de la multiplicación modular gracias a su versatilidad con la constante μ así como por su velocidad.

6.3.3 Algoritmo de reducción de Montgomery

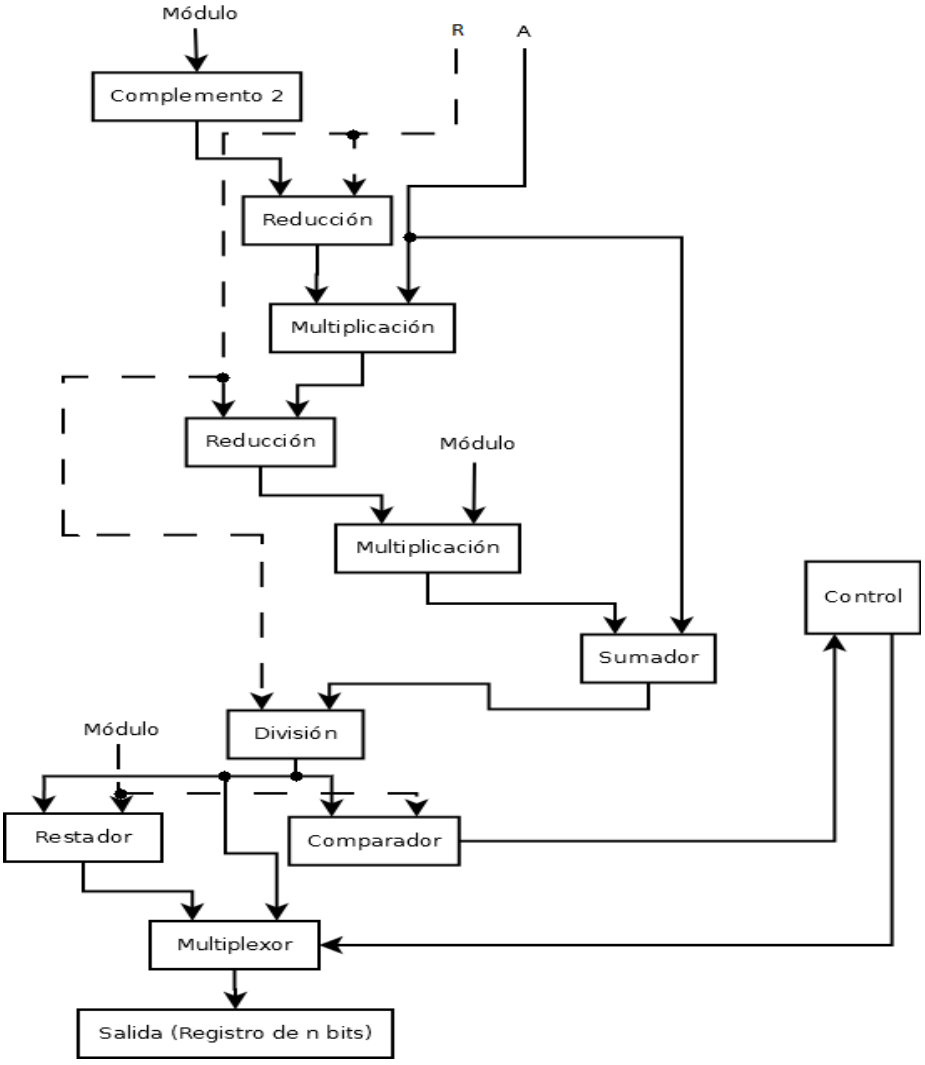


Figura 11: Detalle de un reductor de Montgomery

El multiplicador reductor de Montgomery es el reductor más complejo. No sólo por que tenga la multiplicación sino por el propio algoritmo. Como se mencionó antes si *R* está en base 2 las divisiones y reducciones se pasan a ser triviales. En este caso *R* está dentro del sistema y se ejecutan las operaciones en base a ese *R* especificado en el diseño. Es interesante ver como calcula el módulo aunque para ello necesite más valores que los demás reductores. Sin embargo, el que necesite de este valor hace que los componentes que utilicen este reductor tengan que calcularlo para cada valor de entrada o escoger uno suficientemente grande para resolver (27). El módulo de control es fácilmente implementable con una máquina de estados o un contador de modo que en cada estado se activan las señales de control correspondientes.

Capítulo 7 - Algoritmo de preparación modular

Se han visto en los capítulos anteriores la mayor parte de los componentes que se usarán en la implementación del módulo de multiplicador modular final. En este capítulo se explica el módulo principal encargado del aumento de velocidad en ese cálculo modular. Gracias a este módulo la mayor parte de las multiplicaciones toman valores más pequeños lo que hace que algunas no necesiten hacer la reducción.

7.1 Introducción

La mayoría de los algoritmos de multiplicación modular operan con los datos de entrada A y B para calcular $A \cdot B \bmod m$. Que se haya visto, ninguno hace nada con los datos. Por otra parte puede que el modificar los parámetros de entrada no sea una buena idea. ¿Por qué calcular $C \cdot D$ si lo que se pretende es calcular $A \cdot B$? Además, A,B,C y D son enteros, deben estar relacionados de alguna forma y deben cumplir la siguiente congruencia para módulo m, también entero:

$$A \cdot B \bmod (m) \equiv C \cdot D \bmod (m) \qquad (28)$$

Computacionalmente costaría lo mismo o incluso más, dependiendo de los valores de A,B,C y D. La idea propuesta se basa en una propiedad que no se ha visto reflejada en ningún trabajo anterior sobre en la aritmética modular. Esta propiedad con A,B y m como valores enteros es la siguiente:

$$(A - m) \cdot (B - m) \bmod m = A \cdot B \bmod m \qquad (29)$$

Esta ecuación permite que la multiplicación modular de A y B módulo m sea más sencilla. Sólo tiene una restricción, mucho más general que con Montgomery y que admite un número de elementos mucho más grande en las operaciones modulares. Esta restricción se puede definir como para dos números enteros A y B tales que son mayores que 0 y menores que m.

Se han realizado numerosas simulaciones en software para verificar el resultado y se cumple en todas las ocasiones. Posteriormente se detalla la demostración de esta dicha propiedad

(29). Con esta restricción la multiplicación modular se puede definir cambiando las entradas por la resta de sí mismas y el valor del módulo como en (29). De esta forma, restando el valor de *m* a las entradas se puede conseguir una entrada más pequeña en valor sólo si dicha entrada es mayor que la mitad de *m,* o de lo contrario, el resultado sería mayor que esta y no se gana nada. Se puede definir una multiplicación mucho más sencilla aún en una revisión de la propiedad (29) en 3 casos clave siendo *A* y *B* entradas de una multiplicación modular, *m* el valor del módulo:

- Si A y $B \leq m/2$

  La multiplicación se realiza como siempre, no hay cambios.

- Si A y $B > m/2$

Si las dos entradas son mayores que $m/2$ se puede restar el valor de *m* a cada uno de los operandos para hacer que la operación sea mucho menor tanto en tiempo de computo como en espacio para representarlo.

- Si A ó $B > m/2$ (sólo uno de los dos)

Es el caso más frecuente y el más complejo. Si una de las entradas es mayor que $m/2$ se puede restar el valor de *m* para que el operando sea mucho menor (como mucho sera de $(m/2)-1$) pero hay que hacer un pequeño ajuste en (29). Hay que restar de nuevo la cantidad del módulo y el resultado de la resta en valor absoluto es el resultado del módulo. La demostración de (30) se puede ver a continuación.

$$(m-(m-A)\cdot B \bmod m) = A \cdot B \bmod m \qquad (30)$$

7.1.1 Demostración teórica

La demostración a la ecuación (29) dados *A, B* y *m* números enteros es:

$((A - m) \cdot (B - m)) \bmod m =$

$((A \cdot B) - (A \cdot m) - (B \cdot m) + M^2) \bmod m =$

$(A \cdot B) \bmod m - (A \cdot m) \bmod m - (B \cdot m) \bmod m + (m^2) \bmod m =$

Como:

$(A \cdot m) \bmod m = 0$

$(B \cdot m) \bmod m = 0$

$(m^2) \bmod m = 0$

Se tiene después de sustituir:

$(A \cdot B) \bmod m - 0 - 0 + 0 = (A \cdot B) \bmod m$

De esta forma queda demostrada la ecuación (29) y por tanto la base de la aceleración de la multiplicación modular.

La demostración a la ecuación (30) dados *A, B* y *m* números enteros es:

$(m-(m-A) \cdot B) \bmod m =$

$(m-((mB-AB) \bmod m)) \bmod m =$

Como :

$mB \bmod m = 0$

$(m-(-AB \bmod m)) \bmod m =$

$(m + AB \bmod m) \bmod m =$

Como :

$m + X \bmod m = X \bmod m$

$((m + AB \bmod m)) \bmod m = AB \bmod m$

De esta forma queda demostrada la ecuación (30) y por tanto la simplificación de las multiplicaciones.

7.2 Algoritmo

Dado que el módulo de preparación es sólo la parte inicial, el algoritmo es muy sencillo. El algoritmo de una preparación para la multiplicación modular que permite el cálculo de los *k* bits de los resultados de la preparación en función de las entradas *A(k-1,k-2,..,0), B(k-1,k-2,..,0)* y *m(k-1,k-2,..,0),* es el siguiente:

ENTRADAS:

```
A(k-1,k-2,..,0),B(k-1,k-2,..,0),m(k-1,k-2,..,0)
```

SALIDAS:

```
APreparado(k-1,k-2,..,0),BPreparado(k-1,k-2,..,0)
```

CUERPO:

```
mitadM = floor(m/2);
if (A > mitadM)
	APreparado = m-A;
else
	APreparado = A
end if;
if (B > mitadM)
	BPreparado = m-B;
else
	BPreparado = B
end if;
```

Siendo *A* y *B* números enteros menores que *m* que también es entero. La división de *m* es una división entera. A continuación, un ejemplo del algoritmo y su funcionamiento. Los valores de *A* y *B* son 10 (1010) y 4 (0100) respectivamente y el de *m* es 11 (1011).

1) Se obtiene la mitad del módulo *m*

m/2 = (0101)

2) Se restan los valores de *A* y *B* al módulo *m*

APreparado = 1011 – 1010 = 0001

BPreparado = 1011 – 0100 = 0111

3) Se comprueban que son menores que la mitad del módulo *m* y se devuelven

devolver 0001

devolver 0100

7.3 Arquitectura

El algoritmo de este módulo de preparación se han implementado en hardware para posteriormente comprobar el rendimiento a distintos anchos de entrada.

