Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Implementación en hardware de multiplicadores modulares orientados a emparejamientos bilineales

Tesis que presenta

Cuauhtémoc Chávez Corona

para obtener el Grado de

Maestro en Ciencias

en la Especialidad de

Computación

Directores de Tesis

Dr. Francisco Rodríguez Henríquez

Dr. Debrup Chakraborty

México D. F. **Julio 2012**

Índice general

Resumen

En la actualidad es cada vez más frecuente el uso cotidiano de la criptografía de llave pública, ejemplo de ello son los certificados digitales, los esquemas de votaciones electrónicas y el dinero electrónico entre otros. Estas aplicaciones tienen su sustento en algoritmos como RSA y ElGamal, o en estructuras matemáticas como las cada vez más usadas Curvas Elípticas o los relativamente recientes emparejamientos bilineales. Todos los anteriores a su vez tienen una base común, que es el producto en campos finitos, o producto modular. Debido a su importancia, mejorar esta operación es una tarea constante, que se vale de nuevos algoritmos y tecnologías para hacerla más eficiente. En este trabajo se aborda la implementación una arquitectura para realizar el producto en campos finitos utilizando variaciones del ya famoso algoritmo de Montgomery para multiplicación modular sobre dispositivo de hardware reconfigurable de la familia **Virtex 6** de **Xilinx**. Esta familia en particular cuenta con módulos empotrados conocidos como *DSP48Slices* , los cuales poseen multiplicadores muy rápidos dedicados al procesamiento de señales que son explotados para realizar productos de números grandes de manera muy rápida y obtener arquitecturas que trabajan a frecuencias superiores a los 220 Mhz, para el cálculo de productos en los campos $\mathbb{F}_p$ y $\mathbb{F}_{p^2}$, tomando 15 y 4 ciclos de reloj respectivamente, lo que las hace competitivas con las implementaciones publicadas más recientemente.

Abstract

Today is becoming often a daily use of public key cryptography, example of this are the digital certificates, electronic voting schemes and electronic money inter alia. These applications have their own basis in algorithms such as RSA and ElGamal, or mathematical structures as used increasingly elliptic curves or the relatively recent bilinear pairings. All of the above in turn have a common base, which is the product in finite fields, or modular product. Due to this importance, improve this operation is an ongoing task, which uses new algorithms and technologies by make it more efficient.

This work discusses the implementation architecture for the product in finite fields using variations of the famous Montgomery algorithm for modular multiplication on reconfigurable hardware device family **Virtex 6** from **Xilinx**. This particular family has built modules known as *DSP48Slices*, which have very fast multipliers dedicated to signal processing and are exploited for products of large numbers very quickly and get architectures working at frequencies above 220 MHz, for the calculation of products in the fields $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, taking 15 clock cycles and 4 respectively, making them competitive with more recently published implementations.

Agradecimientos

Mediante estas líneas quiero agradecer a aquellas personas que sin su apoyo, comprensión y confianza, esta tesis no hubiera sido posible, y yo no hubiera aprendido todas las lecciones que en su desarrollo la vida me ha dado.

Primero quiero agradecer a mi familia comenzando con mi madre Genoveva Corona López, que al verme apurado y contra reloj siempre me dio su apoyo, ya sea con su cariño incondicional o con una palabra de aliento, quiero agradecer a mi padre, que siempre me proporciono las enseñanzas que su experiencia le han dejado y siempre me mostró que hay que preocuparse menos de lo que uno cree.

Agradezco a mis hermanos Xochitl y Nezahualcoyotl por ser siempre una motivación, pues su confianza y admiración me hicieron siempre querer ser una persona mejor y perseverante.

Un gigantesco agradecimiento a Mariana Velazquez del Angel, que ya sea cerca o a la distancia, siempre confió en mi, me dio su apoyo, su cariño y mucha comprención, me ayudo a levantarme que caí, y a reafirmar mi confianza en los momentos que dude.

Agradezco a mis amigos Alberto, Alejandro, Nayelly, Laura, Cynthia, Anallely, Herón, Rodrigo, Luis, Edgar, Cuauhtémos Mancillas y a todos aquellos con los que compartí la aventura de la maestría que sin ellos no huboiera sido la misma gran experiencia que recordare toda mi existencia.

También un gran agradecimiento a Sofía Reza, por ser un gran apoyo en el departamento de computación, reduciendo siempre al mínimo la burocracia y auxiliándome siempre en todas mis dudas con respecto a los procedimientos y enseñandome que se puede hacer un gran trabajo manteniendo siempre la alegría.

A mis asesores, el doctor Francisco Rodríguez Henríquez, que siempre puso su atención por que este trabajo llegará a buen término, y siempre estuvo convencido de que se podian obtener buenos resultados, y al doctor Debrup

Chakraborty, que gracias a él, fue posible iniciar este trabajo en primer lugar, les doy las gracias.

Quiero agradecer especialmente al CINVESTAV por ser mi casa de estudios, y por permitirme acercarme al mundo de la investigación científica, mostrandome lo blanco, lo negro y lo gris de esta experiencia, y por brindarme su apoyo en distintas faenas que surgieron a lo largo del desarrollo de este documento. Además agradezco al Consejo Nacional de Ciencia y Tecnología, que me proporciono el financiamiento necesario durante mis estudios. Así como lo hicieron el proyecto UC-Mexus y la doctora Nareli Cruz Cortés con el proyecto 132073, pues sin su apoyo, este proyecto no hubiese sido posible.

Agradezco también al Doctor Jean-Luc Beuchat y a la universidad de Tsukuba, Japón por permitirme trabajar con ellos, además de brindarme muchas experiencias que siempre recordaré en lamentablemente poco tiempo.

Y finalmente agradezco a mi país México y a todas esas personas que conocí en esté tiempo, cuyo nombres son demasiados para incluirlos, pero no por eso son menos importantes.

Capítulo 1

Introducción

1.1. Objetivos

Este trabajo tiene como objetivo el desarrollo de un multiplicador modular eficiente para números con un tamaño de 256 bits, con el fin de emplear dicho multiplicador en el cálculo de emparejamientos bilineales con 128 bits de seguridad, lo anterior sobre un dispositivo *FPGA*[1] utilizando el lenguajes de descripción de hardware VHDL, para lograrlo se consideraron los siguientes objetivos específicos:

1. Realizar una exploración de las distintas tecnologías de la familia de *FPGA*'s **Virtex** de **Xilinx** para poder aprovecharlas en el desarrollo de los diseños.

2. Utilizar intensivamente los componentes dedicados disponibles para construir multiplicadores eficientes.

3. Buscar que el diseño final sea competitivo ya sea en espacio o en tiempo con los diseños ya publicados en la literatura.

4. Explorar la eficiencia del esquema de multiplicación de Montgomery en distintas variantes para averiguar que rendimientos puede proporcionar en este tipo de diseños.

5. Hacer uso del algoritmo de multiplicación de Karatsuba para polinomios y verificar su efectividad.

6. Realizar el desarrollo de una arquitectura para multiplicar elementos en $\mathbb{F}_p$ y otra para elementos en $\mathbb{F}_{p^2}$.

[1]Del inglés *Field Programmable Gate Array*

7. Hacer uso de la estrategia de diseño *pipe-line* para poder obtener un mejor desempeño.

1.2. Antecedentes

Desde que comenzó la comunicación humana han existido secretos, y con ellos surge la necesidad de ocultarlos de aquellos intrusos que busquen tener acceso a ellos sin permiso. Es así como surge la criptografía, que engloba el conjunto de técnicas mediante las cuales se altera la estructura de un mensaje, haciéndolo intelegible a quienes no estén autorizados para conocer el secreto, pero permitiendo la existencia de algún método legítimo que permita obtener el significado original y así pueda ser leido por el destinatario.

Dentro del lenguaje de criptografía, al proceso de alterar un mensaje se le nombra como cifrado, al resultado se le llama cifra y al procedimiento para convertir una cifra en el mensaje original, se le conoce como descifrado. Para poder cifrar y descifrar el mensaje original de forma eficiente es necesario poseer un dato particular al que se le llama llave o clave secreta.

Así como la comunicación dio lugar a la criptografía, la criptografía dio lugar al criptoanálisis, definido como el conjunto de técnicas mediante las cuales los adversarios (aquellos intrusos que quieren conocer el secreto 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 llave.
- Obtener el valor de la llave secreta.
- Poder inyectar mensajes apócrifos en la comunicación, sin que estos sean detectados.

El objetivo más común que se plantean los adversarios es el de poder leer el mensaje cifrado sin la necesidad de tener la llave, por lo que generalmente, cuando se habla de "romper" el criptosistema se refiere a lograr este objetivo de una manera eficiente.

Un sistema de cifrado, que es uno de los varios esquemas criptográficos existentes, se compone de los siguientes elementos fundamentales:

1. El espacio de mensajes: El conjunto de todos los posibles mensajes que pueden ser cifrados usando ese esquema.

2. El espacio de cifras: El conjunto de todas las posibles cifras que podemos obtener usando ese esquema criptográfico, hay que notar que éste y el espacio de mensajes tienen el mismo tamaño.

3. Algoritmo de cifrado: Es la función con la que se vincula un mensaje con una cifra.

4. Algoritmo de descifrado: Función inversa al cifrado, mediante la cual se obtiene el mensaje original a partir de la cifra.

La criptografía moderna se divide en dos ramas principales, la primera es la criptografía simétrica. Este tipo de criptografía toma su nombre de la existencia una sola llave secreta, que usa tanto el que envía para cifrar como el que recibe para descifrar.

En 2001, el NIST (National Institute of Standards and Tecnology), anuncia AES (Advanced Encryption Standard), como ganador del concurso propuesto para definir el estándar que es recomendable usar para cifradores simétricos por bloques. **AES** maneja llaves de 128, 192 y 256 bits y siempre mensajes de 128 bits.

Cuando un cifrador por bloques resiste una serie de pruebas y es considerado seguro (como **AES**), se presupone que el ataque más eficiente para romperlo es el llamado ataque por "fuerza bruta", el cual consiste en probar todas las posibles llaves para descifrar el mensaje.

Por esta razón parte de la fortaleza de los sistemas criptográficos radica en el enorme tamaño del espacio de llaves, con el objetivo de volver impráctico un ataque de fuerza bruta. Sin embargo, conforme la tecnología avanza, y las computadoras se hacen más rápidas y baratas, la seguridad que nos ofrece cierto espacio de llaves va decayendo.

Con base en lo anterior se construye la definición de "bits de seguridad", pues se dice que para un sistema que tiene n bits de seguridad, se necesitan 2^n operaciones para romperlo, por ejemplo, **AES128** tiene 128 bits de seguridad por que se requieren 2^{128} operaciones de descifrado para romperlo.

En un inicio, para hacer entrega de la llave secreta las entidades que buscaban comunicarse necesitaban tener contacto directo, pues enviarla en claro por una gran distancia supondría un grave riesgo de seguridad en el sistema. A esta situación se le conoce como el *Problema del Secreto Compartido* y se considera que es un paso crítico en el uso de la criptografía simétrica

Una forma de resolver esta dificultad se logró en 1976, cuando Witfield Diffie y Martin Hellman, propusieron un novedoso protocolo de secreto compartido, que tiene sus bases en la teoría elemental de números y los campos finitos primos[16].

Un campo finito primo está compuesto por los números enteros $\mathbb{Z}$ partido por un número primo p, sobre los cuales se aplican las operaciones de suma, resta, multiplicación e inversión modular. Es representado como $\mathbb{F}_p$.

El protocolo de Diffie-Hellman entre dos entidades se compone de los siguientes pasos:

1. Las entidades que se quieren comunicar acuerdan como parámetros públicos un primo p y un elemento $g \in \mathbb{F}_p$

2. Posteriormente cada entidad genera un número secreto $a \in \mathbb{F}_p$ y $b \in \mathbb{F}_p$ respectivamente, y aplican la operación $S_A = g^a \bmod p$ y $S_B = g^b \bmod p$, para enviar este valor a la entidad con la que busca comunicarse.

3. Finalmente cada entidad recibe el valor S de la otra entidad, y aplican la operación $S_c = S_B^a \bmod p$ la entidad que generó el valor secreto a, y equivalentemente la entidad que generó el valor b aplica $S_c = S_A^b \bmod p$ por lo que ahora ambas entidades tienen el mismo valor S_c.

Si un adversario intercepta el valor $S_A = g^a \bmod p$, su principal objetivo es obtener los valores secretos a y b para tener acceso a la conversación segura entre las dos entidades. Cuando se tiene $S = g^a$ en los números reales, el valor de a se obtiene como $a = \log_g S$, pero en el caso de la operación $S = g^a \bmod p$, no está definida la operación logaritmo, y de hecho se considera que este problema es especialmente difícil de resolver en campos finitos. En la literatura este problema es conocido como el *Problema de Logaritmo Discreto en Campos Finitos*.

De forma curiosa, el protocolo de Diffie-Hellman, el cual fue concebido para solucionar un problema propio de la criptografía simétrica, sirvió para dar paso a una rama de la criptografía totalmente nueva: La criptografía asimétrica o de llave pública.

En este tipo de criptografía, cada entidad tiene un par de llaves, una pública y otra privada. La llave pública como su nombre lo dice, la puede conocer cualquier entidad y sirve para cifrar aquellos mensajes dirigidos al poseedor de dicha llave. Por otro lado, la llave privada, de la cual sólo debe tener conocimiento su dueño legítimo, sirve para descifrar todo aquello que se cifró con la llave pública. De este modo, cualquiera puede cifrar mensajes dirigidos al dueño del par de llaves, pero sólo quien posea la llave privada podrá descifrarlos. Una analogía del mundo real sería un candado al que cualquiera puede cerrar, pero únicamente quien posea una llave especial puede abrir.

A lo largo de los años se han propuesto muchos protocolos de criptografía asimétrica, como ElGamal[18] y DSA[30], cuya seguridad descanza en el *Problema del Logaritmo Discreto*. Lo más común en estos esquemas es que usen la operación $y = g^x \; mod \; p$ donde el primo p y $g \in \mathbb{F}_p$ son parámetros públicos, $y \in \mathbb{F}_p$ es la llave pública, y $x \in \mathbb{F}_p$ la llave privada, protegida por el problema del logaritmo discreto.

Los campos finitos utilizados en estos protocolos de llave pública pueden ser campos finitos primos.

La criptografía de llave pública tiende a ser más lenta que la criptografía simétrica, pues por ejemplo, aquellos protocolos que se basan en el *Problema del Logaritmo discreto*, como ElGamal[18] requieren del cálculo de varias exponenciaciones con números desmesuradamente grandes. Además existen algoritmos más eficientes que los ataques por fuerza bruta para poder descubrir la llave privada a partir de la llave pública, por lo que un tamaño de llave nos ofrece menos bits de seguridad efectivos si se compara con la criptografía simétrica.

En 1985 surge la criptografía de curvas elípticas, las cuales para su uso en criptografía están definidas sobre un campo finito y es posible realizar una operación conocida como suma de punto, de esta manera en vez de trabajar sólo números, las operaciones se efectúan con puntos. Lo anterior permite que este tipo de criptografía sea más segura debido a que el Problema del Logaritmo Discreto para Curvas Elítpicas, problema en el que se sustenta su seguridad, es mucho más difícil que su análogo, el Problema del Logaritmo Discreto en Campos Finitos.

1.3. Motivación

La eterna contienda entre la criptografía y el criptoanálisis ha provocado que cada día se busquen métodos más sofisticados de cifrado y descifrado, para cumplir con las necesidades de velocidad y seguridad que se requieren hoy en día.

En medio del desarrollo tecnológico provocado por esta competencia surgen la criptografía de curvas elípticas, y el uso de emparejamientos bilineales, para la implementación de nuevos esquemas y protocolos de cifrado y de firma digital, como lo son: el protocolo de Diffie-Hellman tripartito[25] , la criptografía basada en la identidad[35], y los esquemas de firma corta[10] y muchos más [17].

Sin embargo, el cómputo de emparejamientos bilineales se ha caracterizado por ser una operación lenta y costosa, que consume una gran cantidad de operaciones para realizarse, especialmente cuando se trabaja con curvas definidas sobre campos finitos primos.

Conforme han pasado los años se han hecho una serie de contribuciones incrementales que en su conjunto, han reducido el tiempo de ejecución del emparejamiento significativamente, tanto en software, sobre todo con mejoras algorítmicas y su programación, como en hardware, con propuestas de nuevas arquitecturas.

Las curvas de Barreto-Naehrig, o curvas BN [4], tienen la característica de que son fáciles de construir, gracias al método proporcionado por sus autores. En el caso en el que estas curvas estén definidas sobre campos de 256 bits, tienen la característica de que nos ofrecen aproximadamente 128 bits de seguridad tanto cuando se trabaja sólo con

curvas elípticas como cuando se trabaja también con el resultado del emparejamiento. Sin embargo no hay que perder de vista que éstas curvas se pueden definir para cualquier nivel de seguridad.

Las implementaciones en hardware han cobrado una mayor importancia recientemente, pues han surgido nuevos dispositivos que necesitan comunicaciones seguras, como lo son los teléfonos celulares, las agendas electrónicas y las consolas portátiles, los cuales no tienen los recursos para realizar operaciones complejas en un tiempo razonable para el usuario. La situación anterior se resuelve con diseños dedicados de hardware, las cuales son más baratas y liberan otros recursos para otro tipo de operaciones.

El producto modular es una operación crítica en la ejecución de los criptosistemas de llave pública que trabajan sobre campos finitos, curvas elípticas y de emparejamientos bilineales, por ejemplo para calcular estos últimos se requieren miles de productos modulares con números grandes.

De ahí la importancia de realizar multiplicadores modulares eficientes, y para hacerlos aún más rápidos una de las mejores alternativas es diseñar una arquitectura exclusiva para este fin, por medio del lenguaje de descripción de hardware VHDL, para aprovechar las ventajas de velocidad y flexibilidad del diseño digital y así acelerar la ejecución de las aplicaciones.

1.4. Estado del arte

Los trabajos realizados sobre hardware para emparejamientos bilineales han avanzado mucho, sobre todo en aquellos que están definidos sobre curvas de característica 2 y 3, como puede verse en [6] y en su versión extendida [7], donde con la ayuda de un multiplicador Karatsuba, se logran realizar emparejamientos en el campo 2^{691} en menos de 19 μs, con un nivel de seguridad de 105 bits. Y para curvas definidas sobre el campo 3^{313}, se logra una seguridad de 109 bits con un tiempo de cálculo de casi 17 μs.

La ventaja de trabajar con curvas definidas en campos de característica 2 y 3, es que la aritmética es más eficiente comparada con la de los campos $\mathbb{F}_p$ con $p > 3$, lo que permite buscar mejoras en el algoritmo del emparejamiento como se puede ver en [5].

Sin embargo debido a la falta de curvas de característica 2 y 3 que ofrezcan una seguridad aceptable en el resultado del emparejamiento, hay que aumentar mucho la cantidad de bits que se manejen para conseguir una seguridad aceptable (al menos 128 bits, equivalentes a la seguridad ofrecida por el cifrador por bloques **AES128**).

Cuadro 1.1: Comparación de niveles de seguridad entre algoritmos de criptografía de llave pública

Bits de seguridad	Bits de ECC	Bits de RSA/DSA
80	163	1024
128	283	3072
192	409	7680
256	571	15360

Una solución a este problema es trabajar con curvas elípticas definidas sobre campos $\mathbb{F}_p$, con $p > 3$, pues estas curvas nos ofrecen niveles de seguridad mayores, pero el precio que se paga es tener que trabajar con una aritmética de campos finitos primos que es menos eficiente, haciendo más difícil diseñar y trabajar con este tipo de curvas.

Por otro lado también se han realizado varios avances de la implementación en software del emparejamiento bilineal, aprovechando las características del cálculo del mismo, buscando paralelizar operaciones, y usar las nuevas tecnologías de los procesadores, como se puede ver en [22] o en [9].

Un tipo de curvas definidas en $\mathbb{F}_p$, con $p > 3$, son las curvas de Barreto-Naehrig, descritas en [4], las cuales son muy sencillas de diseñar poseen una seguridad alta, por ejemplo, aproximadamente de 128 bits cuando son definidas sobre un campo con $\log_2 p = 256$, además como se puede ver en [8], este tipo de curvas tienen muchas características que pueden ser aprovechadas para hacer más eficiente el emperajamiento bilineal.

Por otro lado los desarrollos en hardware se han enfocado sobre todo en realizar multiplicadores modulares eficientes como se puede ver en [26], que fue la primera implementación del emparejamiento bilineal usando curvas de Barreto-Naehrig, y está basado en un multiplicador de Montgomery[32] tradicional. Sin embargo se puede notar que aún es muy lento comparándolo con los trabajos de software de su época.

Posteriormente apareció el trabajo descrito en [19], en donde se propone un método de multiplicación basado en el multiplicador de Montgomery para polinomios, aprovechando las características de las curvas Barreto-Naehrig para hacerlo más eficiente y compacto. Este diseño se caracteriza por tener una reducción modular muy rápida, basándose en la adecuada elección de los parámetros que definen la curva, este multiplicador se mantiene aún un poco lento, debido a algunos defectos de diseño que fueron corregidos en [20].

Las nuevas tecnologías y componentes incorporados en los modelos más modernos de *FPGA* [11] pueden ser aprovechados para realizar aplicaciones cirptográficas, como se muestra en [24], donde se describe como se puede aprovechar el potencial de los componentes *DSP48Slice* y de las memorias *DualPortRAM* para desarrollar diseños eficientes.

La mayoría de los diseños de multiplicadores orientados a emparejamientos bilineales previos usan variantes del algoritmo de producto de Montgomery, sin embargo en 2011 apareció en [46] un novedoso diseño que presenta un multiplicador basado en el teorema del residuo chino (RNS), con los que se obtienen mejoras substanciales con respecto a los trabajos anteriores.

1.5. Metodología

EL primer objetivo que se planteó resolver en el desarrollo de este trabajo fue el de realizar una arquitectura eficiente para el producto modular en $\mathbb{F}_p$, que estaba orientado para ser usado en emparejamietos bilineales sobre curvas Barreto-Naehrig [4]. Para ello se eligió analizar el diseño más rápido en la literatura de ese entonces, el cual era la arquitectura presentada en [19]. Se realizó un análisis sobre dicho trabajo para entender y replicar la parte algoritmica del diseño. Valiéndose de los componentes de los *FPGA* **Virtex**, la estrategia de diseño de *pipe-line*, el método de multiplicación visto en [38] y el algoritmo de Karatsuba se consigue una arquitectura que consigue ser competitiva.

Por otro lado, durante el desarrollo de la tesis aparecieron nuevas arquitecturas que forzaron a mejorar el diseño propuesto, los cuales estaban diseñados para la familia **Virtex 6**, por lo que de igual manera se dio el salto a esta nueva tecnología y se propuso un nuevo diseño, esta vez para realizar la multiplicación entre elementos en $\mathbb{F}_{p^2}$, realizando además un escalamiento del multiplicador propuesto en base al metodo presentado en [38], con lo que ahora se consigue un diseño más costoso, pero que resulta ser muy rápido y relativamente fácil de escalar.

1.6. Estructura de la tesis

Este trabajo está organizado en 7 capítulos, el primero corresponde a la introducción. Después, en el capítulo 2 se hace una reseña de las principales plataformas de hardware utilizadas hoy en día, describiendo las tecnologías diseño de digital reconfigurable, y justificando la elección de aquellas que fueron usadas para el desarrollo de este proyecto, acompañando lo anterior con varios conceptos básicos relacionados con el diseño digital.

En el capítulo 3, se describe el trasfondo matemático necesario para comprender los algoritmos de criptografía, así como las caracterísiticas de las principales estructuras matemáticas con las que se trabaja.

Posteriormente en el capítulo 4, se hace mención de los algoritmos más importantes de aritmética computacional, como los son la suma y el producto modular con números

grandes, necesarios para comprender los diseños en los que se enfoca la tesis.

Es en el capítulo 5 donde se describe el primer diseño desarrollado en esta tesis que es un multiplicador en el campo $\mathbb{F}_p$, utilizando una variante del algoritmo descrito en [19], pero con un nuevo diseño para hacerlo más eficiente.

Para calcular el emparejamiento bilineal, con curvas de Barreto-Naehrig, basándose en [8], se necesita realizar una gran cantidad de multiplicaciones en el campo $\mathbb{F}_{p^2}$, por esta razón el capítulo 6 se presenta un diseño para poder realizar de forma eficiente los productos en ese campo, utilizando los componentes empotrados de los dispositivos *FPGA*.

Por último se encuentra el capítulo 7 donde se resumen los resultados de estas implementaciones, se realizan comparaciones con otros desarrollos similares a la fecha, y se describen las conclusiones obtenidas a través la construcción de este trabajo.

1.7. Resultados Principales

Las principales aportaciones de este trabajo es el desarrollo de 2 multiplicadores de 64 y 128 bits, utilizando como base los *DSP48Slices* que incluyen los dispositivos *FPGA* de la familia **Virtex 6** de **Xilinx**. El primero fue construido con *DSP48Slices*, explotando sus registros y sumadores internos, mientras que el segundo hace uso de la técnica *Carry-Save* para realizar las sumas de manera eficiente. De esta manera se consiguen 2 arquitecturas de multiplicadores, la primera trabajando a 220 MHz, entregando un nuevo producto en $\mathbb{F}_p$ cada 15 ciclos de reloj con una latencia de 40 ciclos de reloj, y la segunda trabajando a una frecuencia de 235 Mhz, entregando un nuevo producto en $\mathbb{F}_{p^2}$ cada 4 ciclos de reloj, con una latencia de 45 ciclos.

Capítulo 2

Tecnologías de Hardware

En este capítulo se abordan las definiciones relacionadas con la tecnología que fue utilizada en el desarrollo del proyecto. Particularmente, se comienza dando una definición de los *FPGA*, así como de su estructura y funcionalidad; posteriormente, se describen sus principales componentes para terminar con una breve explicación de las herramientas utilizadas en el diseño, desarrollo e implementación de los multiplicadores presentados en los capítulos 5 y 6.

2.1. FPGA

Cuando comenzó el diseño digital y los circuitos integrados, el tiempo requerido en la implementación y pruebas era considerablemente mayor que el tiempo invertido en el diseño del circuito. Además, si dicho diseño no funcionaba, implicaba una pérdida de tiempo y recursos.

Con base en este problema, en la década de los 70's, **Texas Instruments** comenzó a realizar los primeros avances en sistemas reconfigurables, utilizando memorias para guardar el estado de los circuitos, los cuales cambiaban su salida dependiendo de la información en la memoria. Por el estado, se entiende las conexiones de los componentes, con lo que se podían realizar cambios de configuración de los mismos. Posteriormente, surgieron los Dispositivos Lógicos Programables o *PLD*'s por sus siglas en Ingles (*Programmable Logic Devices*), los cuales son arreglos o matrices de compuertas lógicas y otros componentes programables, interconectados por fusibles.

La tecnología y la forma de interconexión han cambiado con el transcurso de los años.

Figura 2.1: Encapsulado de un FPGA Xilinx [42]

Hoy en día los componentes son cada vez más complejos, numerosos, veloces y pequeños; sin embargo, se sigue manteniendo la idea de la programación de conexiones. Así es como han surgido una gran cantidad de familias de *PLD*'s, algunas más complejas que otras, de las cuales, los arreglos de compuertas programables en campo, abreviados como *FPGA*'s (por sus siglas en inglés *Field Programmable Gate Array*) , son los más utilizados por su flexibilidad, capacidad y complejidad.

Los *FPGA*'s son una familia de *PLD*'s que se caracterizan por presentar la estructura de una matriz de componentes lógicos con interconexiones programables. Cabe señalar que dichos componentes lógicos no son sólo compuertas, sino que incluyen sumadores, registros, memorias y multiplicadores. Un ejemplo de encapsulados de un chip *FPGA* se muestra en la figura 2.1.

En tiempos recientes los *FPGA*'s han adquirido gran popularidad debido a su versatilidad y tamaño, permitiendo realizar diseños muy complejos en el mismo encapsulado. Así mismo, debido a que los componentes son cada vez más especializados, las implementaciones en estos dispositivos, se han vuelto más rápidas y eficientes.

Por otra parte, los *FPGA*'s son una evolución directa de los *CPLD*'s (*Complex Programmable Logic Devices*), cuya principal diferencia radica en que estos últimos se constituyen de arreglos de compuertas, mientras que los primeros tienen una arquitectura de matriz, que los hace más flexibles, al mismo tiempo que permiten una cantidad considerablemente mayor de componentes dentro del encapsulado. Un par de factores a considerar antes de utilizar un *FPGA* son su velocidad y su costo, ambos relacionados con los componentes que lo conforman:

- **Velocidad**: Dentro del dispositivo, la señal tiene que recorrer una gran cantidad de conexiones reprogramables, las cuales son más lentas que las conexiones que no lo son. Debido a esto, el tiempo de propagación es mayor en un FPGA que en los dispositivos no programables.

- **Precio**: Dado que los componentes son más complejos y grandes que una compuerta, el precio del dispositivo aumenta.

En general, el *FPGA* no es un dispositivo final 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* (*Very Large Scale Integration*), que son circuitos integrados de función específica, los cuales son más baratos y rápidos que un *FPGA*.

La arquitectura interna de un *FPGA*, más allá de la matriz de componentes, varía mucho dependiendo del fabricante y de los distintos modelos. Las principales compañías productoras de *FPGA*'s son [11]:

- Xilinx: Es la empresa más importante de desarrollo de *FPGA*'s en el mundo, fundada por los creadores del concepto de *FPGA*.

- Altera: Es la competidora más grande de **Xilinx**.

- Actel: Empresa dedicada a los semiconductores. Desarrollador de *FPGA*'s especializados en telecomunicaciones.

Debido a que los componentes que constituyen las arquitecturas de distintos fabricantes son tan diferentes entre sí, en ocasiones es muy difícil realizar comparaciones objetivas entre implementaciones, sin que existan ambiguedades con respecto al buen o mal uso de los recursos.

Sin embargo, dado que hoy en día Xilinx es la empresa más importante en la fabricación de FPGA's, la mayoría de los diseños van orientados a dispositivos de este fabricante.

2.1.1. Familias

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*'s adecuada a las necesidades del proyecto. Las principales familias en la actualidad son [40]:

- **Spartan 6**: Dispositivos pequeños y sencillos, destinados a aplicaciones simples y de enseñanza, con bajo consumo de poder y un precio bajo.
- **Artix 7**: Opción industrial barata, con un bajo consumo de potencia y con capacidades de procesamiento de señales.
- **Kintex7**: Modelos baratos para aplicaciones de alto rendimiento, con mayores recursos que otras familias.
- **Virtex 5 y 6**: La gama más poderosa de **Xilinx**, con la mayor cantidad de componentes especializados, justo para aplicaciones que necesiten el mayor espacio y rendimiento, sin embargo, es la familia más costosa en términos de consumo de energía y precio.
- **Virtex 7**: La familia de más reciente generación, con muchos más componentes que sus predecesoras, mayor velocidad y mayor costo. Útil para aplicaciones que requieran una gran capacidad de espacio y rendimiento de tiempo, donde la velocidad es prioritaria sobre el tamaño y el costo.

En este trabajo se hace énfasis en las familias **Virtex 5 y 6**, a las que pertenecen los dispositivos sobre los cuales se desarrollaron los diseños presentados en los capítulos 5 y 6, respectivamente. Cabe mencionar que estas familias fueron elegidas por sus características de alto rendimiento y por su disponibilidad [1].

En la siguiente sección se detalla la arquitectura y las características principales de la familia **Virtex 6**, la cual es una actualización de la familia **Virtex 5**.

2.1.2. Slices

La arquitectura de un *FPGA* puede verse como una jerarquía de componentes contenidos unos dentro de otros. En esta jerarquía, se encuentran primero los bloques lógicos configurables o *CLB*'s (por sus siglas en ingles *Configurable Logic Blocks*)[44], los cuales están conectados directamente a la matriz de interconexión, que es donde están organizadas las conexiones reconfigurables. Dentro de los *CLB*'s están agrupados dos componentes

[1] Particularmente la familia **Virtex 7** es tan nueva y costosa que aún no contamos con las herramientas adecuadas para el desarrollo de diseños en esta plataforma

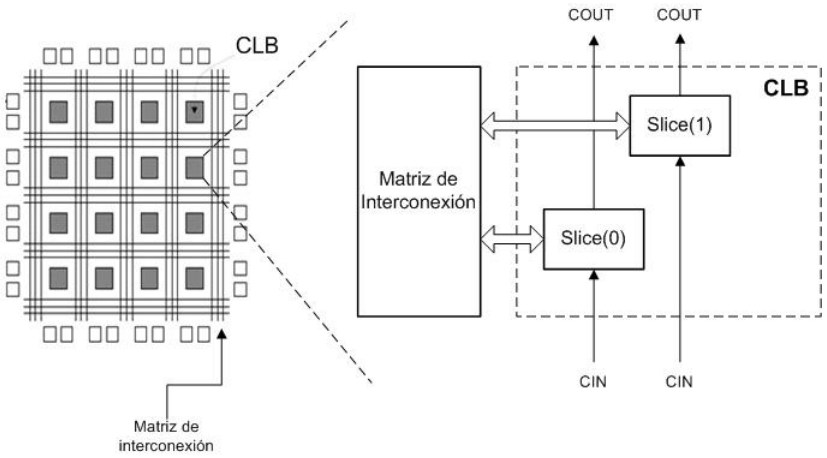

Figura 2.2: Constitución de los *CLB*'s en el *FPGA* [44]

.

básicos llamados *Slices* (Figura 2.2).Cada *Slice* está conformado por multiplexores, registros y cuatro elementos reconfigurables llamados LUT's (*Look Up Tables*), que se pueden observar en la figura 2.3.

Los LUT's son componentes programables capaces de realizar cualquier operación lógica de seis entradas y una salida. Cabe mencionar que el número de slices requeridos en la implementación es considerado una métrica para definir el tamaño del diseño

Además de los *Slices*, se puede disponer de otros elementos dentro de un *FPGA*, como lo son registros extra y las llamadas *DualPortRAM*'s, presentes en las familias *Virtex 5* y *Virtex 6*, las cuales son memorias con doble puerto de tamaño configurable, esto permite hacer dos escrituras, dos lecturas, o una lectura junto con una escritura, en el mismo ciclo de reloj, estas características las hacen muy atractivas para su uso en implementaciones eficientes.

2.1.3. DSPSlices

Además de los elementos anteriores, también existen los llamados *DSPSlices*, 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 acumladores, sumadores, multiplicadores y hasta como compuertas lógicas. Los *DSPSlices* usados

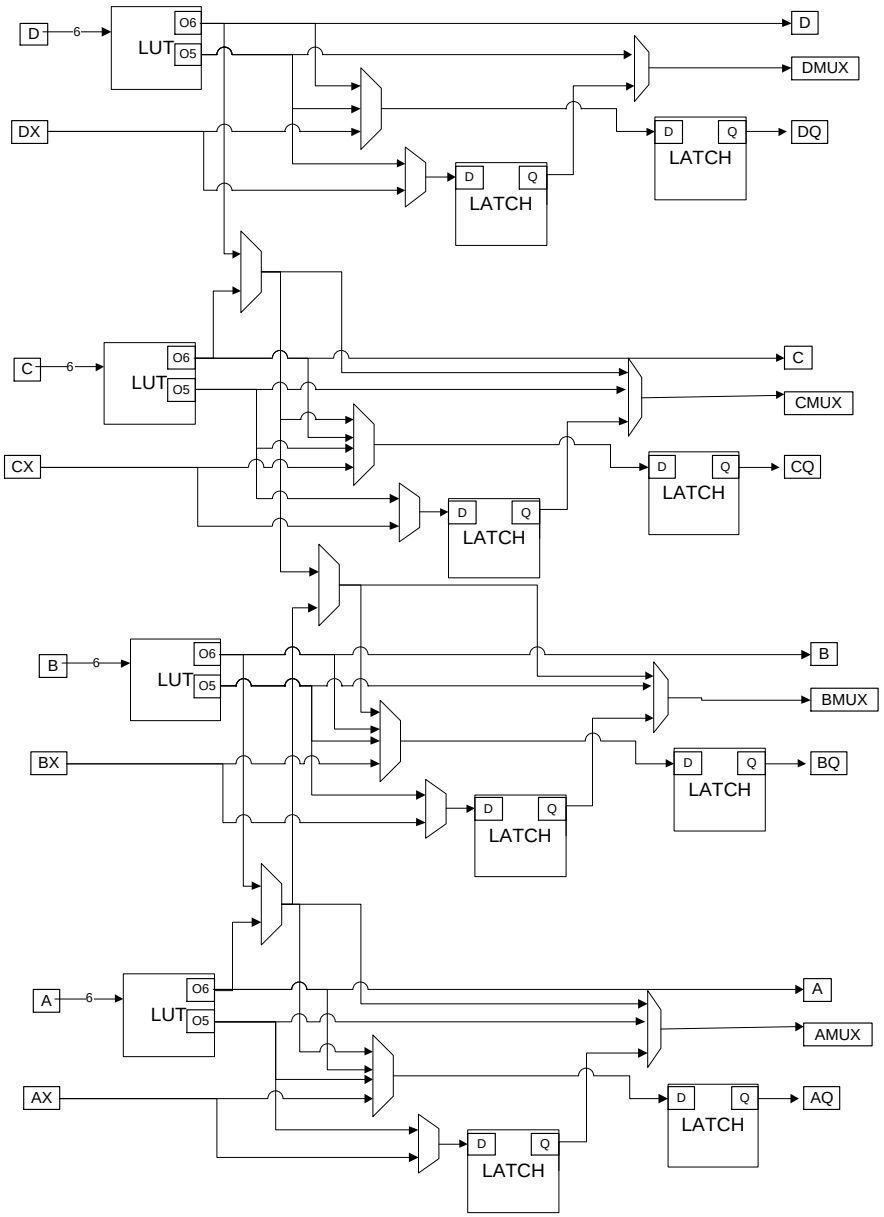

Figura 2.3: Ejemplo de la dispocisión de los LUT's en un *Slice* que puede ser usado como elemento de lógica o de memoria[44]

.