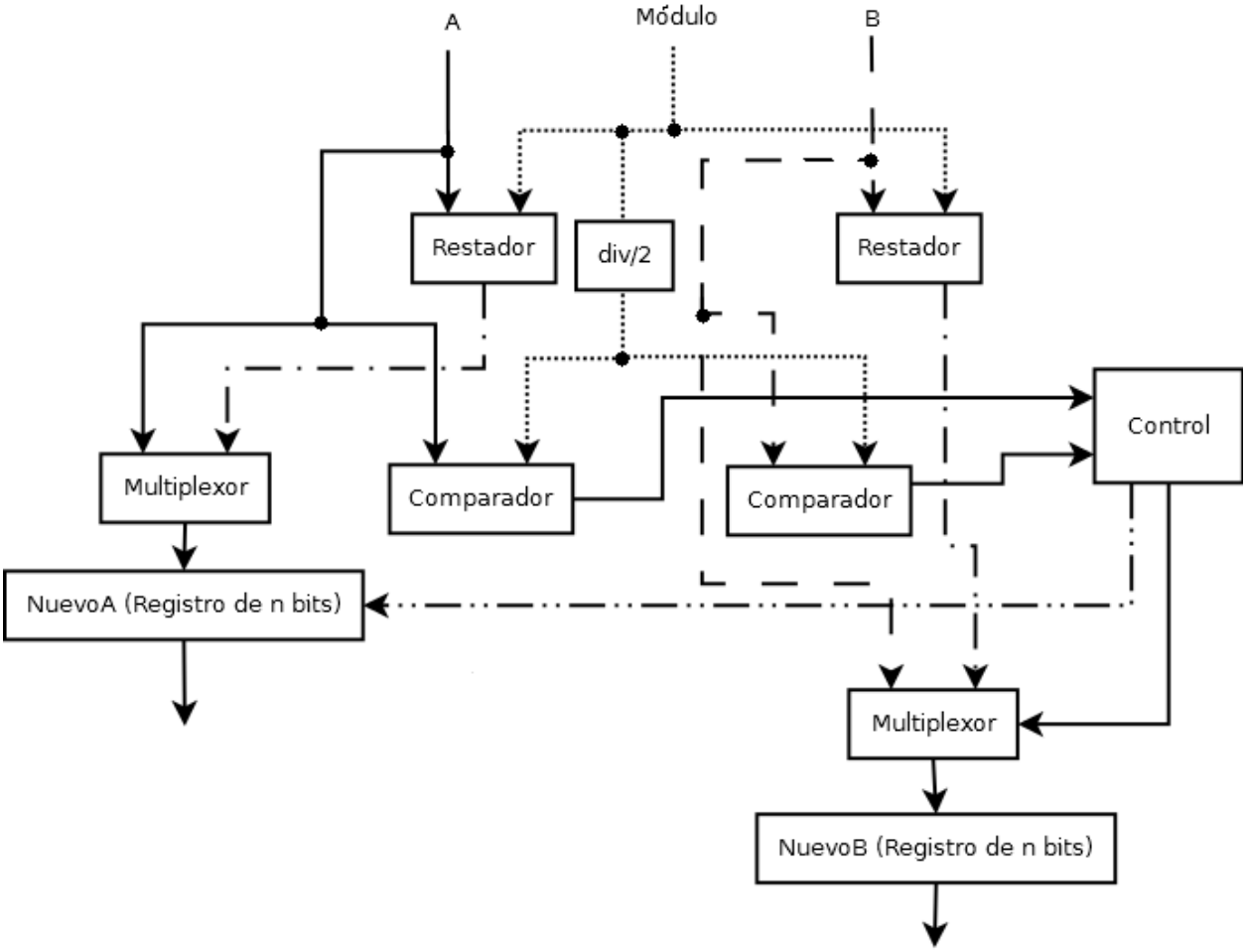


Figura 12: Detalle del módulo de preparación

Este es un módulo muy sencillo. Con 2 restadores y 2 comparadores se tienen todos los elementos necesarios para crearlo. Aunque se puede hacer totalmente combinacional, se ha decidido construirlo de forma secuencial. El motivo es que de esa forma se puede conseguir un periodo mucho menor con poco esfuerzo. El módulo de control es fácilmente implementable con puertas lógicas pues apenas tiene señales de control. En sólo 2 ciclos de reloj (independientemente del ancho de las entradas) se pueden tener ambos datos procesados para su futura multiplicación.

Capítulo 8 - Multiplicación modular

En este capítulo se detalla el componente final de todo el proyecto, un módulo multiplicador modular. Esta operación "$A \cdot B \bmod m$" es una de las más importantes en el campo de la criptografía. Es en este capítulo donde se implementa la aceleración para este tipo de operaciones modulares.

8.1 Introducción

A lo largo de todo el proyecto se han visto los elementos necesarios para una multiplicación modular. Como se ha mencionado antes, con sólo la multiplicación de Karatsuba-Ofman y la reducción de Barret se consiguen unos tiempos bastante más que aceptables (según los datos experimentales obtenidos en este trabajo en el capítulo 9). Utilizando el módulo de la preparación del capítulo anterior podemos conseguir que ciertas multiplicaciones que al principio puedan llevar a una reducción, finalmente no sea necesaria. Gracias a la propiedad (29) y (30) vistas anterior mente.

Desafortunadamente, hay muchas posibilidades que pueden ocurrir al utilizar el módulo de preparación. Opciones que tienen que ser contempladas aumentando ligeramente la dificultad y el área del dispositivo, pero que sin duda, aumentan la velocidad en ciertos casos.

8.2 Algoritmo

El algoritmo de multiplicación modular pone en uso la propiedad de preparación (29), que estará codificado como la función prep(int, int). Esta función devolverá 0 si no se cambia u otro valor si se ha cambiado. El algoritmo de multiplicación modular usando la función prep() que permite el cálculo de los *k* bits del resultado del producto modular en función de las entradas *A(k-1,k-2,..,0), B(k-1,k-2,..,0)* y *m(k-1,k-2,..,0),* es el siguiente:

ENTRADAS:

```
A(k-1,k-2,..,0),B(k-1,k-2,..,0),m(k-1,k-2,..,0)
```

SALIDAS:

```
productoModular(k-1,k-2,..,0)
```

CUERPO:

```
auxA = prep(A,m);
if (auxA = 0)
       auxA = A;
       cambioA = true;
end if;
auxB = prep(B,m);
if (auxB = 0)
       auxB = A;
       cambioB = true;
end if;
auxAB = A · B;
if (auxAB > m)
       auxRed = reduccion(auxAB);
else
       auxRed = auxAB;
end if;
if (!(cambioA XOR cambioB)) // si se cambian ambos o no se cambia ninguno
       productoModular = auxRed
else
       productoModular = auxRed-m;
end if;
```

Como se puede observar el uso de la función prep() hace que se aumentan los casos de sentencias condicionales para tener en cuenta los casos de (30). A continuación, un ejemplo del algoritmo y su funcionamiento en dos casos distintos. En este primer caso no se necesitará una reducción modular. Los valores de entrada son de A y B con 23 $(10111)_2$ y 21 $(11001)_2$, respectivamente, y el de m es 27 $(11011)_2$.

1) Calculamos los valores de preparación de A y B

$nuevoA = m - A = 27 - 23 = 4$

$nuevoB = m - A = 27 - 21 = 6$

2) Multiplicamos:

$AB = 4 \cdot 6 = 24$

3) No hace falta reducir pues $24 < m$

Comprobamos, $21 \cdot 23 \bmod 27 = 24$. Como se puede ver, no ha hecho falta la reducción ya que se ha deducido el resultado de la multiplicación usando el módulo de preparación. De no

haberse utilizado, el resultado de la multiplicación hubiera necesitado la posterior reducción (21 · 23 = 483). En este segundo caso tampoco se necesitará una reducción modular pero sí una resta, pues solo se modifica una de las entradas. Los valores de entrada son A y B con 23 $(10111)_2$ y 6 $(00110)_2$ respectivamente y el mismo valor de m que en el anterior ejemplo:

1) Calculamos los valores de preparación de A y B
$nuevoA = m - A = 27 - 23 = 4$
$nuevoB = B = 6$

2) Multiplicamos:
$AB = 4 \cdot 6 = 24$

3) No hace falta reducir pues $24 < m$

4) Dado que solo uno ha sido modificado, restauramos el valor
$m - AB = 27 - 24 = 3$

Comprobamos, $23 \cdot 6 \bmod 27 = 3$. Como se puede ver, no ha hecho falta la reducción ya que se ha deducido el resultado de la multiplicación usando el módulo de preparación, pero se ha necesitado una pequeña resta. De no haberse utilizado, el resultado de la multiplicación hubiera necesitado la posterior reducción (21 · 6 = 138).

8.3 Arquitectura (Versión 1)

El algoritmo de este módulo de multiplicación modular se ha implementado en hardware para comprobar posteriormente ver el rendimiento a distintos anchos de entrada.