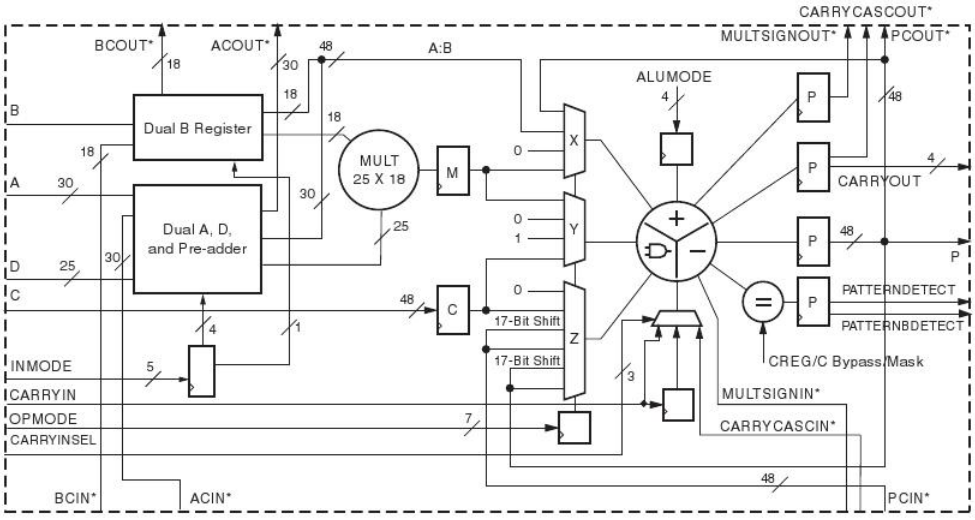

Figura 2.4: Arquitectura Interna de un *DSP48SlicesE1* [45]

.

por la familia **Virtex 6** son los llamados *DSP48SlicesE1* [45], y su arquitectura se puede ver en la figura 2.4.

Como se puede observar en la figura 2.4, estos dispositivos poseen 4 entradas principales (A, B, C, D) y estan conformados por:

- Un pre-sumador de 25 bits entre las entradas A y D.
- Un multiplicador de 25x18 bits entre la entrada B y la salida del presumador.
- Una ALU de 48 bits que puede tomar como entradas: A, B, C, la salida del presumador, una entrada de retroalimentación o la salida del *DSPSlice* vecino conectado en cascada.

Las conexiones internas del *DSP48Slices* se pueden configurar para que realice una amplia gama de funciones, como pueden ser:

- La operación $(A+D)\cdot B \star C$, donde el operador $\star$ puede tomar valores de $+, -, \oplus, \vee, \wedge$, además de que se puede anular alguno de los operandos.
- También es posible usar un *DSP48Slice* en forma de operaciones individuales como puede ser $A * B$ o $(A||B) + C$.

- Es posible configurar los multiplexores internos para que la salida se retroalimente a la entrada C, de forma que se pueda utilizar como acumulador.
- Se puede realizar un desplazamiento de 17 bits sobre la entrada C, ya sea a la izquierda o a la derecha.
- Todas las configuraciones pueden ir precedidas por un máximo de 6 niveles de *pipeline*, gracias a los registros internos del *DSP48Slice*.

Ambas familias, **Virtex 5** y **Virtex 6** poseen componentes similares, la diferencia sustancial es el número, pues mientras que el modelo más pequeño de **Virtex 5** posee un máximo de 42 *DSP48Slices*, el modelo más pequeño de **Virtex 6** posee un máximo de 288, y un modelo que le sigue en tamaño tiene 488, de manera que esta familia es mucho más atractiva cuando se quieren hacer uso de una gran cantidad de *DSP48Slices*.

2.2. VHDL

Hasta el momento se ha hablado únicamente acerca de la configuración de las conexiones en los *PLD*'s, pero no se ha abordado aún el "como configurarlas", y la respuesta es simple: por medio de un lenguaje. Es aquí donde intervienen los llamados lenguajes de descripción de hardware o *HDL* (*Hardware Description Language*), los cuales varían en complejidad dependiendo de los dispositivos y diseños con los que se va a trabajar. Algunos ejemplos de estos lenguajes son:

- Verilog
- Superverilog
- VHDL
- Abel

Básicamente estos lenguajes de descripción de hardware tienen el mismo objetivo: "definir una arquitectura para un circuito digital".

Los *HDL*'s pueden estar orientados a los diseños jerárquicos, donde un conjunto de elementos simples se interconectan entre sí para formar un elemento más complejo, que a su vez es conectado con otros y así sucesivamente hasta formar el diseño final. Esta forma de programación modular es un aspecto que se comparte con los lenguajes de programación

de software, por lo que es conveniente usar este tipo de división para hacer al diseño más comprensible y manejable. Sin embargo es importante notar que existen otras formas de diseño, como la funcional, en el cual se describe únicamente el comportamiento del circuito. Una u otra se elige dependiendo del problema a resolver y las herramientas de desarrollo.

El lenguaje de descripción de hardware más utilizado actualmente es *VHDL*, cuyas siglas provienen de *VHSIC* (*Very Hight Speed Integrated Circuit*) y *HDL*. Basándose en el lenguaje de programación **ADA**, la marina de los E. U. creó el lenguaje *VHDL* con el fin de documentar sus diseños digitales, por lo que está construido para que pueda describir cualquier clase de circuito lógico [11]. 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 [14].

Para el lenguaje *VHDL*, cada compañía proporciona un entorno de desarrollo para sus dispositivos, aportando en ocasiones sus propias bibliotecas de componentes y funciones para hacer más sencilla la implementación.

2.3. Metodología de Diseño

La perspectiva de desarrollo en diseño digital es 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.

Por ejemplo, al realizar una implementación en software, la arquitectura sobre la que se va a trabajar tiene una velocidad y tamaño de palabra definida, mientras que por otro lado en el diseño digital estas restricciones no son tan claras, debido a la libertad de definir los mas mínimos detalles de la arquitectura.

Siempre que existen tantas libertades, hay que realizar un mejor análisis acerca de cuales son las mejores opciones. Además siempre queda la posibilidad de que algunas arquitecturas sean más eficientes en ciertos aspectos o bajo ciertas restricciones que otras, por lo que antes de tomar una decisión hay que hacer un análisis de costo beneficio.

Otra diferencia sustancial es la interpretación del algoritmo que se quiere implementar. Los lenguajes de programación han sido construidos para que dicha interpretación pueda

ser relativamente directa. Lo anterior es posible debido a que el software, por definición, es intangible y se crea a conveniencia, sin embargo el hardware es un conjunto de elementos físicos, que se hay que tratar de adaptar a las interpretaciones matemáticas.

El flujo de diseño usado generalmente es [34]:

- Diseño del circuito y captura en *HDL*
- Simulación Funcional
- Síntesis
- Ubicación y enrutamiento
- Análisis del circuito
- Programación del dispositivo

Una estrategia que demostró su efectividad a lo largo de este trabajo fue aplicar la técnica de "Divide y Vencerás", reduciendo los problemas a tareas sencillas e integrándolos posteriormente. El único inconveniente es que en ocasiones la integración de los componentes lógicos puede ser una tarea complicada, debido a la sincronización.

2.3.1. Simulación Funcional

En los primeros circuitos integrados, la única manera de verificar el correcto funcionamiento de un diseño era construirlo y probarlo físicamente, que es una opción costosa en términos de recursos y de tiempo.

Para dar solución al problema anterior surgieron los simuladores funcionales: programas de software que generan una salida en base al código de descripción de hardware que reciben como entrada. Estos simuladores proporcionan la oportunidad de verificar si la lógica del diseño es correcta; en caso de que no se tengan los resultados esperados, la mayoría cuenta con herramientas de depuración y pruebas, que permiten encontrar errores de una forma sencilla y eficiente.

2.3.2. Síntesis

Se le llama síntesis al proceso mediante el cual se transforma automáticamente la abstracción de algún circuito digital, a su equivalente en compuertas y dispositivos lógicos. Algunos ejemplos de sintetizadores (programa que realiza la síntesis) son aquellos que se encarga de transformar el código HDL a la configuración de interconexiones para algún dispositivo reprogramable, como lo son los sinterizadores **Xilinx** o de **Altera** que convierten el código VHDL o Verilog en flujo de programación para las conexiones inernas de sus *CPLD*'s y *FPGA*'s.

2.3.3. Ubicación y Enrutamiento

Las estimaciones obtenidas después de la síntesis son una aproximación a la lógica del diseño, sin tomar en cuenta el direccionamiento y características físicas de los componentes. Para obtener resultados más precisos existe lo que se conoce como Ubicación y Enrutamiento (*Place and Route*), que es una síntesis mucho más detallada donde se realiza el análisis de recursos y su velocidad, basándose en la disposición de los componentes en el diseño final, la velocidad de sus interconexiones, y características físicas más exactas de los componentes.

Cabe mencionar que en este proceso también se pueden realizar operaciones de optimización, estableciendo restricciones de tiempo, espacio, y consumo de potencia, para que el sintetizador busque la forma óptima de instanciarlo en el dispositivo.

En ocasiones, la metodología más práctica para encontrar las mejores restricciones es la de prueba y error.

También se corre el riesgo de que las restricciones no se puedan cumplir y el diseño quede exactamente igual después del tiempo invertido en tratar de optimizarlo. Por ejemplo, la restricción de tiempo, define cúal es el máximo período con el que el circuito puede trabajar, lo que de manera natural fija cual es la frecuencia máxima de funcionamiento que se puede obtener.

2.3.4. Despliegue Físico

Una vez que termina el proceso de síntesis seguido por el de Ubicación y Enrutamiento, para poder implementar el diseño, hay que pasar a la fase de determinar las conexiones físicas del FPGA. Los dispositivos físicos generalmente van a estar empotrados en una

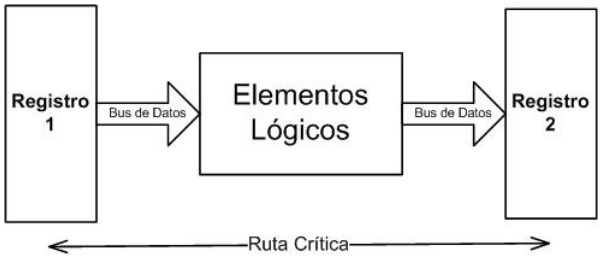

Figura 2.5: Ejemplo de ruta crítica

.

tarjeta de desarrollo, con las conexiones predeterminadas a componentes externos a la FPGA ya incluidos en la tarjeta, como memorias, convertidores analógico-digital, puertos de entrada para ratón y teclado, pantallas de salida, etc. Por lo tanto hay que tener cuidado de a dónde se mapean los puertos, y procurar aprovechar las características de los componentes externos, si es el caso, con que se cuentan en una tarjeta de desarrollo determinada.

Una vez configurado lo anterior, el mismo ambiente de desarrollo arroja un archivo de terminación .bit, el cual puede ser cargado directamente al FPGA, ya sea con la misma aplicación o con otra. La forma de cargar la configuración del diseño al dispositivo varía entre fabricantes y modelos, pero es importante tomar en cuenta que ese archivo de configuración sólo sirve para el dispositivo para el que fue sintetizado.

2.3.5. Métricas de Diseño

Los principales puntos que hay que considerar en la realización de un diseño son:

1. **Frecuencia de Trabajo**: Es la máxima frecuencia a la que puede trabajar el diseño y se define como:

   $$\text{Frecuencia de Trabajo} = 1/\text{Período Máximo}$$

   donde el *Período Máximo* es el tiempo que tarda una señal en recorrer las compuerta y componentes entre dos registros. A este conjunto de compuertas se le conoce como ruta crítica (Figura 2.5).

   Cabe mencionar que entre menor sea el número de elementos en la ruta crítica, menor será el periodo máximo y por lo tanto, mayor será la frecuencia de trabajo.

2. **Ciclos de Reloj**: Es la cantidad de ciclos de reloj que le tomará al diseño concluir su trabajo, este tiempo está dado por el algoritmo y por la forma en que se implementa. Cuando se quiere reducir la ruta crítica la forma más directa de hacerlo es insertar registros en ella, como se ve en la Figura 2.6, no obstante el precio que se paga es un aumento en el consumo de los ciclos de reloj, por lo que se vuelve necesario hacer un análisis de costo/beneficio, para saber cuál es el camino más conveniente.

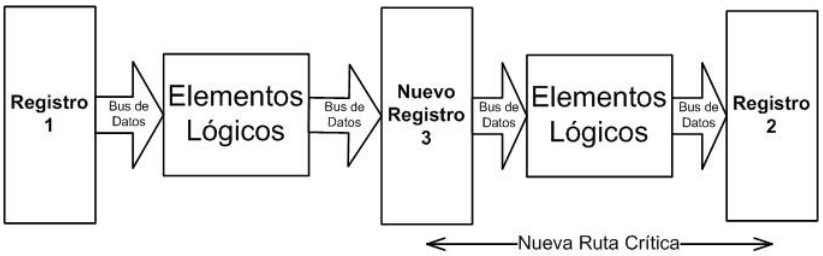

Figura 2.6: Reducción de ruta crítica mediante registros

.

3. **Latencia**: Una de las formas de diseño más usadas es la llamada técnica de "pipe-line"[36], o arquitectura de tubería. Esta técnica consiste en dividir el diseño en módulos secuenciales, que puedan trabajar al mismo tiempo con distintas entradas, tal y como lo ejemplifica la Figura 2.7.

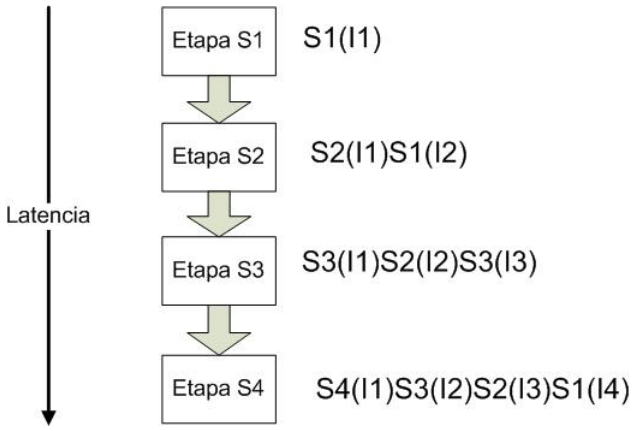

Figura 2.7: Ejemplo de la técnica de tubería o pipe-line

.

Por ejemplo, si se divide una arquitectura en 2 módulos, S1 y S2, la idea es que cuando una entrada I1 ha dejado de procesarse en S1, pasa a S2, pero al mismo tiempo una nueva entrada I2 comienza a procesarse al módulo S1, de esta forma hay dos entradas procesándose en concurrentemente. Con lo anterior se obtiene un nuevo resultado cada vez que se completa un módulo. Al número de ciclos de reloj que le toma a una entrada recorrer todos los módulos se le conoce como latencia.

Agregar más etapas en la tubería, si es posible, puede resultar ventajoso, pues permite aumentar la frecuencia máxima de operación del diseño o disminuir el tiempo que es necesario esperar por un nuevo producto. Sin embargo el precio que se paga es en principio en área, pues cada nueva etapa consume más recursos, y también en latencia, por lo que será aún mayor el tiempo que hay que esperar entre la primera entrada y su correspondiente resultado. Cuando se van a procesar una gran cantidad de datos, que son independientes entre sí, los ciclos consumidos por la latencia se vuelven despreciables, pero cuando las nuevas entradas son dependientes de salidas anteriores, la latencia puede convertirse en un problema importante en el diseño de la arquitectura.

4. **Espacio**: Se define como el número de elementos que consume el diseño final. El espacio puede ser dado en distintos elementos dependiendo tanto de la tecnología como del dispositivo con el que se está comparando. Las principales unidades son:

   - el número de compuertas lógicas
   - el número de Slices
   - el número de CLB's
   - el número de registros, memorias y otros recursos empotrados.

5. **Tasa de operación**:Es la cantidad de información procesada por el sistema con respecto a una unidad de tiempo, en sistemas digitales se utiliza generalmente la medida de bits por segundo o algún derivado (bits por milisegundo, Gbits por segundo), aunque también se suele utilizar número de operaciones por unidad de tiempo.

6. **Eficiencia**: Esta métrica es la unión de las últimas dos, sería el throughtput por unidad de área, por ejemplo, bits por segundo por cierta cantidad de Slices. Dicha métrica es útil para determinar que tan eficientemente es utilizada el área del diseño.

2.4. Herramientas

En el desarrollo de diseños para FPGA's existen una gran variedad de herramientas para la captura del código, la síntesis y las simulaciones, al menos una por cada compañía desarrolladora. Lo anterior es porque 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. En esta sección pasaremos a describir las herramientas proporcionadas por Xilinx para sintetizar y simular nuestros diseños.

2.4.1. ISE

Es la herramienta de desarrollo proporcionada por Xilinx[41] para sus dispositivos, la versión más actual es la 13.2 soportando todas las familias hasta la **Virtex 7**. Las principales funciones que realiza la herramienta **ISE** son:

- Captura de código.
- Síntesis.
- Simulación.
- Generación de reportes de espacio y velocidad.
- Generación de gráficas de interconexión (se puede ver cómo quedan conectados los componentes).
- Proporciona herramientas gráficas para conexión de componentes y elaboración de maquinas de estados.

El ambiente de desarrollo se ilustra en la Figura 2.8.

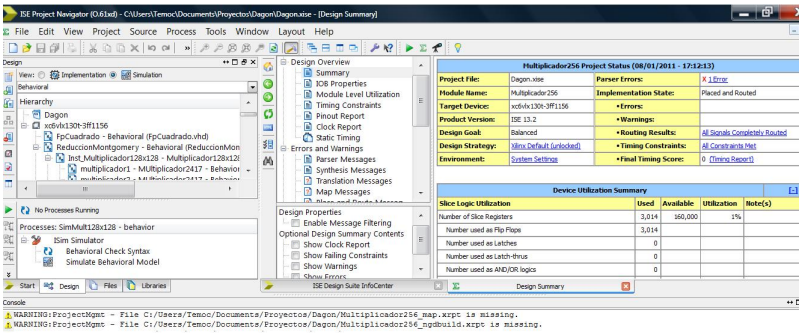

Figura 2.8: Herramienta de desarrollo *ISE*

.

2.4.2. ModelSim

La herramienta de desarrollo *ISE*, hasta su edición 10.3, proporciona también una herramienta de simulación conocida como *ISE simulator*, que aunque funciona relativamente bien, está muy limitada, pues es muy lenta, y carece de herramientas de depuración. Una solución alternativa que se encontró fue la de utilizar la herramienta de simulación creada por Menthor Graphics, llamada ModelSim [23], la cual es un simulador funcional que realiza la simulación a partir del código, y no de la síntesis, como lo hace ISE Simulator, lo que hace a ModelSim, más rápido y flexible, pues proporciona herramientas de depuración rápidas y eficientes, como visor de variables, puntos de ruptura, una rápida configuración del ambiente de trabajo y un fácil acceso a la jerarquía de componentes.

2.4.3. ISim

A partir de la versión 11.1 hasta la actual, ISE cambia de simulador, por uno más eficiente y amigable, conocido como ISim[43]. A pesar de que es una mejora considerable con respecto a lo que fue ISE Simulator, pues es más rápido, estable y tiene más herramientas, aún no es tan flexible como ModelSim, pues la depuración es más complicada, debido a que es más difícil la visualización de variables, y es aún más lento de lo que es ModelSim.

2.5. Sumario

En este capítulo se revisaron las principales caracterísitcas de los dispositivos *FPGA*, como lo son sus componentes, la forma en que se programan y las herramientas usadas

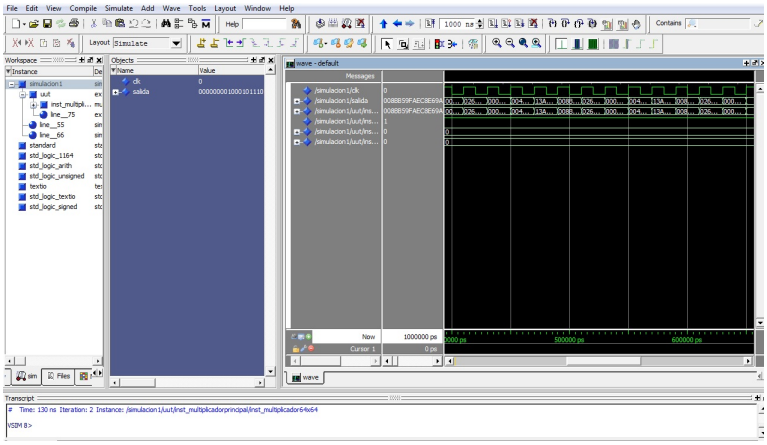

Figura 2.9: Ambiente del simulador ModelSim

.

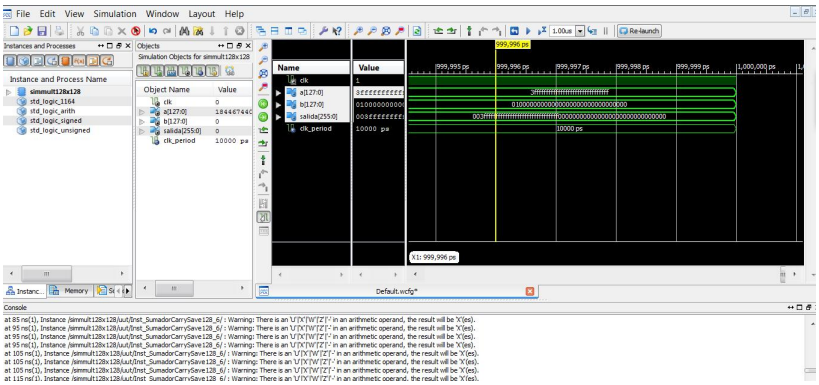

Figura 2.10: Ambiente del nuevo simulador ISim de Xilinx

.

para este fin, así como un pequeño recuento de su historia. Debido a sus características se determinó que la familia *Virtex 6* es la que más se adecúa a los requerimientos de este trabajo, por sus recursos, como los son las memorias *DualPortRam* y sus dispositivos *DSP48Slices*, que tienen mucho potencial para ser aplicados en operaciones aritméticas con números grandes.

Capítulo 3

Definiciones Básicas de Teoría de Números

3.1. Grupos

Un grupo es una entidad matemática denotada como $(\mathbb{G}, \cdot, e)$, donde $\langle \mathbb{G} \rangle$ es un conjunto de elementos, $\langle e \rangle$ un elemento del conjunto llamado *identidad*, y $\langle \cdot \rangle$ es una operación binaria entre los elementos del conjunto, que cumple con las siguientes propiedades [37]:

1. **Cerradura**: Sean $a, b \in \mathbb{G}$, entonces $a \cdot b \in \mathbb{G}$

2. **Asociatividad**: Sea $a, b, c \in \mathbb{G}$, entonces $(a \cdot\ b) \cdot c = a \cdot (b \cdot c)$

3. **Elemento Identidad**: Existe un elemento $e \in \mathbb{G}$ tal que con cualquier $a \in \mathbb{G}$ se cumple que $a \cdot e = e \cdot a = a$

4. **Existencia de Inversos**: Para cualquier elemento $a \in \mathbb{G}$ existe un elemento $a^{-1} \in \mathbb{G}$, tal que $a \cdot a^{-1} = e$

Se dice que un grupo es **Abeliano** si cumple con la propiedad de **Conmutatividad**, es decir si para todo $a, b \in \mathbb{G}$ se satisface $a \cdot b = b \cdot a$.

A continuación se definen algunos conceptos asociados con grupos:

1. **Orden del Grupo**: Dependiendo del conjunto $\langle \mathbb{G} \rangle$, un grupo puede ser finito o infinito. Sí es finito, al número de elementos en $\langle \mathbb{G} \rangle$ se le conoce como el **Orden del Grupo**, y en caso contrario se dice que el grupo es de orden infinito.

2. **Orden de un elemento**: El orden de un elemento $a \in \mathbb{G}$, es el número de veces que hay que aplicar la operación de grupo sobre dicho elemento para obtener como resultado el elemento identidad e, lo cual sólo es posible en un **Grupo Finito**.

3. **Elemento Generador**: Es aquel elemento con un orden igual al orden del grupo, lo cual significa que al aplicar la operación $\langle\cdot\rangle$ sobre sí mismo generará al resto de los elementos del grupo antes de dar como resultado el elemento $\langle e\rangle$.

4. **Grupo Cíclico**: Un grupo cíclico es aquel que posee al menos un elemento generador.

5. **Grupo Escrito de Manera Aditiva**: En un tipo de notación, donde la operación de grupo se escribe como $\langle+\rangle$, y el elemento identidad es comúnmente denotado por $\langle 0\rangle$[1].

   La aplicación de la operación de grupo k-veces sobre un mismo elemento a, se denota con el producto escalar: $\langle ka\rangle = a + a + a + a..... + a$, siendo $k \in \mathbb{Z}$ y $a \in \mathbb{G}$.

   Los inversos en esta notación se representan como un número negativo, es decir, para todo $a \in \mathbb{G}$ existe un $-a \in \mathbb{G}$.

6. **Grupo Escrito de Manera Multiplicativa**: En este caso, la operación de grupo y el elemento identidad se denotan por los símbolos $\langle\times\rangle$ y $\langle 1\rangle$ respectivamente. Para denotar la aplicación de la operación de grupo k-veces sobre un mismo elemento a se denota como $\langle a^k\rangle$ siendo $k \in \mathbb{Z}$ y $a \in \mathbb{G}$. Y los inversos se representan con un exponente negativo, es decir, $a^{-1} \in \mathbb{G}$, es el inverso de $a \in \mathbb{G}$.

7. **Subgrupo**: Sea $(\mathbb{G}, \cdot, e)$ un grupo y $\mathbb{H}$ un subconjunto de $\mathbb{G}$ entonces si $(\mathbb{H}, \cdot, e)$ forma un grupo se dice que es un subgrupo de $(\mathbb{G}, \cdot, e)$. Una de las propiedades de los subgrupos finitos es que su orden divide al orden de grupo que los contiene [37].

3.2. Campos Finitos

Un campo es una entidad matemática representadas como $\langle\mathbb{G}, +, 1, \times, 0\rangle$, que cumple con las siguientes propiedades[30]:

1. $\langle\mathbb{G}, +, 0\rangle$ forma un grupo abeliano.

[1]Hay que notar que la notación aditiva no implica que se hable del 0 de los números reales, o la operción de suma sobre los mismos

2. $\langle \mathbb{G} \setminus \{0\}, \times, 1, \rangle$ forma un grupo abeliano.

3. Satisface la ley distributiva $a \times (b + c) = a \times b + a \times c$ dónde $a, b, c \in \mathbb{G}$.

Algunas definiciones de teoría de números necesarias para el resto de la sección son las siguientes:

1. $\mathbb{N}$ es el conjunto de los números naurales, es decir $\{0, 1, 2, 3, 4, ...\}$

2. $\mathbb{Z}$ es el conjunto de los números enteros: $\{...-2356...-10, ...-2, -1, 0, 1, 2, ..,345..,1345...\}$.

3. **Divisibilidad** : Dados $a, b \in \mathbb{Z}$ con $a \neq 0$, se dice que a divide a b, y se denota como $a|b$, si existe un $q \in \mathbb{Z}$ tal que $b = qa$.

4. **División Entera**: Sea $a \in \mathbb{Z}$ y $b \in \mathbb{N}$, existen dos números $q, r \in \mathbb{Z}$, que satisfacen la igualdad $b = qa + r$, donde q y r son conocidos como el cociente y el residuo, respectivamente, para $0 \leq r < a$.

5. **Máximo Común Divisor**: El máximo común divisor de dos números $a, b \in \mathbb{Z}$ denotado por el $gcd(a, b)$(del inglés *Greatest Common Divisor*) está definido como el máximo número que divide a los enteros a y b. Dicho de otra forma, existe un número $d \in \mathbb{Z}$ tal que $d|a$ y $d|b$, es decir, d es un común divisior de a y b, y si para todo $c \in \mathbb{Z}$ tal que $c|a$ y $c|b$, si $c < d$ entonces d es el máximo común divisor de a y b.

6. **Número Primo**: Es aquel número $p \in \mathbb{Z}$, tal que sólo puede ser divisible por él mismo y por la unidad, por lo tanto para cualquier número $a \in [1, p - 1]$, se tiene que $gcd(p, a) = 1$.

7. **Primos Relativos**: Dos números $a, b \in \mathbb{Z}$ son primos relativos si cumplen con la condición de que $gcd(a, b) = 1$. De esta manera un número primo, es primo relativo de todos los números menores a él.

8. **Función de Euler**: Da como resultado la cantidad de primos relativos menores al número de entrada, se escribe como $\phi(n) = k$, que significa que el número n tiene $k < n$ primos relativos menores a él. En el caso de un número primo p se obtiene $\phi(p) = p - 1$.

9. **Operador Módulo**: Sean $a \in \mathbb{Z}$ y $b \in \mathbb{N}$, se puede escribir $b = qa + r$, donde $q, r \in \mathbb{Z}$, el operador de módulo(**mod**) regresa el valor del residuo r, y el operador de división entera (**div**) regresa el valor del cociente q.

10. **Congruencia**: Se dice que a es congruente con b módulo p, y se denota como $a \equiv b \bmod p$, si y sólo si $a \bmod p = b \bmod p$. Por ejemplo, $18 \equiv 29 \bmod 11$, por que $18 \bmod 11 = 29 \bmod 11 = 7$.

El conjunto de los números enteros partidos por la operación módulo con un primo p, denotado como $\mathbb{Z}_p = [0, p-1]$, forma un campo $\mathbb{F}_p$ debido a que:

- $\langle [0, p-1], +, 0 \rangle$, forman un grupo abeliano, donde $\langle + \rangle$ es la suma modular, es decir, $a + b \bmod p$, donde $a, b \in \mathbb{Z}_p$
- $\langle [1, p-1], \times, 1, \rangle$,forman un grupo abeliano, donde $\langle \times \rangle$ es el producto modular, es decir, $a \times b \bmod p$, donde $a, b \in \mathbb{Z}_p$.

Los campos finitos primos son un caso particular de los llamados **Campos de Galois**, a los que tambien pertenecen las **Extensiones de Campo**, llamadas así por que extienden los campos finitos primos. Un ejemplo se puede observar con los números complejos($\mathbb{C}$), que son una extensión de los números reales ($\mathbb{R}$). Los números complejos son polinomios de la forma $\mathbb{R}i + \mathbb{R}$, donde i es un elemento que no esta en $\mathbb{R}$, se puede seguir agregando n dimensiones para tener un campo $\mathbb{R}^n$. La misma idea puede ser aplicada a los campos finitos primos, que pueden extenderse utilizando una forma polinomial. Por ejemplo:

$$a(x) = a_0 + a_1 x + a_2 x^2 ..... a_{m-2} x^{m-2} + a_{m-1} x^{m-1}. \tag{3.1}$$

Donde $a_i \in \mathbb{F}_p$, y el campo donde se encuentran estos elementos se puede denotar como $\mathbb{F}_{p^m}$, que se lee como la m-ésima extensión del campo $\mathbb{F}_p$. Así como en los campos finitos primos, se necesita un primo que limite el tamaño del campo, en las extensiones se necesita un elemento que cumpla con la misma función. En este caso se usa un polinomio irreducible de la forma $\mathbb{F}_p[x]$ que no se encuentra en el campo $\mathbb{F}_{p^m}$ con la forma:

$$f(x) = 1 + f_0 x + f_1 x + ... + f_{m-1} x^{m-1} + x^m. \tag{3.2}$$

Este polinomio tiene la característica que al ser evaluado el resultado es impar (su coeficiente menos significativo es uno), y mónico(su coeficiente más significativo es uno). Se dice que es irreducible por que sus raíces no existen dentro del campo $\mathbb{F}_{p^m}$. Hay que denotar que estos campos poseen p^m elementos. Con base en lo anterior, otra forma de representar la extensión de campo es $\mathbb{F}_{p^m} = \mathbb{F}_p[x]/f(x)$, que se lee como la m-ésima extensión del campo $\mathbb{F}_p$ partido por el polinomio irreducible $f(x)$.

Es decir, las extensiones de campo son una generalización de los campos primos, donde $\mathbb{F}_p$ es $\mathbb{F}_{p^m}$ con $m = 1$. A $\mathbb{F}_p$ también se le puede llamar **Campo Base**, y a p la **Característica del Campo**.

3.3. Torres de Campo

Hoy en día las extensiones de campos finitos de caracterísitica 2 y 3 son las más utilizadas en implementaciones criptográficas, debido a la eficiencia de la aritmética en el campo base. Por otro lado en campos de característica $p > 2, 3$, la aritmética en el campo base se vuelve menos eficiente, pues hay que realizar sumas, productos y reducciones polinomiales con acarreos.

Además, conforme el tamaño de m aumenta, lo hace también la complejidad de la aritmética, por lo que el manejo de los polinomios puede resultar muy complicado.

Una solución del tipo "Divide y Vencerás", es la técnica de las llamadas **Torres de Campo** como se pueden ver en [3].

Usando la notación $\mathbb{F}_{p^m}[x]/f(x)$ la idea de las torres de campo es cambiar la representación de los polinomios en p^m, agrupándolos ahora en elementos en p^n, donde $n|m$. De esta forma la representación cambia a:

$$\mathbb{F}_{p^m} = \mathbb{F}_p[v]/h(v) \text{ ,donde } h(v) \in \mathbb{F}_{p^n}[v] \text{ es un polinomio de grado } m/n.$$
$$\mathbb{F}_{p^n} = \mathbb{F}_p[u]/g(u) \text{ ,donde } g(u) \in \mathbb{F}_p[u] \text{ es un polinomio de grado } n$$

De esta manera cambiamos la forma de agrupar un polinomio en $\mathbb{F}_p^m$, viéndolo ahora como un polinomio de m/n elementos en $\mathbb{F}_p^n$, simplificando así el manejo del polinomio. Además, es deseable que $m = 2^a 3^b$, donde $a, b \in \mathbb{Z}^+$, de esa forma siempre se estarán trabajando con trinomios y binomios, permitiendo aplicar trucos para multiplicar y elevar al cuadrado. Lo ideal es que se pueda volver a aplicar el mismo procedimiento para el campo $\mathbb{F}_p^n$, y así hasta llegar a elementos en el campo base, de ahí el término de **Torres de Campo**[28].

3.4. Criptografía de Curvas Elípticas

En 1985, Neal Koblitz [27] y Victor S. Miller [31] , propusieron en trabajos independientes, el uso de las curvas elípticas para la construcción de protocolos criptográficos. Una curva elíptica E sobre un campo K está definida por la ecuación de **Weierstrass**[15]:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{3.3}$$

Donde K puede ser el campo de los números reales $\mathbb{R}$, un campo finito primo $\mathbb{F}_p$ o una de sus extensiones $\mathbb{F}_{p^m}$ para $m \in \mathbb{Z}^+$. Además las constantes a_i para $i \in 1, 2, 3, 4, 6$, son elementos del campo sobre el que se definió la curva. Para los campos finitos con caracterísitica 2 la ecuación de **Weierstrass** se puede simplificar como:

$$E : y^2 + xy = x^3 + ax^2 + b \tag{3.4}$$

Para campos de característica 3 la simplificación es:

$$E : y^2 = x^3 + ax^2 + b \tag{3.5}$$

Y para campo finitos de característica $p > 3$, la ecuación simplificada queda de la siguiente forma:

$$y^2 = x^3 + ax + b \tag{3.6}$$

Este trabajo está enfocado a las curvas definidas sobre campos $\mathbb{F}_q$ para $q = p^m$ donde $p > 3$ $m \in \mathbb{Z}^+$. El conjunto de todos los puntos que satisfacen la ecuación de la curva E se denota como $E(\mathbb{F}_q)$, y cada punto de este conjunto puede verse como $P = (x, y)$, donde $x, y \in \mathbb{F}_q$.

Los puntos en las curvas elípticas forman un grupo abeliano que se puede escribir de forma aditiva, la operación de grupo es llamada (Suma de Puntos) y cumple con las siguientes propiedades:

- **Cerradura**: Sean $P, Q \in E(\mathbb{F}_q)$, $P + Q = R$, se cumple que $R \in E(\mathbb{F}_q)$.
- **Asociatividad**: Sean $P, Q, R \in E(\mathbb{F}_q)$, $(P + Q) + R = P + (Q + R)$. [2].
- **Conmutatividad**: Sean $P, Q \in E(\mathbb{F}_q)$, $P + Q = Q + P$
- **Elemento Identidad**: Para que funcione la aritmética con la ley de grupo de los puntos de la curva elíptica se define al elemento identidad como el elemento al infinito $\mathcal{O} = (\infty, \infty)$ que cumple con que $P + \mathcal{O} = \mathcal{O} + P = P$, siendo $P \in E(\mathbb{F}_q)$.
- **Existencia de Inversos**: El inverso de un punto $P = (x, y) \in E(\mathbb{F}_q)$, es el punto reflejado con respecto al eje de las ordenadas, es decir $-P = (x, -y)$ cumpliendo con que $P + (-P) = \mathcal{O}$. Es importante notar que existen puntos de la forma $P = (x, 0)$, lo que significa que el inverso del punto P es él mismo, es decir $P = -P$.