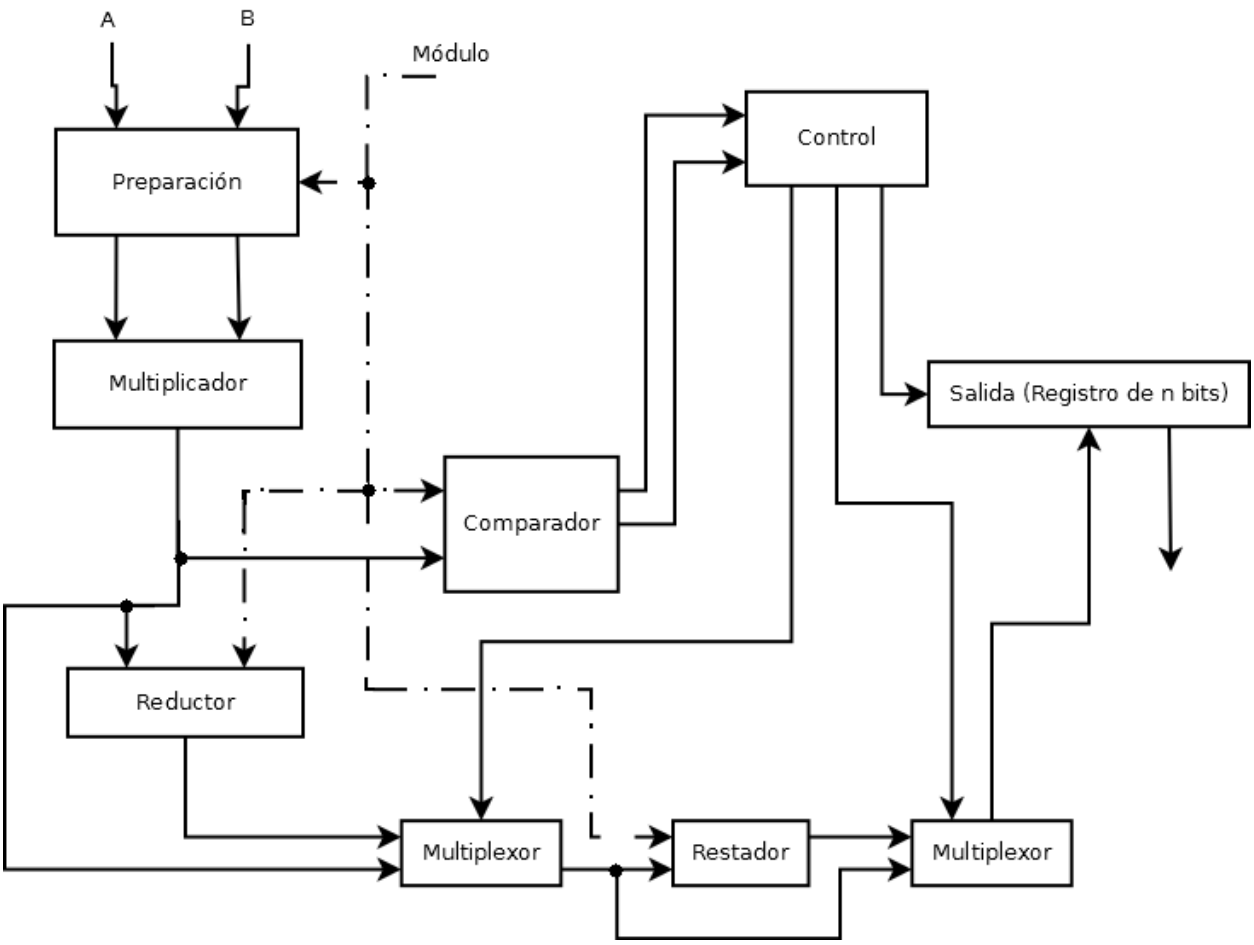


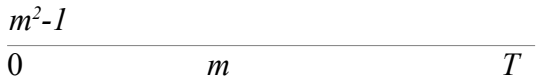
Figura 13: Detalle del módulo de multiplicación modular

La dificultad de este módulo de multiplicación modular radica en el hecho de que hay muchos casos que se pueden dar cuando se utiliza el módulo de preparación. Además el uso de este módulo de preparación añade al resultado total de ciclos, unos pocos ciclos de reloj (concretamente 3). El módulo de control es fácilmente implementable con una máquina de estados o un contador de modo que en cada estado se activan las señales de control correspondientes. Se ha simplificado el módulo de control en el detalle para que sea más sencilla la apreciación del diseño.

El conjunto en total necesitará tantos ciclos como necesite el algoritmo de Karatsuba-Ofman y Barret más 2 por la preparación modular en los casos que haga falta reducir. Pero por otra parte se han aumentado más del doble los casos en los que no se necesita la reducción. Esto es posible ya que los valores de *A* y *B* al restar el valor de *m* tienen un valor menor y por tanto su

producto también es menor. Sobre el eje real y con *A* y *B* enteros mayores que *m*/2 es más fácil verlo

Como se puede apreciar, en esta recta están todos los valores del producto de *A* y *B*. La variable T es el valor máximo que puede tomar la multiplicación modificando los valores de *A* y *B* siendo estos menores que el módulo y mayores que *m*/2 $(A-m)\cdot(B-m) = (m/2)^2$. Esta recta puede contener más valores o menos dependiendo de los valores de *A,B* y *m* pero los valores que están reflejados en esta línea estarán en la misma posición y a una distancia de la misma proporción. Si no usamos el módulo de preparación, el producto de *A* y *B* estará contenido en toda la recta, mientras que si lo usamos la recta se pliega por la variable *T*, llevando los valores a la derecha de m^2-1 a la izquierda. De este modo que la recta quedaría de la siguiente forma:



Se puede ver ahora que los valores del producto de A y B una parte de ellos está de nuevo entre el 0 y *m*. Esos son los valores del producto que ahora no necesitan reducción, el doble.

8.4 Arquitectura (Versión 2)

Al implementar el diseño anterior no se aprovechaba demasiado los beneficios que nos otorga el módulo de preparación. Gracias a este módulo, muchas de las multiplicaciones modulares se ven reducidas a simples multiplicaciones como se ha podido ver anteriormente. Las demás sólo añaden 2 ó 3 ciclos más frente a tener solo la multiplicación y la reducción.

El verdadero potencial del módulo de preparación es que el resultado del producto de dos entradas pasando por este módulo, es siempre menor o igual a la multiplicación de las mismas

sin pasar por él. Teniendo ese hecho en mente la reducción modular de Barret no es de gran ayuda, ya que se basa principalmente en el ancho de las entradas. Como mucho el producto tras pasar por el módulo de preparación tendrá 2 bits menos pero esto es irrelevante ya que todos los componentes que se han creado tienen anchos de 2^n bits por requerimientos de los algoritmos.

Volviendo al capítulo de las reducciones, hay una en concreto que se puede utilizar dado este caso y optimizar así el tiempo. Esta reducción es la reducción naive. Gracias a que siempre el producto de las entradas tratadas sera menor o igual que el producto de las entradas sin tratar, se puede utilizar la característica principal del método naive dentro del módulo de multiplicación modular. Este cambio añade velocidad en el cálculo del módulo en muchos más casos que sin él, dado que no tendrá que realizarse todo el módulo de reducción de Barret si el módulo naive de reducción termina antes.

Con la idea del anterior punto, es posible mejorar la arquitectura añadiendo un par de módulos más para aumentar la velocidad. Aprovechando el paralelismo se puede realizar la reducción Naive y la reducción de Barret al mismo tiempo cuando se tenga que reducir el valor. De esta forma habrá que esperar solamente al reductor más rápido. Este cambio, como se ha dicho antes, añade velocidad al sistema pero aumenta el área y el consumo al tener dos módulos de reducción trabajando en paralelo.

En la figura 14 se muestra la nueva arquitectura modificada. Para diferenciarla con la mostrada en la figura 13 los componentes añadidos se muestran en color rojo.

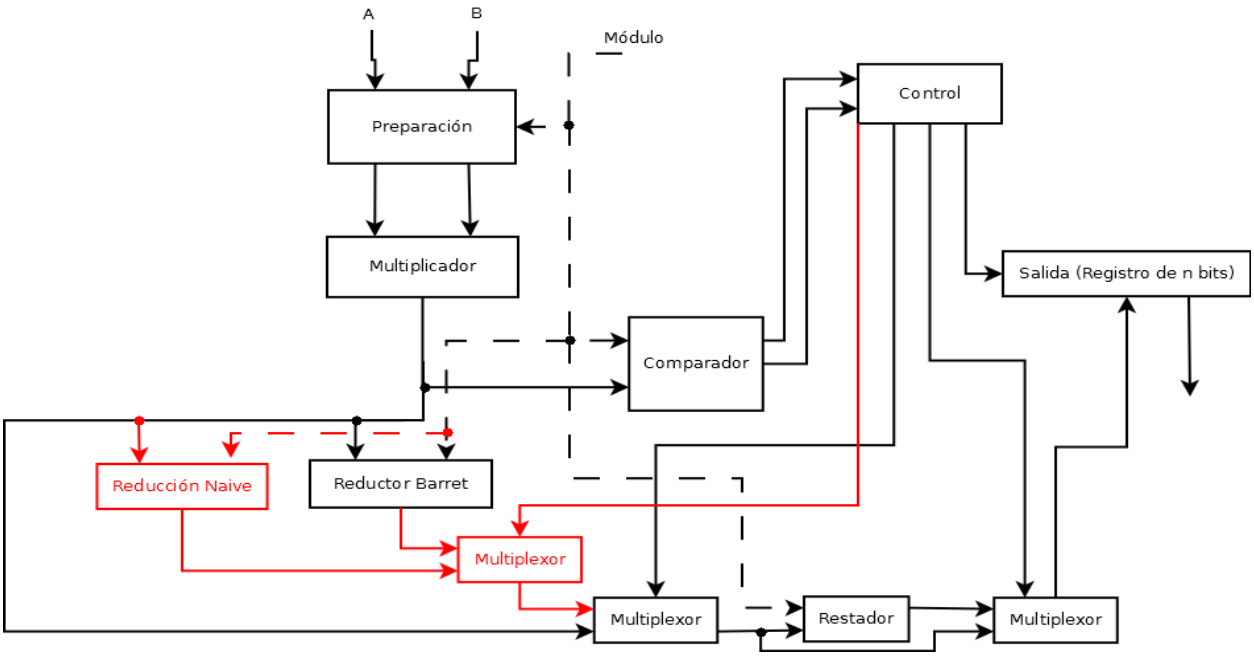


Figura 14: Detalle de un módulo de multiplicación modular revisado

Ahora el conjunto en total necesitará tantos ciclos en el caso peor como necesite el algoritmo de Karatsuba-Ofman y Barret más 2 por la preparación modular en los casos que haga falta reducir. De esta forma cuando haya que reducir sera el componente más rápido (naive o Barret) el que de la solución y se continúe con la ejecución del algoritmo. Volviendo a la recta anterior, el número de casos en los que el producto de *A* y *B* es cercano a *m* se multiplica por 2. Además estas reducciones que antes tardaban los ciclos del módulo de Barret ahora tardan mucho menos porque el reductor naive es mucho más rápido reduciendo los valores del producto que están situados en esos puntos de la recta (línea roja).

m^2-1

0 *m* x *T*

El valor de *x* ((*m-1) + m · (((4*log2(n-1)+20+4*log2(n)))/3)-1*) es el valor limite donde el método naive y el método de Barret tardan lo mismo en reducir. Por partes la ecuación es: el último valor que puede tomar el módulo más el módulo tantas veces como la cantidad de iteraciones que puede realizar el módulo naive antes de que termine Barret Por tanto se ha aumentado la velocidad en estos casos.

Capítulo 9 - Resultados experimentales

En este capítulo se hace una recopilación de todos las implementaciones en hardware así como las estadísticas más comunes de área y tiempos. Este capítulo sirve de referencia de los capítulos anteriores para las resoluciones y toma de decisión. Las implementaciones están realizadas para los anchos de bus más comunes 32, 64 y 128 bits menos para el módulo de multiplicación modular final que está para un ancho más,16 bits. Estas implementaciones se han realizado mediante el programa de Xilinx ISE 14.2, usando una Virtex5 como FPGA de base.