[2]La propiedad de asociatividad no es intuitiva a partir de la operación de grupo, por lo que si se desea ver una demostración detallada se recomienda consultar [15]

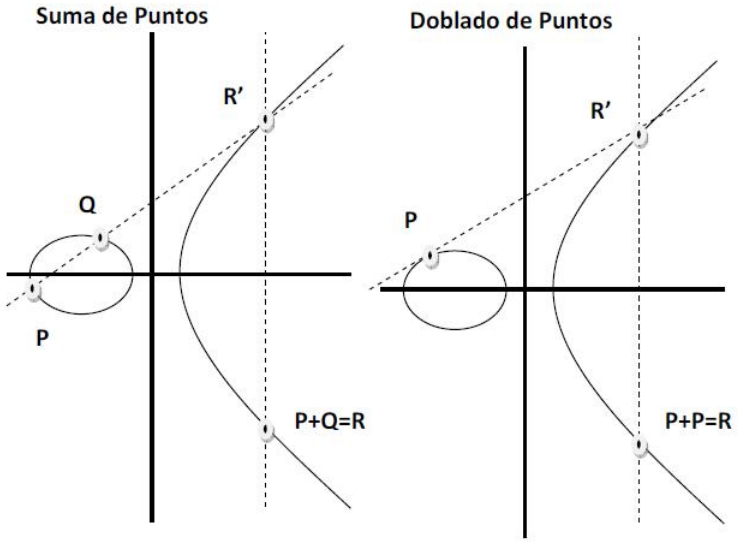

Figura 3.1: Suma y Doblado de puntos en curvas elípticas.

3.4.1. Ley de grupo: suma de puntos

Las curvas elípticas tienen la característica de ser **suaves**, es decir que su determinante es diferente de cero, implicando que si se traza una línea secante sobre la curva, siempre se intersecaran un máximo de 3 puntos.

Basándose en lo anterior se construye la ley de grupo para las curvas definidas sobre el campo de los números reales $\mathbb{R}$. Sean $P, Q \in \mathbb{R}$ los dos puntos que se quieran sumar, la aplicación de la ley de grupo se hace en dos pasos. El primero es el trazo de una recta secante que pase por los puntos P y Q, con lo que se obtiene un tercer punto R'. El segundo paso es el trazo de una recta vertical que interseca el punto R', esa recta a su vez interseca el reflejo del punto R' con respecto al eje de las ordenadas, llamado R, que es el resultado de la operación de suma, es decir $R = P + Q$.

Un caso particular de la suma de puntos, es cuando se realiza la suma de un punto $P \in \mathbb{R}$ consigo mismo, es decir el cálculo de $P + P = 2P$, este caso se le llama **Doblado de Punto**.

Para el doblado de punto la regla de grupo cambia ligeramente, pues ahora se traza una recta tangente al punto $P \in E(\mathbb{R})$, que interseca un punto $R' \in E(\mathbb{R})$, a partir de aquí se procede de la misma forma que con la suma de puntos, trazando una recta vertical que interseca el punto R', y a su reflejo con respecto al eje de las ordenadas, obteniendo el

punto $R = 2P$. Se puede observar la representación geométrica de la operación de grupo para los números reales en la figura 3.1.

Cuando la curva elíptica está definida sobre el campo de los números reales es más fácil observar la interpretación geométrica de la ley de grupo.

Sin embargo cuando la curva está definida sobre un campo finito $\mathbb{F}_q$, las características geométricas de la suma de puntos son más difíciles de observar, pero la aritmética para calcularla se mantiene, como se puede ver en los algoritmos 3.1 y 3.2, para la suma y el doblado respectivamente.

Algoritmo 3.1 Suma de Puntos para Campos de Característica $p \neq 2, 3$

Entrada: $E/\mathbb{F}_{p^m} : y^2 = x^3 + ax + b$, $P = (x_1, y_1), Q = (x_2, y_2) \in E(\mathbb{F}_{p^m})$
Salida: $R = P + Q = (x_3, y_3) \in E(\mathbb{F}_{p^m})$

1: $\lambda \leftarrow \frac{y_2 - y_1}{x_2 - x_1}$
2: $x_3 \leftarrow \lambda^2 - x_1 - x_2$
3: $y_3 \leftarrow \lambda(x_1 - x_3) - y_1$
4: **devolver** $R = (x_3, y_3)$

Como se puede ver en el Algoritmo 3.1, se basa en la ecuación de la recta para encontrar el punto de resultado R, calculando la pendiente λ y evaluándola donde se cruza con la curva E. Hay que recordar que las operaciones se realizan en un campo finito, por lo que la división que se usa para encontrar la pendiente es la búsqueda del inverso multiplicativo del denominador, por lo que la operación de suma tiene un costo de **1 inversión, y 2 multiplicaciones en el campo**.

Algoritmo 3.2 Doblado de Puntos para Campos de Característica $p \neq 2, 3$

Entrada: $E/\mathbb{F}_{p^m} : y^2 = x^3 + ax + b$, $P = (x_1, y_1) \in E(\mathbb{F}_{p^m})$
Salida: $R = 2P = (x_3, y_3) \in E(\mathbb{F}_{p^m})$

1: $\lambda \leftarrow \frac{3x_1^2 + a}{2y_1}$
2: $x_3 \leftarrow \lambda^2 - 2x_1$
3: $y_3 \leftarrow \lambda(x_1 - x_3) - y_1$
4: **devolver** $R = (x_3, y_3)$

En el Algoritmo 3.2, se observa que el doblado sólo requiere un punto para dar el resultado, es fácil ver que un doblado es más costoso que una suma, pues requiere **5 multiplicaciones y una inversión** en el campo base.

3.4.2. Multiplicación Escalar

Como la curva E está sobre un campo finito, la cantidad de puntos del conjunto $E(\mathbb{F}_q)$ también es finito, denotado como $\#E(\mathbb{F}_q)$. Además $\#E(\mathbb{F}_q)$ está acotado por el llamado **Intervalo de Hasse**[39] definido como:

$$q + 1 - 2\sqrt{q} \leq \#E(\mathbb{F}_q) \leq q + 1 + 2\sqrt{q} \tag{3.7}$$

De lo cual obtenemos que entre más grande sea q, más puntos habrá en la curva elíptica con los que se puedan trabajar, $\#E(\mathbb{F}_q) \approx q$ para q lo suficientemente grande. El orden del grupo esta compuesto de la siguiente manera:

$$\#E(\mathbb{F}_q) = hr \tag{3.8}$$

donde r es el factor primo más grande de $\#E(\mathbb{F}_q)$, y h es conocido como el *cofactor*. Los puntos, como elementos de un grupo tienen orden, es decir la cantidad de veces que hay que sumar el punto consigo mismo para que de como resultado el elemento $\mathcal{O}$.

$$kP = \mathcal{O} \tag{3.9}$$

En otras palabras, el orden de un elemento $P \in E(\mathbb{F}_q)$, es el valor de k necesario para satisfacer la ecuación anterior.

Es deseable que existiera un punto $P \in E(\mathbb{F}_q)$ con orden $\#E(\mathbb{F}_q)$, lo que significaría que al aplicar la multiplicación escalar sobre ese elemento con $1 \leq k \leq \#E(\mathbb{F}_q)$ nos daría como resultado el resto de los puntos en la curva, convirtiendo al grupo, en un grupo cíclico. Sin embargo para que lo anterior sea posible el cofactor debe cumplir con que $h = 1$, que no es muy común. Por lo que se trabaja con los puntos de orden r, que son un subgrupo de $\#E(\mathbb{F}_q)$, denotado como $E(\mathbb{F}_q)[r]$. Por lo tanto todos los puntos en $E(\mathbb{F}_q)[r]$ cumplen con:

$$rP = \mathcal{O} \tag{3.10}$$

donde $P \in E(\mathbb{F}_q)[r]$.

3.4.3. Problema del Logaritmo Discreto en Curvas Elípticas

Las propiedades de la multiplicación escalar con los puntos de una curva elíptica son usadas en el diseño de protocolos criptográficos.

Un ejemplo es el siguiente protocolo de secreto compartido entre dos entidades A y B, la solución propuesta usando curvas elípticas consiste en los siguientes pasos principales:

1. Ambas partes A y B acuerdan usar los puntos de la curva $E(\mathbb{F}_q)$, y el generador P del subgrupo $E(\mathbb{F}_q)[r]$, por lo que trabajarán con puntos de orden r.

2. Entonces la entidad A genera un escalar $a \in [1..r]$, aplica la operación escalar con el punto generador para obtener el punto $P_A = aP$. La entidad B hace algo similar para generar un punto $P_B = bP$, con $b \in [1..r]$.

3. Cada parte envía su punto a la otra, así la entidad A recibe el punto P_B, utilizando el escalar que generó para aplicar la multiplicación escalar sobre el punto recibido, para obtener $P_S = aP_B = abP$, la entidad B hace lo mismo, con lo cual ambas entidades tienen el punto P_S, obteniendo un secreto compartido.

El protocolo antes mencionado es conocido como **Protocolo de Diffie-Hellman para Curvas Elípticas** [25]. La forma en que un atacante pudiera obtener el secreto compartido entre A y B, es que interceptara los puntos cuando los intercambian, y obtuviera los escalares a y b, sin embargo, estos escalares no viajan en claro, sino en la forma de los puntos P_A y P_B. por lo que el atacante tendría que resolver la ecuación:

$$P_A = aP \tag{3.11}$$

El cual es un problema particularmente difícil, pues algo como la división de punto no está definido, y se vuelve más complicado si r, el orden de los puntos con los que se está trabajando, es muy grande. A este problema se le conoce como **Problema del Logaritmo Discreto Para Curvas Elípticas**, y es en el que se sustenta la seguridad para los protocolos basados en el uso de curvas elípticas.

3.5. Emparejamientos Bilineales

Los emparejamientos bilineales vinieron a revolucionar la criptografía de curvas elípticas, cuando en la década de los 90's, Alfred Menezes, Tatsuaki Okamoto y Scott Vanstone desarrollaron un método para reducir el problema de logaritmo discreto en curvas elípticas (conocido como ataque MOV) [29], al problema del logaritmo discreto en campos finitos. Este método está basado en el uso de los emparejamientos bilineales, y fué en un inicio un duro golpe a la seguridad de las curvas elípticas, pero después abrió un mundo de posibilidades, creando la Criptografía Basada en Emparejamientos. Un emparejamiento es una proyección de la siguiente forma:

$$\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_3 \tag{3.12}$$

Donde $\mathbb{G}_1$y $\mathbb{G}_2$ son grupos aditivos(como los puntos de una curva elíptica), y $\mathbb{G}_3$ son elementos de un grupo multiplicativo, si se escoge $\mathbb{G}_1 = \mathbb{G}_2$, se dice que es un emparejamiento simétrico. Además debe de cumplir con las siguientes propiedades que son:

- Computabilidad: Debe de existir una forma eficiente de calcular el emparejamiento.
- No degenerado: sea $P \in \mathbb{G}_1$ y $Q \in \mathbb{G}_2$ entonces $\hat{e}(P, Q) = 1$ si y sólo si P o Q son el elemento identidad del grupo al que pertenezcan, en otro caso $\hat{e}(P, Q) \neq 1$.
- Bilenealidad: sean $P_1 \in \mathbb{G}_1$ y $Q_1, Q_2 \in \mathbb{G}_2$, entonces:

$$\hat{e}(P_1, Q_1 + Q_2) = \hat{e}(P_1, Q_1)\hat{e}(P_1, Q_2)$$
$$\hat{e}(P_1 + P_2, Q_1) = \hat{e}(P_1, Q_1)\hat{e}(P_2, Q_1)$$

Esta última propiedad es especialmente útil en criptografía de curvas elípticas, pues deriva en la propiedad *inmediata* definida como: Sea P un punto generador del campo $\mathbb{G}_1$, que en este caso, $\mathbb{G}_1 = \mathbb{G}_2$, Q_1 y Q_2 dos puntos cualesquiera de $\mathbb{G}_1$, entonces:

$$\hat{e}(Q_1, Q_2) = \hat{e}(k_1 P, k_2 P) = \hat{e}(P, P)^{k_1 k_2} = g^{k_1 k_2} \tag{3.13}$$

Donde g es un generador en el grupo escrito de forma multiplicativa y es el resultado de $\hat{e}(P, P)$. Además k_1 y k_2 son las constantes mediante las cuales se puede generar Q_1 y Q_2 respectivamente a partir de P. La característica anterior, es la que hace a los emparejamientos tan atractivos para la criptografía, pues crea nuevas oportunidades para el desarrollo de los futuros esquemas de seguridad.

3.5.1. Definiciones

Grado de Encajamiento: Si la curva elíptica con la que se está trabajando está definida sobre un campo $\mathbb{F}_q$, donde $q = p^m$, entonces el resultado del emparejamiento bilineal es un elemento en una extensión k-ésima de este campo, es decir, son elementos en un campo $\mathbb{F}_{q^k}$. Ese valor k, es conocido como el **Grado de Encajamiento**. El grado de encajamiento se define como el mínimo valor de k que cumple con la siguiente condición:

$$r | q^k - 1 \tag{3.14}$$

Donde r es el oden del subgrupo de puntos con los que se está trabajando, y q define al campo finito $\mathbb{F}_q$ sobre el que está definida la curva elíptica.

Raíces de la Unidad: Un elemento $g \in \mathbb{F}_q$ es la k-ésima raiz de la unidad, si al aplicar sobre éste la operación de multiplicacion k-veces da como resultado 1, es decir, $g^k = 1$. Es por esta razón que dice que el resultado del emparejamiento son las r-ésimas raíces de la unidad del campo $\mathbb{F}_{q^k}$.

3.6. Curvas Amables con Emparejamientos Bilineales

El principal parámetro para elegir trabajar con una curva u otra al utilizar emparejamientos, es el grado de encajamiento, pues como ya se mencionó, éste depende de la curva, y nos dirá qué tan grande será el resultado del emparejamiento. Este resultado debe de ser lo suficientemente grande para que no se pierda la seguridad del problema del logaritmo discreto, pero no demasiado como para que los valores que haya que manejar sean intratables.

Según se explica en [21], el grado de encajamiento debe de cumplir las siguientes condiciones:

1. El orden r de los puntos cumple con la condición de que $r \geq \sqrt{q}$ y que $r|\#E(\mathbb{F}_q)$.
2. El grado de encajamiento k con respecto a el orden de los puntos r, cumple con $r < \log_2(r)/8$.
3. Sea el campo $\mathbb{F}_q$ sobre el que está definida la curva E, donde $q = p^m$, y k el grado de encajamiento, entonces $m \cdot k = 2^a 3^b$, donde $a, b \in \mathbb{N}$, para que se pueda construir una torre de campo eficiente.

Las condiciones anteriores dan una idea del tamaño y forma que debe de tener un grado de encajamiento, para que se pueda considerar "amable con los emparejamientos bilineales".

3.6.1. Curvas de Barreto-Naehrig

Uno de los avances importantes más recientes en la criptografía basada en emparejamientos ha sido el descubrimiento de las Curvas de Barreto-Naehrig, también llamadas Curvas BN, por Paulo S. M. L. Barreto y Michael Naehrig en el 2005 [4]. La ventaja principal de estas curvas ordinarias es que su diseño es muy sencillo, y cuando trabajamos con un campo definido sobre un primo de 256 bits, obtenemos aproximadamente 128 bits de seguridad, tanto del lado de las curvas elípticas como en el resultado del emparejamiento.

La forma de construirlas es mediante las siguientes 3 ecuaciones:

$$p = p(t) = 36t^4 + 36t^3 + 24t^2 + 6t + 1$$
$$r = r(t) = 36t^4 + 36t^3 + 18t^2 + 6t + 1$$
$$tr = tr(t) = 6t^2 + 1$$

En donde nuestra tarea consiste en buscar un valor de t para el cual la evaluación de $p(t)$ y $r(t)$ resulten valores primos, siendo p el primo que define el campo sobre el que está la curva y r el orden de los puntos. Como podemos ver si usamos un parámetro t de 64 bits, el orden de nuestros puntos será de 256 bits, lo que nos garantiza 128 bits de seguridad al trabajar con los puntos de la curva, y gracias a esta construcción, el grado de encajamiento que tenemos es de 12, con lo que también tenemos 128 bits de seguridad en el resultado del emparejamiento que habita en el subgrupo de las raíces r-ésimas de la unidad en la extensión de campo $\mathbb{F}^*_{p^k}$. Por otro lado, a partir del **Intervalo de Hasse**[39], se obtiene que $\#E = | \leq q \pm 2\sqrt{q} + 1|$, donde $q = p^m$. A la diferencia entre $\#E$ y q, se le conoce como traza, es denotada como $tr \leq |\pm 2\sqrt{q}+1|$ y en las curvas de Barreto-Naehrig se cumple que $tr = 6t^2 + 1$.

3.7. Sumario

A través de este capítulo se revisaron los conceptos básicos sobre los que se construyen las primitivas utilizadas para la criptográfia de curvas elípticas, como lo son las definiciones de grupos y campos finitos, así como el concepto de Torre de Campo, necesario para las operaciones con extensiones de campos finitos primos. Posteriormente se describen la criptografía de curvas elípticas, sus fundamentos, y el problema díficil en el que se sostienen. Para terminar se da una breve introducción a lo que son los emparejamientos bilineales y las características de las curvas “amables” con este tipo de operaciones.

Capítulo 4

Aritmética Computacional

En esta sección se describe una variedad de algoritmos de aritmética computacional útiles para aplicaciones criptográficas de clave pública, enfocándose en realizar sumas, restas, multiplicaciones y reducciones modulares de manera eficiente. Es importante mencionar que sólo se describen los algoritmos utilizados para operaciones en los campos $\mathbb{F}_p$, que son los que se requieren en este trabajo, dejando de lado las operaciones que se realizan sobre las extensiones de campo $\mathbb{F}_{p^k}$

4.1. Algoritmos de Suma y Resta

Los algoritmos de suma y resta modular son algoritmos básicos usados en las implementaciones criptográficas de llave pública, debido a que éstos usan constantemente aritmética de campos finitos, y al final todas las operaciones involucran de manera o de el cómputo de sumas y restas.

El algoritmo 4.1 es el algoritmo de suma modular más simple, la idea que se explota allí es no permitir que el resultado salga del campo en el que se está trabajando, lo cual se logra de una manera simple con una verificación y una resta.

Sin embargo, el algoritmo 4.1 no se puede aplicar de forma directa sobre números relativamente grandes (más de 128 bits) debido al tamaño de palabra fijo de las arquitecturas comunes en plataformas de software.

Para dar solución a ese problema se usa la llamada representación multiprecisión, donde $A \in \mathbb{Z}$, se representa como un número en base $b \in \mathbb{Z}$, que generalmente es una potencia de 2 y $b-1$ es el máximo valor que puede representar con una palabra de procesador. De esta manera el número puede ser representado como una secuencia de palabras o dígitos en base b como se muestra a continuación:

Algoritmo 4.1 Algoritmo Básico de Suma Modular

Entrada: $x, y \in \mathbb{F}_p$ y p
Salida: $x + y \mod p$

```
w ← x + y
si w > p entonces
  w ← w − p
fin si
devolver w
```

$$X = \sum_{i=0}^{n-1} X_i \cdot b^i = (X_{n-1}, X_{n-2}, ..., X_1, X_0)_b \tag{4.1}$$

Utilizando esta representación multiprecisión para los números en $\mathbb{F}_p$, el algoritmo 4.2 señala como realizar la operación de suma modular multiprecisión.

Algoritmo 4.2 Algoritmo de Suma Modular Multiprecisión

Entrada: $X = (x_{n-1}, ...x_1, x_0)_b$, $Y = (y_{n-1}, ...y_1, y_0)_b$,$X, Y \in \mathbb{F}_p$
Salida: $X + Y \mod p = W = (w_{n-1}, ..., w_1, w_0)_b$

```
acarreo ← 0
para i=0 to n-1 hacer
  w_i ← x_i + y_i + carry
  si w_i > b entonces
    acarreo ← 1
  si no
    acarreo ← 0
  fin si
  w_i ← w_i mod b
fin para
si w > p entonces
  w ← w − p
fin si
devolver w
```

El algoritmo 4.2, realiza la suma de los dos números de entrada palabra por palabra, y en cada suma parcial se realiza la verificación de que no haya ocurrido algún acarreo, y en caso de existir, éste se almacena en la variable *acarreo* para ser utilizado como entrada

de la suma siguiente. Al final se realiza una última verificación para asegurarse de que el resultado aún esté dentro del campo $\mathbb{F}_p$, en caso de no ser así, con únicamente una resta se tiene el resultado correcto. Lo anterior funciona debido a que el máximo valor que pueden tomar X y Y es $p-1$, lo que implica que el máximo valor de su suma es $2p-2$, al cual basta con restarle una vez p, para obtener $p-2$, el cual es un número válido en $\mathbb{F}_p$. Hay que notar que para realizar el último ajuste se necesita realizar una resta modular multiprecisión, la cual se describe en el algoritmo 4.3.

Algoritmo 4.3 Algorimo Multiprecisión de Resta

Entrada: $X = (x_{n-1}, ...x_1, x_0)$, $Y = (y_{n-1}, ...y_1, y_0)$,$X, Y \in \mathbb{F}_p$
Salida: $X - Y \mod p = W = (w_{n-1}, ..., w_1, w_0)$

```
préstamo ← 0
para i=0 to n hacer
  w_i ← x_i − y_i + préstamo
  si w_i < 0 entonces
    préstamo ← 1
  si no
    préstamo ← 0
  fin si
  w_i ← w_i mod b
fin para
si w < 0 entonces
  w ← w + p
fin si
devolver w
```

El algoritmo 4.3 funciona de manera muy similar al algoritmo 4.2, pues también realiza la operación aritmética palabra a palabra, sólo que en este caso es la resta la operación que se efectúa, además de que ahora la variable *préstamo* toma valores de 0 y -1, a diferencia de su contraparte que era la variable *acarreo*. Otra similitud es que al final también se hace una comparación, pero ahora lo que se verifica es que el valor resultante no sea un número negativo, los cuales no estan definidos en $\mathbb{F}_p$. Para que el resultado esté en $\mathbb{F}_p$ basta con restarle una vez el número p, esto funciona por que el mínimo valor que puede tomar X es 0, y el máximo que puede tomar Y es $p-1$, por lo que el mínimo resultado que se puede obtener es $-(p-1)$, y sumando p a este último obtenemos 1, que está en el campo $\mathbb{F}_p$. Si se observan los algoritmos de suma y resta, se puede notar que uno requiere del otro, por lo que a primera vista puede dar la impresión de que pueden llegar

a ciclarse, sin embargo por las acotaciones de los números de entrada ya mencionados anteriormente resulta imposible que la resta invocada por el algoritmo de suma, vuelva a llamar al algoritmo de suma, y viceversa.

4.1.1. Sumadores de acarreo almacenado(*Carry-Save Adders*)

Siendo p el primo que define el campo $\mathbb{F}_p$, entonces todas las sumas y restas que se realizan dentro del campo tienen una complejidad de $O(\log_2 p)$, en otras palabras, son operaciones lineales con respecto al número de bits de los operandos de entrada. Esta dependencia es causada por los acarreos, debido a que no se puede calcular la suma de un par de bits hasta no haber obtenido el acarreo resultante de la suma anterior. En la imagen 4.1 se puede observar la disposición de un arreglo de sumadores completos(*Full Adders*), que están interconectados por la línea de *carry*, de forma que para poder calcular el resultado de la suma, hay que esperar a que el acarreo de todos los sumadores se propague completamente.

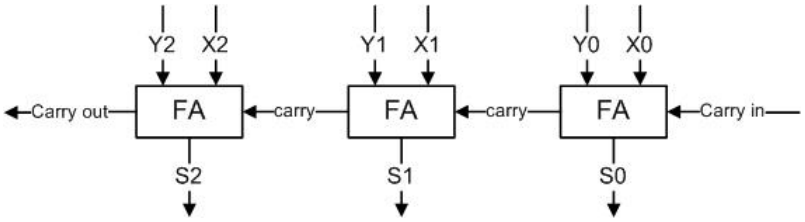

Figura 4.1: Arquitectura de un arreglo de sumadores completos

.

Una alternativa para poder reducir el tiempo de esta propagación es usar la representación de acarreo almacenado o **Carry-Save**, la cual consiste en dividir el resultado de las sumas en 2, por un lado en una variable C se almacenan los acarreos, y en otra variable S el resultado de las sumas sin acarreo, de forma que, se puede realizar todo el cálculo en paralelo, tomando un tiempo $O(1)$ para obtener los resultados C y S, como se puede ver en la figura 4.2.

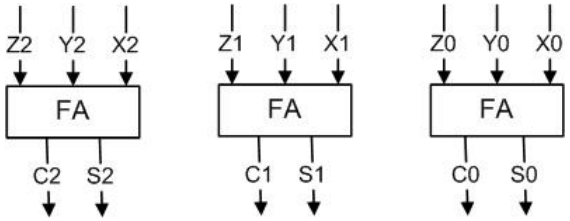

Figura 4.2: Arquitectura de sumadores en configuración de salvado de acarreo

.

Los bloques de esta figura realizan las siguientes operaciones booleanas:

$$S_i = X_i \oplus Y_i \oplus Z_i$$
$$C_i = (X_i \wedge Y_i) \vee (Y_i \wedge Z_i) \vee (Z_i \wedge X_i)$$

Para poder seguirlos tratando, se debe de hacer la suma completa entre la variable C y S, porque en algún momento se podrá realizar la propagación de los acarreos. Sin embargo la ventaja viene en que se puede mantener los valores de C y S por separado mientras sea conveniente, pues unirlos representa el costo de una suma $(C + S)$, mientras que el resultado que se obtiene es el de 2 sumas $(X + Y + Z)$, con lo que reducimos a la mitad el costo de la adición en cuanto a tiempo se trata, tratándose de más de dos operandos.

4.2. Multiplicación Modular

La multiplicación modular es la base los protocolos de llave pública actuales, ya sea como parte de la exponenciación como es el caso de aquellas que se basan en el **Problema del Logaritmo Discreto**, o en el producto escalar de puntos, para la criptografía de curvas elípticas. Esta operación se define como el resultado de $x \cdot y \mod p$, donde $x, y \in \mathbb{F}_p$ y p es un primo que define el campo. A simple vista la multiplicación modular requiere de una multiplicación completa, y de una división, para obtener el residuo de la división de $\frac{x \cdot y}{p}$. Sin embargo, no se requiere una división completa, sólo es de interés el residuo. Pensando en esta idea se han desarrollado una gran variedad de algoritmos para realizar esta reducción eficientemente. No se puede decir que exista el mejor algoritmo de producto modular, si no que bajo ciertas condiciones, algunos algoritmos resultan más eficientes que otros, permitiendo cierta libertad de elección, y dando la facilidad de combinarlos en ciertas ocasiones.

Para realizar el producto modular la operación suele dividirse en 2 etapas, que son la propia multiplicación de $x \cdot y$, y la reducción del resultado módulo p. A continuación se describen los algoritmos para cada una de esas dos etapas.

4.2.1. Algoritmo de multiplicación clásico

Este es el algoritmo de multiplicación más elemental de todos, llamado en ocasiones como el método de *Libro de Texto*.

En el algoritmo 4.4 se muestra cómo realizar el algoritmo de *Libro de texto* utilizando una representación de multiprecisión para los operandos.

Algoritmo 4.4 Algoritmo de *libro de texto* para realizar la multiplicación

Entrada: $X = (x_{n-1}, x_{n-2}...x_1, x_0)_b$, $Y = (y_{n-1}, y_{n-2}...y_1, y_0)_b, X, Y \in \mathbb{F}_p$
Salida: $X \cdot Y = W = (w_{2n-1}, w_{2n-2}, ..., w_1, w_0)$

```
para i=0 to 2n-1 hacer
    w_i ← 0
fin para
para i=0 to n-1 hacer
    c ← 0
    para j=0 to n-1 hacer
        (u, v) ← w_{i+j} + x_i · y_j + c
        w_{i+j} ← v
        c ← u
    fin para
    w_{j+n-1+1} ← c
fin para
devolver W = (w_{2n-1}, w_{2n-2}, ..., w_1, w_0)
```

Aunque en primera instancia el algoritmo 4.4 pudiese parecer complicado, simplemente consiste en realizar el producto de todos los dígitos de X con todos los dígitos de Y, para después sumarlos, de forma que al final se obtiene el producto $W = X \cdot Y$, que tiene el doble de dígitos que X y Y, es decir si cada uno de estos tiene n dígitos, W tendría $2n$ dígitos. Es importante notar que la complejidad de este algoritmo es cuadrática con respecto al número de bits de los operandos que están en $\mathbb{F}_p$, es decir $O(\log_2^2 p)$, debido a que cada vez que un bit del operando X se multiplica con todos los bits del operando Y. Lo mismo pasa en el caso de los dígitos, en dónde cada dígito de X tienen que ser multiplicado con todos los dígitos de Y. Este método es simple y requiere de poco almacenamiento, por lo

que es muy utilizado cuando se pueden realizar las multiplicaciones parciales de manera muy rápida o hay poco espacio para almacenamiento.

4.2.2. Algoritmo de multiplicación de Karatsuba

Un de los inconvenientes del método de *libro de texto* es su complejidad cuadrática, debido a que es una operación muy usada para otros algoritmos, como es el caso de la exponenciación. Por mucho tiempo se creyó que no se podía reducir la complejidad cuadrática del algoritmo de multiplicación, hasta que en 1960 Anatolii Alexeevitch Karatsuba propuso una nueva forma de realizar esta operación reduciendo su complejidad.

El Algoritmo de Karatsuba es del tipo "divide y vencerás", en el cual consiste en dividir sus los operandos de la siguiente forma:

$$x = xh \cdot 2^{\ell} + xl$$
$$y = yh \cdot 2^{\ell} + yl$$

En donde x y y son números en representación binaria, de n bits y $\ell = n/2$. En otras palabras xh y xl son las partes alta y baja del número x respectivamente, y lo mismo pasa con los yh y yl para y. Utilizando esta representación, el producto se puede ver de la siguiente forma:

$$\begin{aligned} x \cdot y &= (xh \cdot 2^{\ell} + xl)(yh \cdot 2^{\ell} + yl) \\ &= 2^{2\ell} \cdot xh \cdot yh + 2^{\ell}(xh \cdot yl + xl \cdot yh) + xl \cdot yl \end{aligned}$$

Donde podemos agrupar los productos de la siguiente forma:

$$z_0 = xh \cdot yh$$
$$z_1 = xh \cdot yl + xl \cdot yh$$
$$z_2 = xl \cdot yl$$

Se requieren 4 multiplicaciones de ℓ bits para realizar la multiplicación, sin embargo a partir de los productos anteriores se puede observar que existe la siguiente equivalencia:

$$\begin{aligned} z_1 &= xh \cdot yl + xl \cdot yh \\ &= xh \cdot yl + xl \cdot yh + xh \cdot yh + xl \cdot yl - xh \cdot yh - xl \cdot yl \\ &= (xh + xl) \cdot (yh + yl) - z_0 - z_2 \end{aligned}$$

De esta forma es posible reescribir el producto $x \cdot y$ como:

$$\begin{aligned} x \cdot y &= (xh \cdot 2^{\ell} + xl)(yh \cdot 2^{\ell} + yl) \\ &= 2^{2\ell} \cdot z_2 + 2^{\ell}((xh + xl) \cdot (yh + yl) - z_0 - z_2) + z_0 \end{aligned}$$

Como se puede ver, el número de productos necesarios para la multiplicación se ha reducido de 4 a 3, y se puede usar el mismo método para realizar los subproductos, por lo que se realizaría la multiplicación con 9 productos con operandos de $n/4$ bits, y así sucesivamente hasta llegar a multiplicaciones de un sólo bit, por lo que se requieren $3^{\log_2 n}$ multiplicaciones, pero como se cumple que $3^{\log_2 n} \leq 3n^{\log_2 3}$ [1], entonces la complejidad del algoritmo es $O(n^{\log_2 3})$, que es claramente menor a la complejidad $O(n^2)$ del algoritmo clásico de multiplicación.

4.2.3. Variantes del algoritmo de Karatsuba

La versión descrita por primera vez del algoritmo de Karatsuba utiliza la partición binaria de los operandos, sin embargo también funciona si se dividen en 3, 4, 5, etc, y resultará eficiente dependiendo del tamaño en bits de las entradas originales, a estas variantes se les conoce como multiplicadores a la Karatsuba. Un inconveniente que presenta el algoritmo de Karatsuba es que para unir los subproductos son necesarias varias sumas, y manejar los acarreos resultantes hace que la ventaja sobre el algoritmo de multiplicación clásico sea mínimo.

Por otro, lado las multiplicaciones entre polinomios no tienen acarreos, lo que los hace idóneos para utilizar el algoritmo de Karatsuba, precisamente basado en esta idea se basa el trabajo desarrollado por Peter L. Montgomery, que en 2005 [32] presentó un conjunto de fórmulas para calcular la multiplicación entre polinomios de grado 4, 5 y 6 con un número mínimo de productos. Por ejemplo en el algoritmo 4.5 se puede observar la propuesta que hizo de multiplicación para polinomios de grado 4, es decir, con 5 coeficientes, que da como resultado un polinomio de 9 coeficientes. Este producto mediante el método clasico

necesita 25 productos entre los coeficientes, mientras que la propuesta de [32] únicamente requiere de 13 productos y 73 sumas, sin embargo el precio que se paga es una gran cantidad de sumas, y la necesidad de una buena cantidad de variables temporales, y una pérdida en la regularidad del algoritmo de Karatsuba.

Algoritmo 4.5 Algoritmo de multiplicación a la Karatsuba propuesto en [32] para polinomios de grado 4

Entrada: $a(x) = \sum_{i=0}^{4} a_i x^i$ y $b(x) = \sum_{i=0}^{4} b_i x^i$
Salida: $c(x) = \sum_{i=0}^{i} = 8c_i x^i$

$p_0 \leftarrow (a_0 + a_1 + a_2 + a_3 + a_4)(b_0 + b_1 + b_2 + b_3 + b_4)$
$p_1 \leftarrow (a_0 - a_2 - a_3 - a_4)(b_0 - b_2 - b_3 - b_4)$
$p_2 \leftarrow (a_0 + a_1 + a_2 - a_4)(b_0 + b_1 + b_2 - b_4)$
$p_3 \leftarrow (a_0 + a_1 - a_3 - a_4)(b_0 + b_1 - b_3 - b_4)$
$p_4 \leftarrow (a_0 - a_2 - a_3)(b_0 - b_2 - b_3)$
$p_5 \leftarrow (a_1 + a_2 - a_4)(b_1 + b_2 - b_4)$
$p_6 \leftarrow (a_3 + a_4)(b_3 + b_4)$
$p_7 \leftarrow (a_0 + a_1)(b_0 + b_1)$
$p_8 \leftarrow (a_0 - a_4)(b_0 - b_4)$
$p_9 \leftarrow (a_4)(b_4)$
$p_{10} \leftarrow (a_3)(b_3)$
$p_{11} \leftarrow (a_1)(b_1)$
$p_{12} \leftarrow (a_0)(b_0)$
$c_0 \leftarrow p_{12}$
$c_1 \leftarrow p_7 - p_{11} - p_{12}$
$c_2 \leftarrow p_2 - p_5 - p_7 - p_8 + p_9 + 2p_{11} + p_{12}$
$c_3 \leftarrow p_0 - p_1 - 2p_2 + p_3 + 2p_5 - p_6 + 3p_8 - 3p_9 - 2p_{11} - 2p_{12}$
$c_4 \leftarrow -p_0 + 2p_1 + 2p_2 - 2p_3 - p_4 - p_5 + p_6 + p_7 - 4p_8 + 3p_9 + p_{10} + p_{11} + 3p_{12}$
$c_5 \leftarrow p_0 - 2p_1 - p_2 + p_3 + 2p_4 - p_7 + 3p_8 - 2p_9 - 2p_{10} - 3p_{12}$
$c_6 \leftarrow p_1 - p_4 - p_6 - p_8 + p_9 + 2p_{10} + p_{12}$
$c_7 \leftarrow p_6 - p_9 - p_{10}$
$c_8 \leftarrow p_9$
devolver $c(x)$

4.2.4. Algoritmos de reducción con y sin restauración

Ya se explicaron algunas variantes de la multiplicación, ahora es el momento de describir los algoritmos de reducción. El primero en la lista es el algoritmo de reducción con restauración. El algoritmo asume que la división es una serie de restas. Por ejemplo, si queremos reducir un resultado $w = kp + r$, donde k es el cociente y r es el residuo, es decir, el resultado de la operación módulo, por lo que una reducción significa evaluar $r = w - kq \equiv w \bmod p$. En la reducción con restauración lo que se busca es realizar esas k restas sin tener que calcular una división, y eso se logra mediante el algoritmo 4.6. Este algoritmo reduce la cantidad de restas utilizadas, pues en vez de realizar k restas, se realizan $\log_2 k$ restas, o en el caso general, n substracciones.