9.1 Sumadores

En el capítulo de sumadores (Capítulo 4) se comparan los 3 algoritmos de sumas escogidos para comprobar cuál es el mejor para su futura implementación. Los módulos son sumador paralelo con acarreo en serie (RCA), sumador con anticipación del acarreo (CLA) y Kogge-Stone. Estos son los resultados de las implementaciones obtenidos para los anchos de banda antes mencionados.

9.1.1 Resultados de implementación

Sumador	k	LUT's	Retardo
RCA	32	48	9.241 ns
	64	96	18.042 ns
	128	194	36.187 ns
CLA	32	53	8.239 ns
	64	125	16.040 ns
	128	251	30.642 ns
Kogge-Stone	32	121	6.174 ns
	64	329	7.846 ns
	128	896	8.266 ns

Tabla 3: Comparación Sumadores

Como se puede apreciar el sumador de Kogge-Stone es el más rápido de los 3 presentados. Esto es debido a que el cálculo del acarreo se resuelve de forma escalonada y por niveles. En el sumador CLA, aunque también lo resuelve por niveles, la lógica que con lleva es mucho mayor ya que no tiene resultados parciales como en Kogge-Stone. En RCA el acarreo tiene que pasar por todos los sumadores completos para terminarse de ahí que sea el algoritmo más lento. Por otra parte, el algoritmo de Kogge-Stone es el algoritmo que más ocupa en área con respecto a los anteriores. Esto es debido (como se vio en el capítulo 4) a que para generar los bits de generación y propagación utiliza resultados parciales que se van resolviendo en etapas anteriores, cosa que con los demás algoritmos no ocurre porque no necesitan de resultados parciales. Se observa además que el aumento de área en Kogge-Stone tiene un comportamiento muy parecido al aumento de retardo en las opciones secuenciales y con aceleración. El crecimiento en ambos casos parece exponencial. En el caso contrario el comportamiento es mucho más suave.

Como se explicó al principio el objetivo de este trabajo es incrementar la velocidad del diseño. Por estas razones, se ha escogido el sumador de Kogge-Stone como componente básico para las sumas. A continuación se muestra a modo de gráfico la velocidad para los 3 algoritmos en los 3 anchos de banda estudiados. De esta forma se puede ver de manera más clara la diferencia.

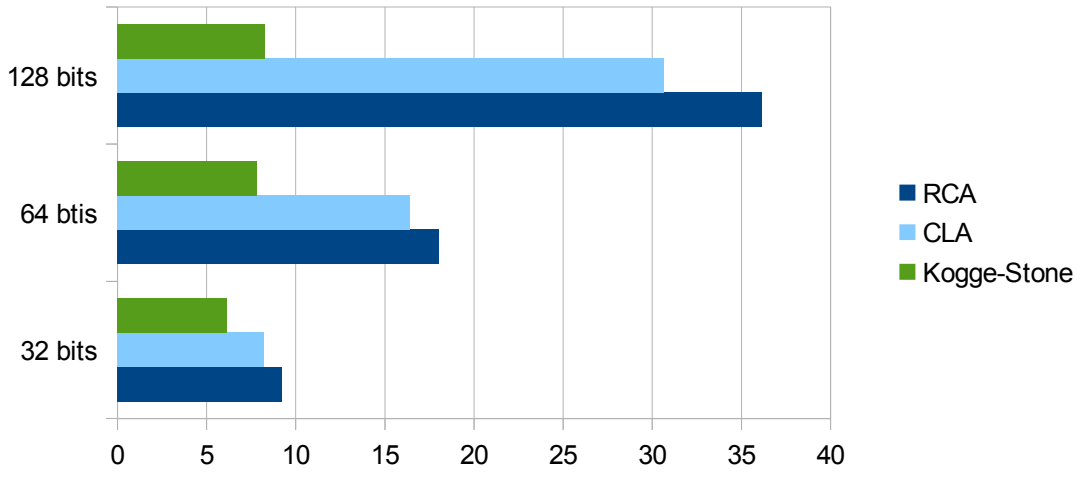


Figura 15: Retardo en ns de los sumadores

9.2 Multiplicadores

En el capítulo de multiplicadores (capítulo 5) se comparan los 3 algoritmos de multiplicación escogidos para comprobar cual es el mejor para su futura implementación. Los módulos de multiplicación son: sumas y desplazamientos, Karatsuba-Ofman y Booth. Estos son los resultados de las implementaciones obtenidos para los anchos de banda antes mencionados.

9.2.1 Resultados de la implementación

Multiplicador	k	LUT's	Flip-Flop's	Periodo	Ciclos	Tiempo total
Sumas y desplazamientos	32	148	176	2.739 ns	32	87.648 ns
	64	274	332	2.929 ns	64	187.456 ns
	128	534	652	4.657 ns	128	596.093 ns
Karatsuba-Ofman (Caso base = 8x8)	32	1002	150	4.425ns	9	39.825 ns
	64	5165	767	5.165ns	13	67.145 ns
	128	22647	2263	6.461ns	17	109.837 ns
Booth	32	255	267	3.097 ns	48	306.92 ns
	64	549	523	3.543 ns	96	340.128 ns
	128	1013	1037	4.261 ns	192	818.112 ns

Tabla 4: Comparación multiplicadores

Como se puede apreciar el multiplicador de Karatsuba no es el que permite un reloj más rápido de los 3 presentados, pero es el que menos ciclos de reloj necesita para terminar la operación. Nuestro algoritmo de recursivo de Karatsuba tiene como caso base una multiplicación combinacional que dura 1 sólo ciclo de reloj. El sumador CSA que implementa también se realiza en 1 solo ciclo. Esos son los motivos por le cual el periodo es tan grande. En Booth y en sumas y deplazamientos el periodo lo dicta lo que tarda en realizarse la resta o la suma respectivamente ya que también se realizan en 1 ciclo. El algoritmo de Karatsuba el algoritmo que menos tiempo (total) necesita para terminar la operación con anchos de bus mayores que 8 gracias a que los ciclos dependen del $\log_2$ del ancho y no del ancho en si de las entradas. La multiplicación de Karatsuba-Ofman ofrece una velocidad aceptable frente a los demás mientras

que es excepcionalmente modular y se pueden generar versiones de la misma para un determinado ancho de bus de forma rápida.

Como contrapartida, el multiplicador de Karatsuba es el que más área necesita. Esto es en parte por culpa de la recursión ya que se implementa en cada multiplicador 4 multiplicadores iguales con una entrada menor. Los demás algoritmos al ser iterativos no tienen este problema, pues utilizan el mismo componente para iterar una y otra vez en si mismo. Una vez más el objetivo del trabajo es la velocidad ante el área. A continuación se muestran a modo de gráfico la velocidad en el caso peor para los 3 algoritmos en los 3 anchos de banda estudiados. De esta forma se puede ver de manera más clara la diferencia. Mencionar que los ciclos utilizados en el algoritmo de Booth corresponden a una entrada con pares de bits distintos cada 3 bits para tener un caso real.

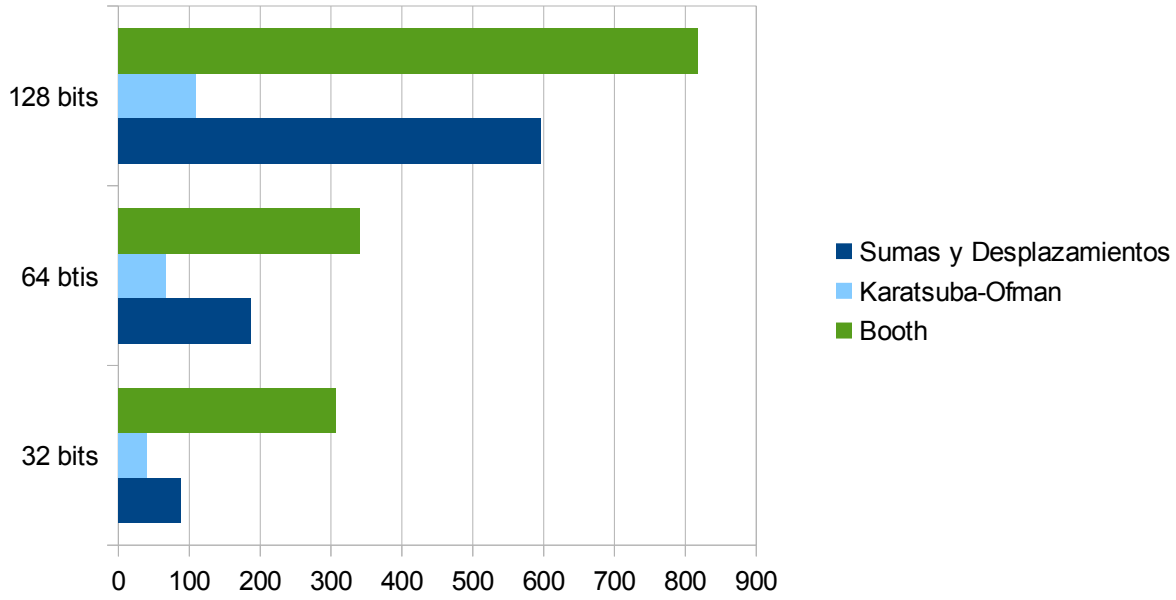


Figura 16: Tiempo total en ns de los multiplicadores

9.2.2 Simulaciones

En este apartado se muestran una serie de capturas de la herramienta ISim que permiten ver una simulación de los sistemas anteriormente vistos. Estos sistemas al ser secuenciales, se

puede comprobar el número de ciclos de reloj que se necesitan para una determinada operación. Como muestra del funcionamiento del componente descrito en VHDL, se muestra el mismo caso de ejemplo de simulación con un ancho de 8 bits.

- Algoritmo de sumas y desplazamientos

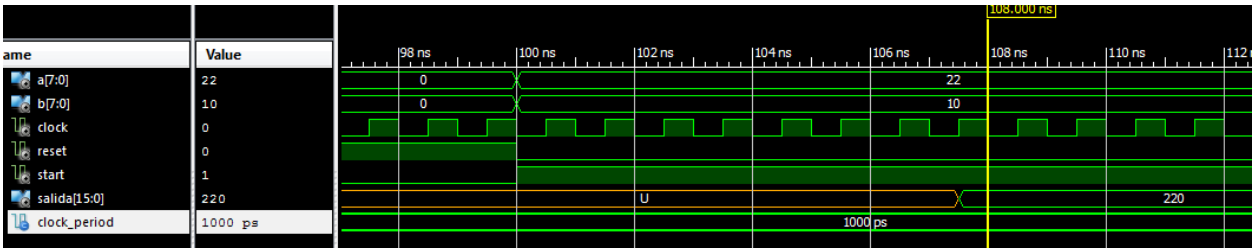


Figura 17: Simulación de sumas y desplazamientos

La simulación del primer multiplicador muestra como una multiplicación de 8 bits tarda 8 ciclos de reloj. Estos corresponden al número de bits de ancho de las entradas.. La frecuencia del reloj en esta simulación no corresponde a la frecuencia que soporta el componente sino que se ha reducido para poder mostrarlo.

- Algoritmo de Karatsuba-Ofman

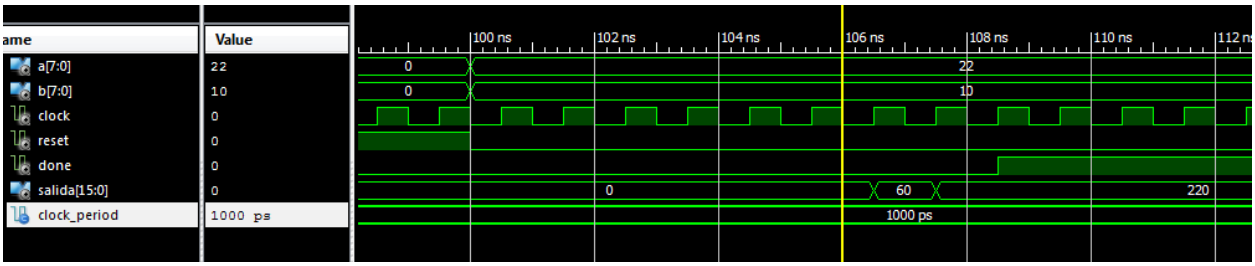


Figura 18: Simulación de Karatsuba-Ofman

La simulación del multiplicador de Karatsuba-Ofman muestra como una multiplicación de 8 bits que tarda 9 ciclos de reloj. Siguiendo la formula (capítulo 5) (4*1)+5 que corresponde al número de ciclos del caso base más 4 por el número de niveles que tiene que hacer la recursión (1 en este caso ya que el caso base es 2x2 bits). Esto corrobora la expresión vista en el capítulo 5 $4*(\log_2 n)-2)+5$. La frecuencia del reloj en esta simulación no corresponde a la frecuencia que soporta el componente sino que se ha reducido para poder mostrarlo.

- Algoritmo de Booth

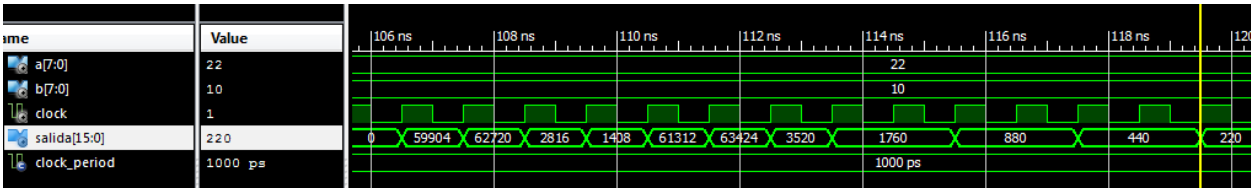


Figura 19: Simulación de Booth

La simulación del multiplicador de Booth muestra como multiplicación de 8 bits que tarda 12 ciclos de reloj. Estos corresponden a la ecuación vista (capítulo 5) $2+x+2y$ y el número 22 en binario $(00010110)_2$. La ecuación con los datos quedaría de la siguiente forma: 2+4+2*3. La frecuencia del reloj en esta simulación no corresponde a la frecuencia que soporta el componente sino que se ha reducido para poder mostrarlo.

9.2.3 Resultados de recursión Karatsuba-Ofman

El módulo utilizado del algoritmo de Karatsuba-Ofman puede ser configurado para tener distintos casos bases. Esto significa que se puede definir el ancho de la última multiplicación de la recursión para comprobar qué ancho es el optimo en bits para el caso base teniendo en cuenta el número de ciclos y el tiempo de ciclo usado. El estudio de los casos base ha dado como resultado la siguiente tabla:

Multiplicador	k	LUT's	Flip-Flop's	Periodo	DSP48SLICES	Ciclos	Tiempo total
Karatsuba-Ofman (Caso base = 2x2)	32	3838	1343	5.721 ns	-	17	97.24 ns
	64	16332	6298	7.364 ns	-	21	154.56 ns
	128	74615	27669	7.777 ns	-	25	194.42 ns
Karatsuba-Ofman (Caso base = 4x4)	32	3110	442	4.123ns	-	13	53.599 ns
	64	15185	1180	5.104ns	-	17	86.768 ns
	128	54913	4759	6.460ns	-	21	135.66 ns
Karatsuba-Ofman (Caso base = 8x8)	32	1002	150	4.425ns	-	9	39.825 ns
	64	5165	767	5.165ns	64	13	67.145 ns
	128	22647	2263	6.461ns	256	17	109.837 ns

Tabla 5: Comparación casos base Karatsuba-Ofman

Como se puede observar utilizando un caso base pequeño obtenemos unos periodos de reloj realmente buenos comparados con los otros. Por otra parte necesita de mucha más área que ninguno ya que tiene que instanciar más veces la multiplicación recursiva. El periodo es tan bueno en el caso base más bajo que mejora los tiempos totales incluso usando más ciclos de reloj que ningún otro. Por otra parte el uso de un caso base con una multiplicación de 8 bits tiene un periodo muy similar al caso base anterior (4 bits) y utiliza menos ciclos. Se puede observar también que a medida que se sube el caso base los tiempos mejoran no por el periodo (que también mejora pero poco), sino por el ahorro de ciclos en las recursiones que ya no se realizan. Utilizaremos el caso base de 8 bits debido a limitaciones de memoria que presenta la máquina donde se está compilando el componente. Al utilizar menos componentes en la recursión la memoria que requiere para compilar un módulo que tenga un multiplicador es mucho menor que utilizando un multiplicador que requiera muchos componentes en la recursión.

9.2.4 Comparación con otros autores

En este apartado se comparan los resultados obtenidos con los resultados de otros autores. Los autores que se comparan en esta sección son [44]. Los resultados de la comparación son los siguientes utilizando una Virtex5 igual que los demás autores.

Multiplicador	k	LUT's	Flip-Flop's	Periodo	Ciclos	Tiempo total
Karatsuba-Ofman 2x2	8	167	72	2.89 ns	9	26.01 ns
	16	869	317	4.25 ns	13	55.25 ns
	32	3838	1343	5.72 ns	17	97.24 ns
	64	16332	6298	7.364ns	21	154.56 ns
Booth Combinacional [44]	8	188	-	13.49 ns	-	13.49 ns
	16	628	-	25.31 ns	-	25.31 ns
	32	2276	-	49.09 ns	-	49.09 ns
Booth secuencial radix-4 [44]	8	58	25	2.90 ns	9	26.10 ns
	16	135	42	3.12 ns	17	53.0 ns
	32	248	74	4.22 ns	33	139.1 ns
	64	473	140	4.22 ns	65	274.0 ns

Tabla 6: Comparación multiplicadores con otros autores

Se puede observar en la tabla 6 que nuestro multiplicador de Karatsuba con un caso base de 2 bits obtiene unos buenos tiempos respecto al algoritmo de Booth secuencial dado en [42], utilizando muchos menos ciclos aunque más tiempo de ciclo. La versión combiancional es vencida en tiempos usando un caso base más alto como se puede comprobar en la tabla 5. Esto es debido principalmente a que no tiene que usar tantas veces la recursión y se ahorran muchas operaciones y ciclos. Usando un caso base de 4 o de 8 bits para la multiplicación se consiguen unos tiempos similares y de mejora respectivamente.

El área en LUT's y FF (Flip-Flop's) sigue siendo uno de los puntos flacos del algoritmo, ya que ocupa mucho más que sus rivales. Esto sigue ocurriendo incluso subiendo el caso base como se puede apreciar en los resultados de la tabla 5. La razón es la misma que la que da la velocidad, la recusión. Si se utiliza la recursión se necesita más área ya que necesita más componentes pero la velocidad aumenta ya que divide el problema en problemas más pequeños.

Los valores de las celdas con "-" son debidos a que los componentes combinacionales no tienen periodo ni usan componentes de tipo Flip-Flops. En su lugar en el tiempo de periodo se ha colocado el tiempo de delay que necesitan para terminar la operación y el tiempo total es el mismo.

9.3 Reducciones

En el capítulo de reducciones (capítulo 6) se comparan los 3 algoritmos de reducción escogidos para comprobar cual es el mejor para su futura implementación. Los algoritmos son la reducción naive, la reducción de Barret (sin el cálculo de μ) y la multiplicación modular Montgomery. Se ha optado por no incluir el cálculo de μ en el algoritmo de Barret para obtener el mismo nivel de ventaja sobre el algoritmo de Montgomery ya que está implementado para operaciones módulo 239. Estos son los resultados de las implementaciones obtenidos para los anchos de banda antes mencionados.

9.3.1 Resultados de la implementación

Reducción	k	LUT's	Flip-Flop's	Periodo	Ciclos	Tiempo total
Naive	32	724	117	4.266 ns	*A/m*	*A/m · Periodo*
	64	1695	208	5.133ns	*A/m*	*A/m · Periodo*
	128	3616	424	6.036ns	*A/m*	*A/m · Periodo*
Barret	32	7844	23897	4.611 ns	56	258.216 ns
(Sin cálculo μ)	64	30229	8169	5.083 ns	60	304.980 ns
(Con cálculoμ)	32	8820	2228	4.697ns	571	2681.987 ns
Montgomery	32	293	107	1.397ns	52	72.644 ns
	64	634	271	1.971ns	100	197.100 ns
	128	1277	529	2.835ns	196	555.66 ns

Tabla 7: Comparación Reductores

La reducción de Barret ofrece un número bastante bueno con respecto a los demás módulos. En una comparación justa, con Barret calculando la constante μ y con Montgomery sin optimizar para ningún módulo Barret obtiene una velocidad con anchos de banda bajos (entre 0 y 1024 bits) bastante buena frente a otros algoritmos como el de Montgomery que permanecen casi constantes [33]. El motivo por el cual usar el algoritmo de Barret como reducción no es otro

que la versatilidad en la cantidad de módulos que puede realizar. Además es el algoritmo que necesita (de media) menos ciclos. El algoritmo naive depende de la entrada *A* y del valor del módulo *m*, lo cual lo hace inestable y el algoritmo de Barret vence en este campo a Montogmery. Si tenemos en cuenta que muy posiblemente quien dicte el valor del periodo sea la multiplicación en el módulo final, al final el algoritmo de Montogmery obtendrá peores tiempos. Esas son las razones por la cual se escogió el algoritmo de Barret a Montomery aun siendo mejor este último. La razón por la cual en Barret no hay datos para 128 bits es que la máquina que compila el diseño se queda sin memoria RAM a causa de la recursión que produce Karatsuba-Ofman. En una implementación con el cálculo de μ vemos se como dispara la cantidad de ciclos necesarios para terminar. La causa es la división entera de 64 bits que tiene que realizar.