Algoritmo 4.6 Algoritmo de reducción con restauración

Entrada: $p \in \mathbb{Z}$, donde $n = \log_2 p$ y W donde $\log_2 W \leq 2n - 1$
Salida: $W \mod p$

```
p ← p · 2^n
para i=0 to n hacer
    Wtemp ← W − P
    si Wtemp > 0 entonces
        W ← Wtemp
    fin si
    p ← p/2
fin para
devolver W
```

El algoritmo 4.6 al restar $p \cdot 2^n$ garantiza que el resultado W ahora tenga $2n - 2$ bits, por lo que en cada iteración el número W es reducido en un bit, así al final W tendrá a lo más el mismo número de bits que p, y $W < p$, con lo que $W \in \mathbb{F}_p$. Hay que notar que el resultado de las restas puede dar un resultado negativo, por lo que seguir restando podría provocar un error, para evitarlo, inmediatamente despúes de una resta, si hay un resultado negativo se deshace el paso anterior, de ahí el nombre de "con restauración", garantizando que no se generan valores negativos.

Sin embargo la restauración representa la necesidad de tener una variable extra para trabajar, que si se quiere evitar se puede hacer uso de una variante llamada "reducción sin restauración", mostrada en el algoritmo 4.7.

Este algoritmo sin restauración usa la misma idea de reducir el elemento de entrada W, pero cuando en la resta se obtiene un resultado negativo, en la siguiente iteración en vez de realizar una resta se hace una suma, por lo que es conveniente si se quiere ahorrar

Algoritmo 4.7 Algoritmo de reducción sin restauración

Entrada: $p \in \mathbb{Z}$, donde $n = \log_2 p$ y W donde $\log_2 W \leq 2n - 1$
Salida: $W \mod p$

```
p ← p · 2^n
para i=0 to n hacer
  si W > 0 entonces
    W ← W − P
  si no
    W ← W + P
  fin si
  p ← p/2
fin para
devolver W
```

espacio, pero para números muy grandes hay que tener definidas tanto la operación de suma como de resta de múltiple precisión.

4.2.5. Algoritmos de reducción rápida

Las reducciones con y sin restauración funcionan bien sin importar el número p que se esté usando, sin embargo se puede observar que tiene una complejidad cuadrática con respecto al número de bits $n = \log_2 p$, pues siempre se necesitarán realizar n sumas, y muchas comparaciones. Pero, aunque estas reducciones son eficientes existe una mejor manera de hacerlo, y es mediante una reducción rápida, valiéndose de la selección de p con la forma:

$$p = 2^n + 2^k + 1$$

El número p tiene un peso de Hamming[1] muy bajo, lo que significa que cada vez que se hace una iteración de la reducción (con o sin restauración), serán muy pocos bits los afectados en el operando de entrada (exactamente 3 en el caso de p en la ecuación de arriba), y la forma en cómo afecten los bits menos significativos al operando de entrada dependerá de la parte alta del mismo. Por ejemplo si se intenta reducir con $p = 2^n + 1$, en la primera iteración del la reducción, el bit $2n - 1$ del operando de entrada se volverá 0, y su valor será restado al bit en la posición $n - 1$, lo mismo pasará con el resto de los bits,

[1]Es llamado peso de Hamming a la cantidad de bits con valor 1 que tiene un número en su representación binaria y se suele denotar como $H(p)$.

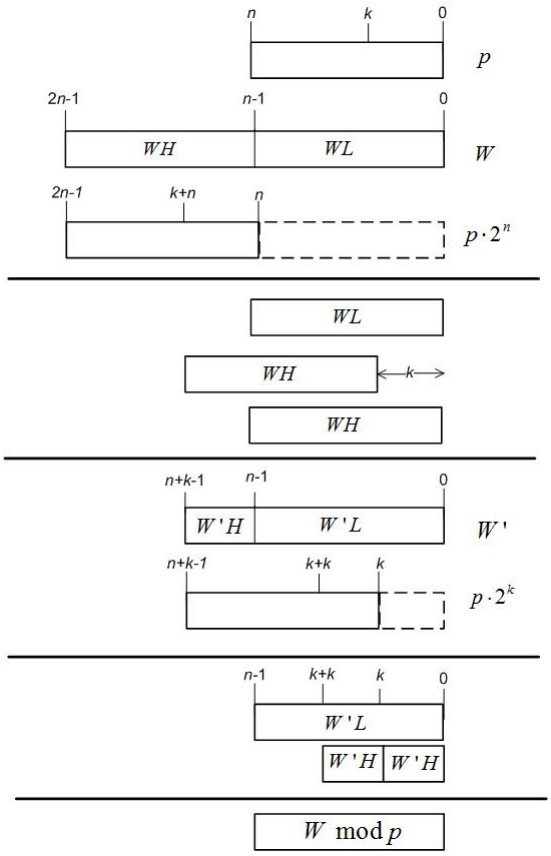

Figura 4.3: Reducción rápida para primos con la forma $2^n + 2^k + 1$

por lo que al final, la reducción se verá como si se le restaran los n bits más significativos a los n bits menos significativos, es decir, la parte baja menos la parte alta del operando de entrada. Para el caso de $p = 2^n + 2^k + 1$ ocurre algo similar, pero ahora a la parte baja, se le resta la parte alta dos veces, una sin desplazamientos y otra con desplazamientos, como se puede apreciar en la figura 4.3.

Como se puede ver la reducción con estos números solo cuestan algunas sumas, es decir la reducción necesita un número constante de sumas, con lo que la reducción se vuelve de complejidad $O(n)$. En el caso particular de la figura 4.3, se cumple que $k < n/2$, en caso contrario, se necesitarían más sumas para la reducción. De la misma manera conforme se eleva el peso de Hamming de p, éste pasa a representarse como $p = 2^n + s$, y se requieren más sumas para reducir W.

La reducción rápida se puede generalizar como se ve en el algoritmo 4.8, por ejemplo, la

Algoritmo 4.8 Generalización del algoritmo de reducción rápida

Entrada: $W = X \cdot Y$, donde $X, Y \in \mathbb{F}_p$, $p = 2^n + s$
Salida: $W \mod p$
  $\gamma \leftarrow W$
  mientras $\gamma > p$ **hacer**
    $\mu \leftarrow \gamma \text{ div } 2^n$
    $\gamma \leftarrow \gamma \mod 2^n - \mu s$
  fin mientras
  devolver

reducción de la figura 4.3 representaría dos iteraciones de este algoritmo. Sin embargo es importante que el valor de s se mantenga con un peso de Hamming bajo, pues conforme suba, el producto $\mu \cdot s$ se volverá más complejo. En la práctica se fija el valor de p, con lo cual se sabe exactamente el número de iteraciones que hay que ejecutar para reducir correctamente el número W.

4.2.6. Algoritmo de Multiplicación de Montgomery para Primos

En 1985 Peter Montgomery propuso un algoritmo de reducción que revolucionó la forma de realizar implementaciones eficientes en aritmética prima, el algoritmo es conocido como producto de Montgomery o reducción de Montgomery [30].

El algoritmo es sencillo, aunque requiere seguir una serie de transformaciones y realizar pasos de precómputo para poder obtener una verdadera eficiencia del método. El primer paso es encontrar fuera de línea los parámetros que cumplan con la ecuación:

$$\ell\ell^{-1} - p \cdot p' = 1$$

Donde p es el primo con el que se define el campo $\mathbb{F}_p$, y para que la ecuación anterior se cumpla hay que elegir un ℓ que cumpla con $gcd(p, \ell) = 1$, es decir, p y ℓ deben ser primos relativos. Para eficiencia del algoritmo generalmente se elige a $\ell = 2^n$ donde n es el número de bits de p.

El algoritmo de Montgomery se basa en trasladar los operandos a otro espacio donde se vuelve más sencillo realizar las operaciones, en este caso el espacio es conocido como "espacio de residuos de Montgomery" [12]. Teniendo dos operandos de entrada $a, b \in \mathbb{F}_p$,

la forma de llevarlos al espacio de residuos de Montgomery es:

$$\bar{a} = a \cdot \ell \mod p$$
$$\bar{b} = b \cdot \ell \mod p$$

Donde $\bar{a}$ y $\bar{b}$ son los elementos en el espacio de residuos de Montgomery. Una vez que los operandos están en este espacio, se utiliza el llamado producto de Montgomery que se puede ver en el algoritmo 4.9.

Algoritmo 4.9 Producto de Montgomery

Entrada: $\bar{a}$, $\bar{b}$,los parámetros ℓ, p', y el primo p
Salida: $\bar{c} = \bar{a} \cdot \bar{b}\ell^{-1} \mod p$
$\bar{c} \leftarrow \bar{a} \cdot \bar{b}$
$u \leftarrow \bar{c} \cdot p' \mod \ell$
$\bar{c} \leftarrow (\bar{c} + u \cdot p)/\ell$
devolver $\bar{c}$

Este algoritmo recibe los elementos $\bar{a}$ y $\bar{b}$, y regresa $\bar{c} = \bar{a} \cdot \bar{b} \cdot \ell^{-1} \mod p$. Lo anterior proviene de que este resultado en realidad es $c = a \cdot b \mod p$ en el espacio de Montgomery. Pues de lo anterior tendríamos que $\bar{c} = a \cdot b \cdot \ell \mod p$. Que en efecto es lo que nos da como resultado el algoritmo 4.9, pues:

$$\begin{aligned} \bar{a} \cdot \bar{b} \cdot \ell^{-1} \mod p &= a \cdot \ell \cdot b \cdot \ell \cdot \ell^{-1} \mod p \\ &= a \cdot b \cdot \ell \mod p \\ &= c \cdot \ell \mod p \\ &= \bar{c} \end{aligned}$$

Por lo que el resultado del producto de Mongomery se puede volver a usar para hacer más multiplicaciones dentro del mismo espacio de residuos de Mongomery. A simple vista un par de operaciones que podrían parecer complicadas en el algoritmo 4.9 son las operaciones de módulo ℓ y división por ℓ, pero como ℓ es una potencia de 2 estas operaciones sólo se limitan a eliminar los bits de la parte alta y la parte baja, respectivamente.

De esta manera es posible computar un producto modular utilizando 3 multiplicaciones, tomando en cuenta que la búsqueda de los parámetros de entrada y la transformación de los operandos se hacen fuera de línea, por lo que el algoritmo de Montgomery es muy

conveniente cuando se van a realizar una gran cantidad de multiplicaciones, con el mismo operando, o cuando es posible modificar los operandos sin salir del espacio de residuos de Mongomery, como en la exponenciación modular que es el ejemplo más común.

Es importante observar que el resultado $\bar{c}$ corresponde al producto aún en el espacio de residuos. Para obtener el resultado en $\mathbb{F}_p$ hay que aplicar la operacion de proyección inversa que es $c = \bar{c}\ell^{-1} \mod p$, que es como si se utilizara el algoritmo 4.9 con las entradas $\bar{c}$ y 1. También existen versiones de la reducción de Montgomery multiprecisión como se muestra en [12]. De estas versiones multipresición existen 2 variantes importantes, la primera es la llamada *SOS*(*Separated Operand Scanning*) vista en el algoritmo 4.10, el cual primero realiza la multiplicación multiprecisión, y posteriormente la operación de la reducción de Montgomery palabra por palabra.

La otra variante que existe, es la conocida como *CIOS*(*Coarsley Integrated Operand Scanning*), descrito en el algoritmo 4.11, en la cual, se realiza el producto y la reducción de forma combinada palabra por palabra, de forma que nunca se tiene un resultado de la multiplicación de $2n$ palabras, si no a lo más de $n + 1$ palabras que inmediatamente es reducido, este es uno de los algoritmos más populares para su implementación en software.

4.3. Multiplicación en $\mathbb{F}_{p^2}$

En [8], se decribe que para realizar un emparejamiento bilineal en curvas de Barreto-Naehrig[4], son necesarias una gran cantidad de operaciones en el campo $\mathbb{F}_{p^2}$, principalmente multiplicaciones y operaciones de elevar al cuadrado, lo que hace insdispensable el desarrollo de algoritmos eficientes para estas operaciones.

4.3.1. Algoritmo de Multiplicación

El campo $\mathbb{F}_{p^2}$, está compuesto de polinomios de primer grado con la forma $X(u) = x_0 + x_1 u$, donde $x_0, x_1 \in \mathbb{F}_p$, y usa cualquier polinomio, particularmente cualquier binomio de la forma $f(u) = u^2 + \beta$ para definir el campo, por lo que $u^2 = -\beta$, de esta manera se realiza una multiplicación eficiente utilizando el algoritmo de Karatsuba para reducir el número de multiplicaciones, en el campo $\mathbb{F}_p$, es decir:

$$\begin{aligned}(x_0 + x_1 u)(y_0 + y_1 u) &= x_0 \cdot y_0 + ((x_0 \cdot y_1 +)(x_1 \cdot y_0))u + x_1 \cdot y_1 \cdot u^2 \\ &= (x_0 \cdot y_0) - \beta(x_1 \cdot y_1) + ((x_0 + x_1)(y_0 + y_1) - x_0 \cdot y_0 - x_1 \cdot y_1)u\end{aligned}$$

Con base en las ecuaciones anteriores, el algoritmo 4.12, realiza la operación de Karatsuba, es importante notar que los productos no se reducen inmediatamente después de

Algoritmo 4.10 Producto de Montgomery SOS(Separated Operand Scanning)

Entrada: $\bar{a} = (a_{n-1}...a_0)_{2^W}$, $\bar{b} = (b_{n-1}...b_0)_{2^W}$, el primo $p = (p_{n-1}...p_0)_{2^W}$,y los parámetros $\ell = 2^{n \cdot W}$ y $p' = -p^{-1} \mod 2^w$

Salida: $\bar{a} \cdot \bar{b} \cdot \ell^{-1} \mod p$

1: $t = (t_{2n-1}...t_0)_{2^W} \leftarrow 0$
2: **para** i desde 0 hasta $n-1$ **hacer**
3: $C \leftarrow 0$
4: **para** j desde 0 hasta $n-1$ **hacer**
5: $(C, S) \leftarrow t_{i+j}$
6: $t_{i+j} \leftarrow S$
7: **fin para**
8: $t_{i+n} \leftarrow C$
9: **fin para**
10: **para** i desde 0 hasta $n-1$ **hacer**
11: $C \leftarrow 0$
12: $U \leftarrow (t_i \cdot p') \mod 2^W$
13: **para** j desde 0 hasta $n-1$ **hacer**
14: $(C, S) \leftarrow t_{i+j} + U \cdot p_i + C$
15: $t_{i+j} \leftarrow S$
16: **fin para**
17: t_{i+j}, C
18: **fin para**
19: **para** i desde 0 a n **hacer**
20: $t_i \leftarrow t_{i+n}$
21: **fin para**
22: **si** $t > p$ **entonces**
23: $t \leftarrow t - p$
24: **fin si**
25: **devolver** t

Algoritmo 4.11 Producto de Montgomery CIOS(Coarsley Integrated Operand Scanning)

Entrada: $\bar{a} = (a_{n-1}...a_0)_{2^W}$, $\bar{b} = (b_{n-1}...b_0)_{2^W}$, el primo $p = (p_{n-1}...p_0)_{2^W}$,y los parámetros $\ell = 2^{n \cdot W}$ y $p' = -p^{-1}$ mód 2^w

Salida: $\bar{a} \cdot \bar{b} \cdot \ell^{-1} \bmod p$

1: $t = (t_{2n-1}...t_0)_{2^W} \leftarrow 0$
2: **para** i desde 0 hasta $n-1$ **hacer**
3: $C \leftarrow 0$
4: **para** j desde 0 hasta $n-1$ **hacer**
5: $(C, S) \leftarrow t_j + a_j b_i + C$
6: $t_j \leftarrow S$
7: **fin para**
8: $(C, S) \leftarrow t_n + C$
9: $t_n \leftarrow S$
10: $t_{n+1} \leftarrow C$
11: $C \leftarrow 0$
12: $U \leftarrow (t_0 \cdot p') \bmod 2^W$
13: **para** j desde 0 hasta $n-1$ **hacer**
14: $(C, S) \leftarrow t_j + U \cdot p_j + C$
15: $t_j \leftarrow S$
16: **fin para**
17: $(C, S) \leftarrow t_n + C$
18: $t_n \leftarrow S$
19: $t_{n+1} \leftarrow t_{n+1} + C$
20: **para** j desde 0 a n **hacer**
21: $t_j \leftarrow t_{j+n}$
22: **fin para**
23: **fin para**
24: **si** $t > p$ **entonces**
25: $t \leftarrow t - p$
26: **fin si**
27: **devolver** t

que fueron calculados, si no hasta el final, cuando ya se convertirán en los coeficiantes w_0 y w_1 del resultado final.

Algoritmo 4.12 Algoritmo de multiplicación en $\mathbb{F}_{p^2}$ presentado en [8]

Entrada: $X, Y \in \mathbb{F}_{p^2}$, donde $X = x_0 + x_1 u$ y $Y = y_0 + y_1 u$, y $u^2 = -5$
Salida: $W = X \cdot Y \in \mathbb{F}_{p^2}$

$s \leftarrow x_0 + x_1$
$t \leftarrow y_0 + y_0$
$d_0 \leftarrow s \cdot t$
$d_1 \leftarrow x_0 \cdot y_0$
$d_2 \leftarrow x_1 \cdot y_1$
$d_0 \leftarrow d_0 - d_1 - d_2$
$w_1 \leftarrow d_0 \bmod p$
$d2 \leftarrow 5d_2$
$d1 \leftarrow d1 - d2$
$w_0 \leftarrow d1 \bmod p$
devolver $W = w_0 + w_1 u$

Además de la multiplicación, también es conveniente definir un algoritmo para elevar al cuadrado, en [8] se menciona que elegir a $\beta = 5$ permite a que las características del polinomio irreducible $u^2 + 5$ puede brindar ventajas para reducir la cantidad de multiplicaciones necesarias en el campo $\mathbb{F}_p$, como a continuación se puede ver, al desarrollar las ecuaciones de este producto:

$$\begin{aligned}(x_0 + x_1 u)(x_0 + x_1 u) &= x_0^2 + 2x_0 \cdot x_1 + x_1^2 u^2 \\ &= x_0^2 - 5x1^2 + 2x_0 \cdot x_1 \cdot u \\ &= (x_0 - x_1)(5x_1 + x_0) - 4x_0 \cdot x_1 + 2x_0 \cdot x_1 \cdot u\end{aligned}$$

Al completar la diferencia de cuadrados, es posible reducir la cantidad de multiplicaciones en el campo $\mathbb{F}_p$ necesarias para completar un producto en $\mathbb{F}_{p^2}$ de 4 a 2, es lo que hace más eficiente elevar un número al cuadrado que realizar una multiplicación, en base las ecuaciones anteriores se realiza el algortimo 4.13.