Por otra parte es interesante el poder explotar los beneficios que ofrece el método naive. Cuando la entrada y el valor del módulo toman un valor muy próximo entre sí, es mucho más rápido que cualquier otro algoritmo. En cuanto el área ocupada, el método de Montgomery vuelve a ser quien ofrece mejores resultados, al igual que con la velocidad. Esto es porque las operaciones que se han implementado no son de carácter general y si especializados en un módulo *m*.

9.3.2 Simulaciones

En este apartado se muestran una serie de capturas de la herramienta ISim que permiten ver una simulación completa de los sistemas anteriormente vistos. Estos sistemas al ser secuenciales, se puede comprobar el número de ciclos de reloj que se necesitan para completar una determinada operación. Como muestra del funcionamiento del componente descrito en VHDL, se muestra el mismo caso de ejemplo de simulación con un ancho de 16 bits.

- Algoritmo naive

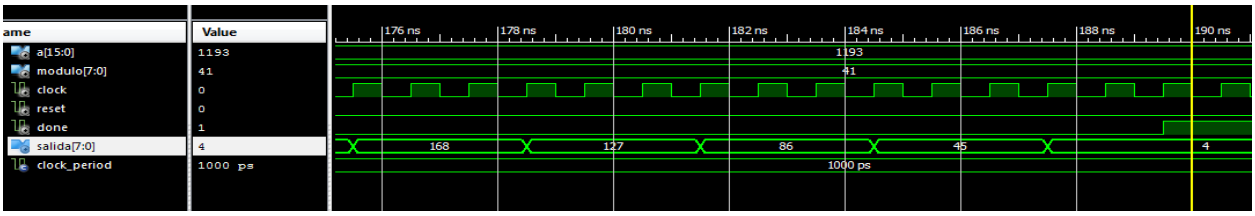


Figura 20: Simulación de la reducción naive

La simulación del componete de la reducción naive no es esclarecedora en cuanto al número de ciclos que necesita. Depende siempre de los datos de entrada y se puede obtener con la ecuación [0,*A/m*] donde *A* es el valor de la entrada y *m* el valor del módulo. En este caso ha necesitado 87 ciclos para hacer la reducción. La frecuencia del reloj en esta simulación no corresponde a la frecuencia que soporta el componente sino que se ha reducido para poder mostrarlo.

- Algoritmo de Barret

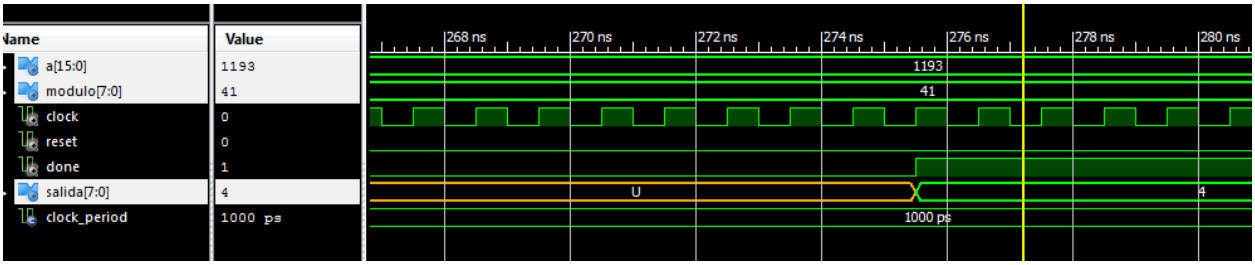


Figura 21: Simulación reductor de Barret

La simulación muestra como para una operación de módulo, con el cálculo de μ, hacen falta 177 ciclos de reloj. Otra simulación posterior con μ sin pre-calcular dio el resultado en sólo 48 ciclos. Si tenemos en cuenta que para ello han hecho falta 2 multiplicaciones, una de 16 bits y otra de 32 bits, el número de ciclos es bueno. El número de ciclos viene dado por la ecuación (4*log2(n-1)+20+4*log2(n)) sin el cálculo de μ, y $(2^{\log2(n)}+1)*6+3$ ciclos más con el cálculo de esta constante.

- Algoritmo de Montgomery

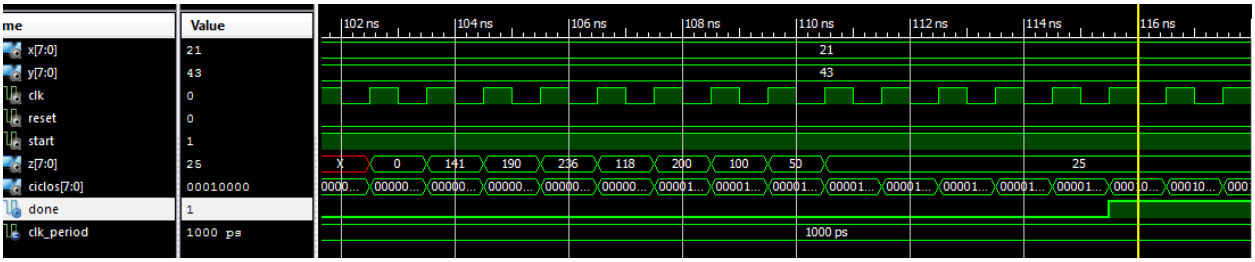


Figura 22: Simulación reductor de Montgomery

La simulación del componente muestra como en 16 pulsos puede hacer la operación $A \cdot R^{-1} \bmod m$ donde m es 239, $A = x \cdot y = 1008$ y $R = 1024$. El componente usado esta diseñado y optimizado para ese valor de módulo, con las mejoras pertinentes como divisiones y multiplicaciones por dicho valor. El resultado aunque bueno, sólo permite ese tipo de módulo y lo que se busca en el proyecto final es de carácter más general.

9.4 Módulo preparación

En el capítulo del módulo de preparación (capítulo 7) se muestra un módulo adicional a la multiplicación para ofrecer mejores tiempos en el cálculo de una multiplicación modular. Estos son los resultados de las implementaciones obtenidos para los anchos de banda antes mencionados.

9.4.1 Resultados de la implementación

Módulo	k	LUT's	Flip-Flop's	Periodo	Ciclos	Tiempo total
Preparación	32	533	72	1.913 ns	3	5.739 ns
	64	1307	208	1.952 ns	3	5.856 ns
	128	3269	264	1.983 ns	3	5.949 ns

Tabla 8: Estadísticas preparación

Difícilmente se pueden mejorar los resultados obtenidos en este módulo. Aparte de simple, está construido con los, posiblemente, mejores algoritmos para la causa que existen. El número de ciclos es irreducible a menos que se haga combinacional ya que se necesita 1 ciclo para la resta/comparador, 1 ciclo para la salida y otro para la entrada. El tiempo de periodo lo dicta el elemento más lento el comparador. El comparador que se usa no es un comparador sencillo de igual o distinto. Se necesitaba para el algoritmo un comparador que tenga como señales de salida el mayor, menor o igual, lo que complica la lógica y por tanto el tiempo que necesita para terminar la operación.

9.4.2 Simulación

En este apartado se muestran una captura de la herramienta ISim que permite ver una simulación completa del componente antes mencionado. Este sistema al ser secuencial, se puede comprobar el número de ciclos de reloj que se necesita para terminar una determinada operación. Como muestra del funcionamiento del componente descrito en VHDL, se muestra el mismo caso de ejemplo de simulación con un ancho de 8 bits.

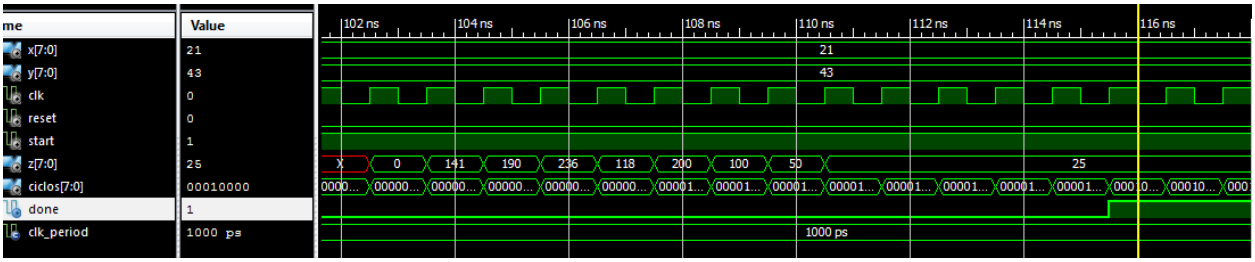


Figura 23: Simulación preparación

La simulación del componete nos informa de lo rápido que termina el módulo en operar. Sólo 3 ciclos de reloj contando el último ciclo de terminado y el de comienzo. La mejora de la resta usando de base el sumador de Kogge-Stone, la paralelización de las restas y de las comparaciones han podido lograr tan bajo resultado.

Dado que se necesitara para futuras implementaciones, se ha añadido una nueva salida "CambioAB". Esta salida ofrece información de que variable se ha modificado y cual no. Este cambio no afecta ni a la velocidad ni al algoritmo anteriormente visto.

9.5 Módulo multiplicador modular

En el último capítulo de implementaciones (Capítulo 8) se implementa un módulo de multiplicación modular con la mejora añadida. Estos son los resultados de las implementaciones obtenidos para los anchos de banda antes mencionados. Todas las implementaciones salvo en la comparación final están siendo implementadas sin el cálculo de la constante μ para el módulo de Barret.

9.5.1 Resultados de la implementación

Módulo	k	Slices de LUT	Flip-Flops	Periodo	Ciclos	Tiempo total
Multiplciador modular (arquitectura 2)	16	4772	2403	4.053ns	[17-67]	[69-272]ns
	32	18276	7808	4.791ns	[21-71]	[101-340]ns
	64	65539	24938	5.583 ns	[25-75]	[139-418]ns

Tabla 9: Resultados del módulo de multiplicación modular sin el cálculo de μ

En este apartado se ha mostrado la implementación del componente de la multiplicación modular acelerada. Además se ha conseguido mejorar la arquitectura (Capítulo 8) para que evite esperas innecesarias en cálculos para proveer el resultado mucho más rápido. Gracias al uso del reductor naive el multiplicador puede dar el resultado de la operación antes de que termine el módulo de reducción de Barret, mejorando la velocidad de respuesta. En los capítulos siguientes se comparan estos resultados con los obtenidos por otros autores. Como se puede ver el periodo en este caso es muy superior a Montgomery si se hubiera escogido ese reductor el número de ciclos hubiera aumentado y por tanto aumentado el tiempo total de la multiplicación.

El rango de valores dado en las columnas de "Ciclos" y "Tiempo total" mostrados en la tabla 9 viene dado por la mejora de la arquitectura 2 usando el módulo de reducción naive. El producto de las entradas *A* y *B* puede dar valores muy próximos a *m,* si se da ese caso, el número de ciclos viene dado por el reductor naive y no por el reductor de Barret. Los valores extremos mínimo y máximo en el número de ciclos son: los casos donde no hace falta reducir y el tiempo que tarda el reductor de Barret en dar el resultado respectivamente. La columna de "Tiempo total" es el resultado de la multiplicación truncada de las columnas "Periodo" y "Ciclos".

9.5.2 Simulaciones

En este apartado se muestran una captura simulación completa del componente antes mencionado. Este sistema al ser secuencial, se puede comprobar el número de ciclos de reloj que se necesita para terminar una determinada operación. Como muestra del funcionamiento del componente descrito en VHDL, se muestra el mismo caso de ejemplo de simulación con un ancho de 8 bits de entradas para la multiplicación.

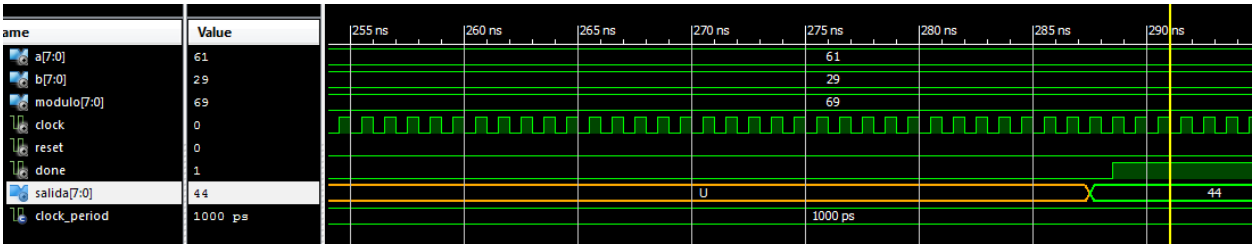


Figura 24: Simulación multiplicador modular arquitectura 1

Vemos como hace la multiplicación modular en lo que tarda en hacer la multiplicación y la reducción, 188 ciclos. A continuación, para comprobar la mejora, se muestra la simulación de la segunda arquitectura mejorada con las reducciones paralelas.

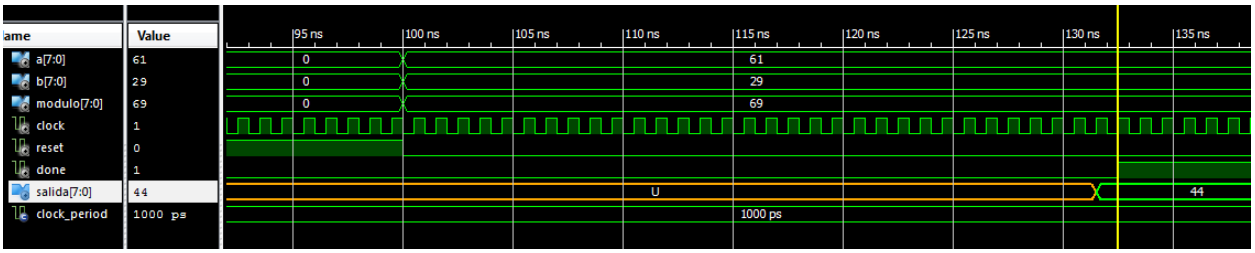


Figura 25: Simulación multiplicador modular arquitectura 2

Como se puede apreciar en esta nueva arquitectura el mismo caso tarda solamente 33 ciclos. La diferencia con la anterior arquitectura es de 155 ciclos. Esto es debido principalmente a que el módulo de Barret tiene que calcular μ y posteriormente hacer sus cálculos. Con esta nueva arquitectura solo esperamos esos ciclos únicamente si es necesario. Este caso ocurrirá bastantes veces gracias a la mejora de la multiplicación modular. De no estar presente este módulo la primera simulación hubiera tardado 3 ciclos menos, es decir, 185 ciclos. Una cantidad que se sigue alejando mucho de los 33 ciclos que tarda esta nueva arquitectura. De esta forma, la cantidad de ciclos que se usa para la realización de la multiplicación modular donde n es el número de bits de la entrada C es el producto de A y B que son las entradas al módulo de multiplicación y m el valor del módulo, es la siguiente:

- $4*log_2(n-2)+8$ cuando la multiplicación da como resultado un valor sin reducir

- $4*log_2(n-2)+8+A/m$ cuando el módulo naive termina antes que el módulo de Barret(sin el cálculo de μ)
- $(28+4*log_2(n)+4*log_2(n-1)+4*log_2(n-2))$ cuando el método de Barret (sin el cálculo de μ) termina antes que el método naive.

9.5.3 Comparación con otros autores

En este apartado se comparan los resultados obtenidos con los resultados de otros autores. Los autores que se comparan en esta sección son [42] con los algoritmos de "M*ultiply and Reduce", "Shift and Add"* y *"Montgomery"* y únicamente usando la multiplicación de Karatsuba y la reducción de Barret de este proyecto. Los resultados de la comparación son los siguientes utilizando una Virtex5 igual que los demás autores.

Multiplicador	k	Slices	Flip-Flops	Periodo	Ciclos	Tiempo total
Multiplicación modular (arquitectura 2)	16	4772	2403	4.053ns	[12-70]	[48.6-283.7] ns
	32	18276	7808	4.791ns	[16-74]	[76.6-354.5] ns
	64	65539	24938	5.583 ns	20-78	[111.6-435.4] ns
Multiplicación modular (arquitectura 1)	16	4048	2286	4.053ns	12 ó 70	48.6 ó 283.7 ns
	32	16581	7601	4.791ns	16 ó 74	76.6 ó 354.5 ns
	64	62923	24509	5.583 ns	20 ó 78	111.6 ó 435.4 ns
Karatsuba + Barret (este proyecto)	16	3987	1987	4.051ns	9 ó 67	36.1 ó 271.5 ns
	32	16483	7024	4.789ns	13 ó 71	62.2 ó 340.0 ns
	64	61847	21228	5.580 ns	17 ó 75	94.8 ó 418.5 ns
"Multiply and reduce"[42]	16	72	124	44,6 ns	32	1427.2 ns
	32	126	237	59.1 ns	64	3782.4 ns
"Shift and add" [42]	16	63	70	78.7 ns	16	1259.2 ns
	32	119	135	140.8 ns	32	4505.6 ns
Montgomery [42]	16	59	56	38.9 ns	16	622.4 ns
	32	108	105	40.9 ns	32	1308.8 ns
Multiplicación modular (arquitectura 2 sin μ)	16	5212	2734	4.053ns	[12-330]	[48.6-1337.4] ns
	32	21485	9012	4.791ns	[16-645]	[76.6-3090] ns

Tabla 10: Comparación multiplicadores modulares

Podemos comprobar que el algoritmo presentado en este trabajo funciona muy bien frente a los multiplicadores modulares de otros autores. Esto es posible gracias al conjunto de varios factores. El uso de componentes rápidos como Karatsuba o Kogge-Stone, el uso de estrategias de aceleración como el módulo de preparación y el uso de elementos en paralelo como el uso del reductor Naive y el reductor de Barret a la vez.

Se puede comprobar que aunque se utilicen solamente Karatsuba y Barret (sin utilizar el módulo de preparación) los tiempos son muy parejos. Pero el rango de valores es distinto. Mientras que solo con Karatsuba y Barret son dos valores muy distintos y discretos, con la multiplicación modular que se ha descrito en este trabajo se recorre todo un rango de valores en los cuales el módulo de Karatsuba y Barret daría el tiempo más alto. Es cierto que los valores máximos y mínimos del tiempo para realizar el módulo de Karatsuba y Barret son superiores en 3 ciclos (lo que tarda el módulo de preparación), el algoritmo con la arquitectura 2 tiene una media de tiempo mucho más baja.

Con respecto al resultado de otros autores se puede observar que, en una comparación justa con "*Multiply and reduce*" y nuestro algoritmo con el cálculo de la constante μ (para no tener ventajas) es mejor nuestro algoritmo, no en ciclos debido a la división, pero si en tiempo total y en periodo. Además, se puede observar que a anchos de entradas más grandes la comparación "*Shift and add*" y nuestro módulo de multiplicación modular el tiempo diverge en favor de nuestro algoritmo.No hay que olvidar que el módulo de Montgomery en [42] solo sirve para m impares y por tanto no calcula R ni R^{-1}, debido a que el *mcd* entre *R* y *m* siendo *m* impar y *R* una potencia de 2 es 1. Entonces las divisiones y los módulos se verían reducidos a desplazamientos y a sesgos. No sería justo compararlo con la multiplicación calculando la constante μ dado que Montgomery parte con estas ventajas. Si lo comparamos con la arquitectura sin calcular la constante μ es mejor en tiempos y dependiendo del valor del producto de *A* y *B* lo es en ciclos.

El rango de valores dado en las columnas de "Ciclos" y "Tiempo total" mostrados en la tabla 10 viene dado por la mejora de la arquitectura 2 usando el módulo de reducción naive. El producto de las entradas *A* y *B* puede dar valores muy próximos a *m,* si se da ese caso, el número

de ciclos viene dado por el reductor naive y no por el reductor de Barret. Los valores extremos mínimo y máximo en el número de ciclos son: los casos donde no hace falta reducir y el tiempo que tarda el reductor de Barret en dar el resultado respectivamente. La columna de "Tiempo total" es el resultado de la multiplicación truncada de las columnas "Periodo" y "Ciclos".

El par de valores separados por "ó" dado en las columnas de "Ciclos" y "Tiempo total" mostrados en la tabla 10 viene dado por los casos donde no hace falta reducir y el tiempo que tarda el reductor de Barret en dar el resultado respectivamente. La columna de "Tiempo total" es el resultado de la multiplicación truncada de las columnas "Periodo" y "Ciclos".

Capítulo 10 - Conclusiones y trabajo futuro

En este capítulo se detallan las conclusiones finales y el trabajo futuro relacionado con el proyecto.

10.1 Conclusiones

En este trabajo se ha recogido información sobre diferentes componentes para crear un multiplicador modular. Se han estudiado sumas, multiplicaciones y reducciones y se han visto cuales eran los que cumplían mejor con las expectativas de velocidad que se perseguían. Dentro de la multiplicación, se ha realizado una nueva implementación recursiva de Karatsuba-Ofman que proporciona unos buenos resultados en comparación con los resultados obtenidos con otros autores. Después de la selección se procedió a la construcción del módulo de preparación de los operandos para la multiplicación modular. El módulo de preparación modular es uno de los encargados de dar la velocidad al multiplicador. Otro de los componentes que mejoran la velocidad es el uso en paralelo de los dos reductores (naive y Barret). Gracias a esta implementación paralela se pueden dar resultados en poco tiempo sin tener que esperar a que termine la operación de reducción de Barret.

Son gracias a estos pilares, componentes rápidos, utilización de nuevas estrategias modulares y uso de estrategias en paralelo, por los cuales se han obtenido unos resultados muy buenos en el objetivo que se buscaba. La velocidad no solo ha venido dada por un número reducido de ciclos necesarios para el multiplicador modular de este trabajo sino también por el tiempo de ciclo que necesita para su correcto funcionamiento. Dado que muchas de las implementaciones con fines criptográficos usan módulos primos, se puede pensar que el algoritmo de Montgomery, ya que los primos son impares se puede elegir una variable R como potencia de 2 para evitar divisiones y módulos complejas, ofrece más velocidad que el estudiado. Este problema se puede evitar, por ejemplo, con un conjunto de registros donde estén alojados los valores de μ para diversos valores de m. De esta forma se puede dar el valor de μ sin tener que calcular y realizar el cálculo de la reducción de Barret de forma rápida.

El multiplicador modular presentado, tiene un punto débil, el área. Sin entrar en un estudio profundo sobre el tema, en el análisis de cualquier elemento el más rápido era el que más área consumía. Este hecho hace que no sea posible (en ciertos casos) implementar el diseño de este trabajo o que el tiempo para poder implementarlo sea algo elevado.

10.2 Trabajo Futuro

En el desarrollo del trabajo han ido surgiendo varias ideas de mejora o posibles estudios para un futuro. Estas ideas han sido descartadas principalmente por la longitud del proyecto o porque afectan de forma tangencial al objetivo principal del mismo. Son varias las opciones que pueden desarrollarse en un futuro tomando como referencia este trabajo.

- Estudios para la mejora en área y consumo: Este trabajo está enfocado principalmente a la optimización de la velocidad de la multiplicación modular sin tomar en cuenta el área o el consumo. Una futura variante sería estudiar el impacto sobre el área y el consumo de dicho componente y de los módulos que lo integran. En este caso se podrían utilizar, por ejemplo, en la multiplicación de Karatsuba-Ofman y una mejora en el caso base.

- Especialización: Otra posible vía de mejora puede que sea la especialización del componente para un determinado módulo. Es decir, que la multiplicación sea módulo un número fijo. Actualmente está desarrollado para cualquier valor del módulo. La revisión podría darse cambiando los elementos y optimizarlos para un determinado valor, por ejemplo el cálculo de μ en la reducción de Barret.

- Implementación del componente en un módulo criptográfico: Tanto del desarrollo del mismo como el estudio y comparación con otros módulos para el caso.

Referencias y Bibliografía

[1] W. Die and M. E. Hellman. New directions in cryptography. IEEE Transactions on information Theory Vol. IT-22 nº6. 1976.

[2] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. Hewlett-Packard Labs. 1985.

[3] Victor S. Miller. Use of elliptic Curves in cryptography. Exploratory Computer Science, IBM Research. 1985.

[4] Xianjin Fang, Longshu li. On Karatsuba Multiplication Algorithm. Chengdu, pages 274-276 2007

[5] Chin-Bou Liu, Chua -Huang Huang, Chin-Laung Lei. Design and Implementation of Long-Digit Karatsuba's Multiplication Algorithm Using Tensor Product Formulation. Departament of Information Engineering and Computer Science. 2003

[6] Hugo Izqueirdo Donoso. Implementación hardware de algoritmos criptográficos para rfid. Universidad Carlos III. 2009.

[7] Karl Fiedrich Gauss. Disquistiones Artihmeticae. Lipsiae.1801

[8] Tom M. Apóstol. Modular functions and Dirichlet series in number theory. Springer-Verlag, 1990

[9] IEEE P1363, Standard Specifications for Public Key Cryptography.

[10] A. Karatsuba, Yu. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. Proceedings of the USSR Academy of Sciences. 1962

[11] D. Knuth. The Art of computer Programming. A RISC Computer for the New Millennium Volume 2. 1997

[12] Guillermo Lopez. Karatsuba algorith: fast multiplication of large integers. MIT. 2009

[13] Jean-Pierre Deschamps, José Luis Imaña, Gustavo D. Sutter .Hardware implementation of finite-field Arithmetic. McGraw Hill. 2009.

[14] André Weimerskirch, Christof Parr. Generalizations of the Karatsuba Algorithm for Efficient Implementations. IACR Cryptology. 2006.

[15] P.D. Barret .Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. Computer Securuty LTD. Oxon. 1986

[16] N. Koblitz. Elliptic curve cryptosystems. Mathematics of Computation Vol 48. nº177. 1987.

[17] J. Lopez and R. Dahab. Fast multiplication on elliptic curves over GF(2m) without precomputation. Springer. 1999

[18] D. Hankerson, J. Lopez-Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. Auburn University, University of Valle, University of Waterloo. 2000

[19] G. Orlando and C. Paar. A scalable GF(p) elliptic curve processor architecture for programmable hardware. General Dynamics Comminucation Systems, ECE Departemnt of Polytechnic, Worcester. 2001

[20] N. Smart. The hessian form of an elliptic curve. Cryptographic Hardware and Embedded Systems. Springer. 2001

[21] Ahmad Khaled M. Al-Kayali. Elliptic Curve Cryptography and Smart Cards. Sans intitute. 2004.

[22] An Liu, Peng Ning. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. Dept. of Comput. Sci., North Carolina State Univ., Raleigh, NC. 2007
[23] Karatsuba, A. The complexity of computations. Proceedings of the Steklov Institute of Mathematics Vol 211 pages 169-183. 1995.
[24] Serdar S. Erdem. Improving the Karatsuba-Ofman Multiplication Algorithm for Special Applications. Master Thesis of Electrical and Computer Engineering.2001
[25] Haining Fan, Jiaguang Sun, Ming Gu and Kwok-Yan Lam. Overlap-free Karatsuba-Ofman Polynomial Multiplication Algorithms. EET Information security, vol. 4, no. 1, pp. 8-14, .2010.
[26] M Ramalatha, KD Dayalan, P Dharani. High speed energy efficient ALU design using Vedic multiplication techniques. Advances in Computational Tools for Engineering Applications. 2009.
[27] P.D. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. Computer Security Ltd. Oxon.1986
[28] Tolga Acar and Dan Shumow. Modular Reduction without Pre-Computation for Special Moduli. Microsoft Research. 2010
[29] M. Johnson, B. Phung, T. Shackelford, S. Rueangvivatanakij. Modular Reduction of Large Integers Using Classical, Barrett, Montgomery Algorithms.
[30] William Hasenplaugh, Gunnar Gaubatz, Vinodh Gopal. Fast Modular Reduction. Computer Arithmetic. ARITH '07. 2007.
[31] Peter Montgomery. Modular Multiplication Without Trial Division .Mathematics of Computation, Vol. 44, No. 170. 1985
[32] C. K. Koc¸, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. Department of electrical and computer engineering and RSA laboratories. Micro, IEEE. Volume:16 , 1996.
[33] M. Johnson, B. Phung, T. Shackelford, S. Rueangvivatanakij. Modular Reduction of Large Integers Using Classical, Barrett, Montgomery Algorithms.
[34] Simon Knowles. A Family of Adders. Computer Arithmetic. Proceedings. 15th IEEE Symposium Pages 277-284. 2001.
[35] P.M. Kogge, H.S.Stone. A parallel Algorithm for Efficient solution of a General Class of recurrence Equations. Computers, IEEE Transactions on. Volume:C-22. 1973
[36] Vipul Gupta, Sumit Gupta, Sheueling Chang, Douglas Stebila. Performance analysis of elliptic curve cryptography for SSL. In Proc. 1st ACM Workshop on Wireless Security (WiSE) 2002, pp. 87–94. ACM, 2002.
[37] R.Brent and H.Kung. A regular layout for parallel adders, IEEE Transaction on Computers, vol. C-31, no.3, pp. 260-264, March 1982.
[38] Andrew D. Booth. A signed binary multiplication technique. University of Oxford. 1951
[39] D. R. Coelho. The VHDL Handbook. Kluwer Academic Publishers, Norwell, MA, USA, 1989.
[40] Certicom, Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameter. 2000
[41] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. Computing 7, pages 281-291. Springer. 1971

[42]G. Sutter, J.-P. Deschamps, E. Boemo. "Area-Time-Power of Modular Multipliers implemented in FPGA", Jornadas de Computación Reconfigurable y Aplicaciones 2004.
[43] Osama Al-Khaleel, Chris Papachristou, Francis Wolff,Kiamal Pekmestzi. FPGA-based Design of a Large Moduli Multiplier for Public-Key Cryptographic Systems. Computer Design, 2006. ICCD 2006.
[44] J.-P. Deschamps, G.D. Sutter, E. Cantó. "Guide to FPGA Implementation of Arithmetic Functions". Lecture Notes in Electrical Engineering, No. 149, Springer, 2012.

Apéndice 1. Enlaces

[1] http://www.codeproject.com/Articles/22452/A-simple-C-implementation-of-Elliptic-Curve-Crypto
[2] http://arxiv.org/ftp/arxiv/papers/1109/1109.1877.pdf
[3] http://www.doulos.com/knowhow/vhdl_designers_guide/a_brief_history_of_vhdl/

Apéndice 2. Archivo de figuras y tablas