4.3.2. *Reducción Displicente*

La "*Reducción Displicente*", es un concepto que proviene desde la suma de fracciones, en donde se realiza la reducción hasta que la misma sea conveniente, por ejemplo si se

Algoritmo 4.13 Algoritmo para elevar al cuadrado en $\mathbb{F}_{p^2}$ presentado en [8]

Entrada: $X \in \mathbb{F}_{p^k}$, donde $X = x_0 + x_1 u$, y $u^2 = -5$
Salida: $W = X^2 in \mathbb{F}_{p^2}$
$t \leftarrow x_1 + x_1$
$d_1 \leftarrow t \cdot x_1$
$t \leftarrow x_0 - x1 + p$
$w_1 \leftarrow x_0 + 5x_1$
$d_0 \leftarrow t \cdot w_1$
$w_1 \leftarrow d_1 \bmod p$
$d_1 \leftarrow d_1 + d_1$
$d_0 \leftarrow d_0 - d_1$
$w_0 \leftarrow d_0 \bmod p$
devolver $W = w_0 + w_1 u$

tiene la suma:

$$\frac{7}{6} - \frac{3}{6} + \frac{2}{6}$$

Es más conveniente realizar las sumas primero, y al final reducir:

$$\frac{7 - 3 + 6}{6}$$

Lo mismo se puede aplicar a las reducciones modulares:

16 mod 23+21 mod 23+4 mod 23 = 16+21+4 mod 23 = 18 mod 23 = 48 mod 23 = 18 mod 23

En esto se basan los algoritmos 4.12 y 4.13 para realizar los productos con sólo dos reducciones. La reducción displicente(también conocida como *Lazy Reduction* en inglés), tiene una eficiencia demostrada en [2] para el cálculo de emparejamientos bilineales, pues su uso permite reducir drásticamente la cantidad de operaciones necesarias. Otro ejemplo de ello es la implementación hecha en hardware mostrada en [46], que utiliza una combinación de la reducción displicente en conjunto con el teorema del residuo chino, para realizar la reducción modular de manera más eficiente.

4.4. Sumario

A lo largo de este capítulo se exploraron distintos algoritmos necesarios para realizar operaciones aritméticas con números grandes, destinados a ser usados principalmente

en aplicaciones criptográficas. Los algoritmos que se exploran son los de suma, resta, multiplicación y reducción modular, haciendo énfasis en los últimos dos, pues el producto modular (multiplicación con reducción) es una operación clave en la criptografía de curvas elípticas, además se explora detalladamente el algoritmo de multiplicación de Montgomery, el cual por su eficiencia es parte central de este trabajo, como se verá en los siguientes capítulos. Para finalizar se dio una muy breve introducción de la multiplicación en el campo $\mathbb{F}_{p^2}$, operación importante para el cálculo de emparejamientos bilineales en curvas definidas sobre campos finitos primos.

Capítulo 5

Multiplicador en $\mathbb{F}_p$

A continuación describiremos el primer diseño del presente trabajo, que es un multiplicador en $\mathbb{F}_p$ que realiza producto modular de números de 256 bits, en el campo descrito para las curvas ordinarias de Barreto-Naerigh[4]. Hay que señalar que este diseño se basa en ideas mostradas por J. Fan en [19].

5.1. Algoritmos para multiplicación en $\mathbb{F}_p$

Antes de describir la arquitectura se hará mención de las características algoritmicas necesarias para implementarla. Al realizar el producto modular que es $a \cdot b \bmod p$ el orden lógico de ideas es la realización primero de la multiplicación y luego la realización de la reducción. La arquitecura descrita en este capítulo trabaja con números de 256 bits convertidos en polinomios de 5 elementos, por lo que se requiere primero un multiplicador eficiente de polinomios de grado 4. Para ello se utilizo un multiplicador basado en Karatsuba que se describe en la siguiente sección, para después explicar la reducción polinomial utilizada y algunas caracteríticas interesantes para la implementación.

5.1.1. Multiplicador de Karatsuba

Para el algoritmo de Karatsuba se consideró la variante para polinomios de cuarto grado propuesta por P. L. Mongomery[32], en el cual se realiza el producto de polinomios de 5 coeficientes cada uno con únicamente 13 productos, en vez de los 25 que se necesitan en la multiplicación tradicional, sin embargo, este algoritmo está diseñado para su implementación en software, que implica muchas sumas y variables temporales, lo cual no es conveniente para la implementación en hardware, por lo que se diseñó una variante propia

de Karatsuba con polinomios que resultó más sencilla de implementar, con menos sumas, pero a un costo de un producto más, por lo que se necesitan 14 productos para realizar la multiplicación, en el algoritmo 5.1 se describe nuestra variante, y en el cuadro 5.1 una comparativa entre algoritmos, mustrando que el desarrollado aquí es el más equilibrado.

La idea con la que se diseñó este algoritmo es que siempre se esté realizando al menos una multiplicación nueva entre coeficiente y coeficiente del polinomio de salida final, con el propósito de que en una arquitectura implementada en hardware, el multiplicador siempre esté ocupado, y de esta manera se puedan estar procesando nuevos coeficientes al tiempo que los anteriores ya se dieron como salidas, y que se tenga un número pequeño de variables temporales para ahorrar espacio de almacenamiento y tiempo en el acceso a los valores.

Cuadro 5.1: Comparación entre algoritmos de multiplicación de polinomios de 5 coeficientes

Algortitmo	Multiplicaciones	Sumas	Control
Libro de Texto	25	20	Simple
Karatsuba, este trabajo	14	40	Complejo al realizar sumas
Karatsuba [32]	13	73	Muy complejo, Muchos temporales

5.1.2. Multiplicador 64×64

La multiplicación de dos números de 256 bits $c = \bar{a} \cdot \bar{b}$, se ha convertido en una multiplicación de 2 polinomios, de 5 elementos, $c(t) = a(t)b(t)$ donde cada elemento tiene a lo más 64 bits. Por la razón anterior, un bloque básico importante en el diseño de la arquitectura propuesta aquí es un multiplicador de 64×64 eficiente.

El dispositivo con el que se trabajó es un *FPGA* **Virtex 6**, lo que significa que podemos aprovechar los componentes *DSP48slices*, descritos en la sección 2.1.3 del capítulo 2 para construir el multiplicador.

De esta manera usando como base multiplicadores de 24×17, se construye un multiplicador de 64×64, que a su vez sirve como base para un multiplicador de Karatsuba para elementos de 256×256, como se esquematiza en la figura **??**.

Los *DSP48slices* son componentes para procesamiento de señales, que entre otras cosas, incluyen multiplicadores asimétricos de 25×18 bits. Es posible utilizarlos como multiplicadores simétricos de 18×18 bits, pero esto resulta ineficiente, pues se desperdician los bits más significativos de un operando, y conlleva un manejo extra que hay que hacer sobre los bits del resultado.

Una alternativa frente a los inconvenientes anteriores, se presenta en el trabajo propuesto en [38] en el cual usan los *DSP48slices* asimétricos de un *FPGA* para poder realizar multi-

Algoritmo 5.1 Algoritmo de Karatsuba de 5 Términos

Entrada: $a(t) = \sum_{i=0}^{4} a_i t^i$, $b(t) = \sum_{i=0}^{4} b_i t^i$
Salida: $c(t) = a(t)b(t)$

1: $c(t)(= \sum_{i=0}^{8} c_i t^i) \leftarrow 0$;
2: Fase de Productos
3: $p_0 = a_0 b_0$;
4: $p_1 = a_1 b_1$;
5: $p_2 = (a_0 + a_1)(b_0 + b_1)$;
6: $p_3 = a_2 b_2$;
7: $p_4 = (a_0 + a_2)(b_0 + b_2)$;
8: $p_5 = a_3 b_3$;
9: $p_6 = (a_2 + a_3)(b_2 + b_3)$;
10: $p_7 = (a_1 + a_3)(b_1 + b_3)$;
11: $p_8 = (a_0 + a_1 + a_2 + a_3)(b_0 + b_1 + b_2 + b_3)$;
12: $p_9 = a_4 b_4$;
13: $p_{10} = (a_0 + a_4)(b_0 + b_4)$;
14: $p_{11} = (a_0 + a_1 + a_4)(b_0 + b_1 + b_4)$;
15: $p_{12} = (a_2 + a_4)(b_2 + b_4)$;
16: $p_{13} = (a_2 + a_3 + a_4)(b_2 + b_3 + b_4)$
17: Fase de Sumas
18: $c_0 = p_0$
19: $c_1 = p_2 - p_1 - p_0$
20: $c_2 = p_4 + p_1 - p_0 - p_3$
21: $S1 = p_6 - p_5 - p_3$
22: $c_3 = p_8 - p_7 - p4 - c_1 - S1$
23: $c_4 = p_{10} - p_9 - p_0 + p_3 + p_5 - p_1 + p_7$
24: $c_5 = p_{11} - p_1 - p_{10} - c_1 + S1$
25: $c_6 = p_{12} - p_9 + p_5 - p_3$
26: $c_7 = p_{13} - p_5 - p_{12} - S1$
27: $c_8 = p_9$
28: **devolver** $c(t)$

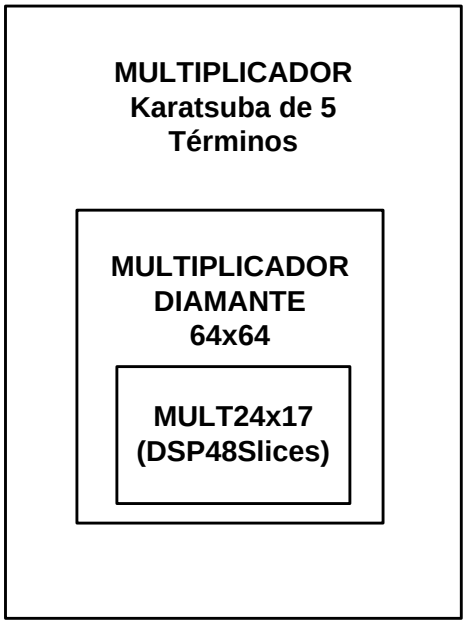

Figura 5.1: Jerarquía de los distintos multiplicadores usados para el diseño

plicadores grandes. La principal idea tras esta propuesta es la forma en la que se acomodan los subproductos.

Para realizar la multiplicación de los elementos de 64 bits, utilizando los multiplicadores asimétricos los operandos deben de ser divididos como se muestra en la figura 5.2.

Si buscamos realizar un producto usando el método tradicional o de libro de texto con los operandos divididos de manera asimétrica, lo que obtenemos es lo que se muestra en la Figura 5.3.

En donde las palabras $A0, A1, A2$ y $B0, B1, B2, B4$, son los dígitos de los operandos de entrada, al dividirlos de manera asimétrica se puede apreciar que se requieren 12 multiplicadores y 11 sumas con acarreo para obtener el resultado. Pero los subproductos pueden reacomodarse como se ilustra en la Figura 5.4.

De forma que obtenemos el mismo resultado únicamente realizando 6 sumas, debido a que las otras adiciones fueron reemplazadas por la concatenación de los resultados. La ventaja de este método es su efectividad para multiplicadores asimétricos. Así, teniendo operadores asimétricos de 24 y 17 bits, pues aunque los *DSPslices* son de 25×18 necesitamos tomar un bit de cada operando para el signo.

El multiplicador se implementó como se puede apreciar en la figura 5.5, utilizando un *DSP48Slice* por subproducto, aprovechando los registros internos a nuestra dispocisión y

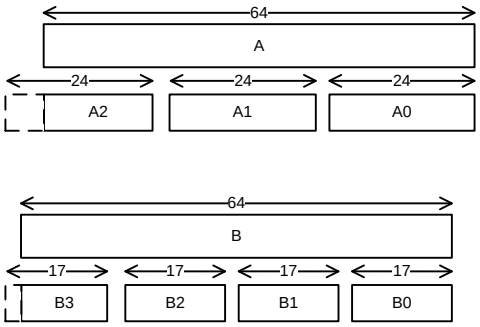

Figura 5.2: División asimétrica de los operandos

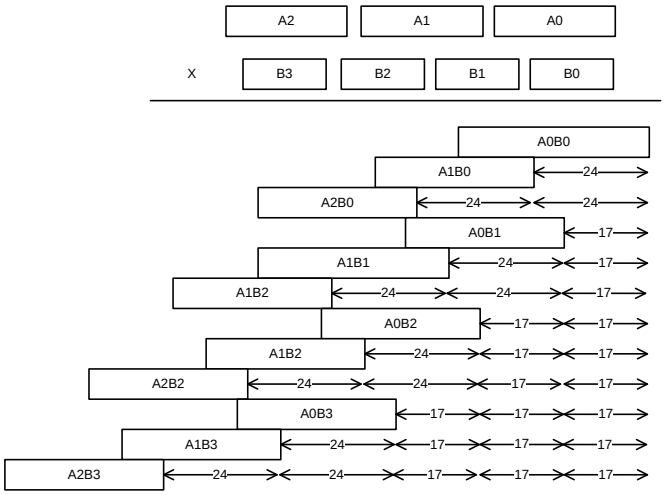

Figura 5.3: Método de tradicional de multiplicación o de libro de texto

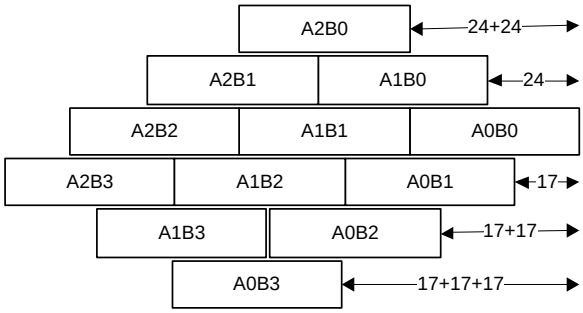

Figura 5.4: Configuración de los subproductos en el diseño del multiplicador 64×64

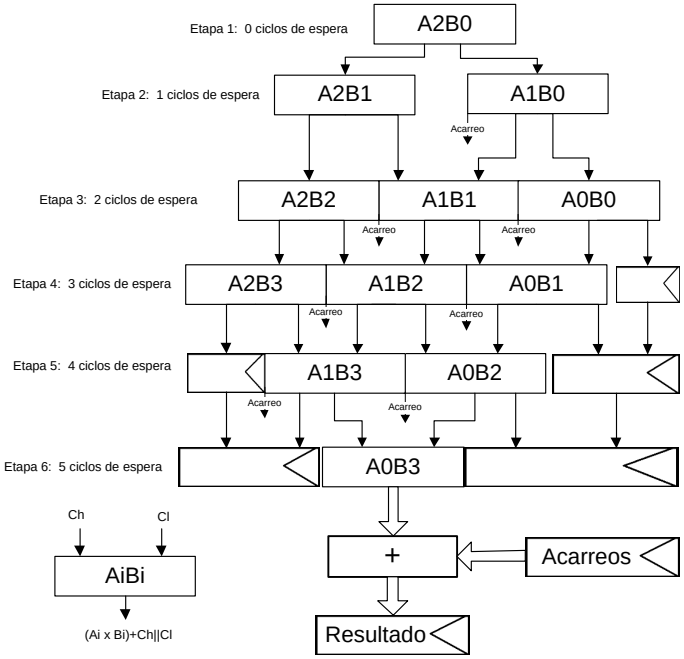

Figura 5.5: Implementación del multiplicador de 64×64

colocando los componentes en etapas. De esta manera la primera etapa no espera ningún ciclo para emitir el resultado, la segunda etapa espera 1 ciclo, la segunda 2, y así hasta la sexta etapa que la sexta etapa espera 5 ciclos, de forma que se pueden utilizar los sumadores internos cuando sea necesario en cada etapa y después de 6 ciclos de reloj, se obtiene el resultado de la multiplicación. Gracias a los registros de *Pipe-Line* internos de los *DSP48Slices* cada ciclo de reloj se emite un nuevo resultado una vez que la tubería esta llena.

Como paso final se deben de sumar acarreos generados en las adiciones intermedias del proceso, pero únicamente es una suma de 6 bits, por lo que en su mayoría este diseño se sustenta en el uso completo de los dipositivos *DSP48slices* obteniendo así una frecuencia de trabajo de 310Mhz, otra ventaja es que esta parte del diseño no necesita control interno, pues sólo hay que llenar la *tubería* y se obtienen los productos de manera automática, pues los componentes empotrados hacen el resto.

La forma en que se progrmaron los componentes *DSP48Slices* fue usando la herramienta *IPCores* inconrporada en el ambiende de desarrollo **ISE** de **Xilinx**. Cuando se crea un nuevo componente basandose en un *IPCore*, se debe seleccionar el grupo de macros y ahí se encontrara la configuración de los *DSP48Slices* como se muestra en la figura 5.6.

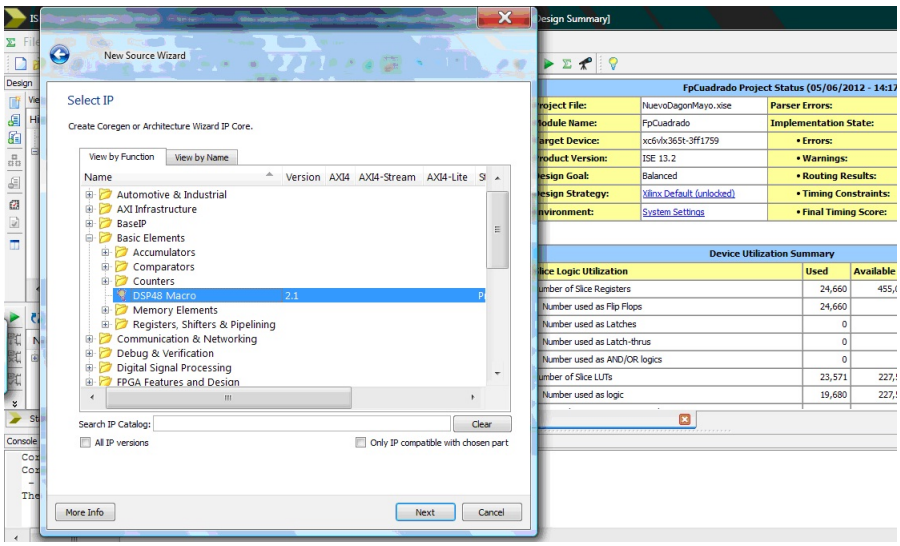

Figura 5.6: Pantalla de selección de componente a generar con la herramienta *IPCores*

Una vez seleccionada la opción se nos desplegarán los distintos cuadros de configuración, el primero mostrado en la figura 5.7, dónde se selecciona el modo en el que trabajará el componente *DSP48Slice*, que puede ser multiplicador, sumador, sumador multiplicador etc.

Para finalizar se nos muestra la pantalla de la figura 5.8, dónde se indican los registros disponibles dentro del componente *DSP48Slices*, y es posible habilitarlos, ya sea individualmente o por etapas, teniendo un máximo de 6 etapas.

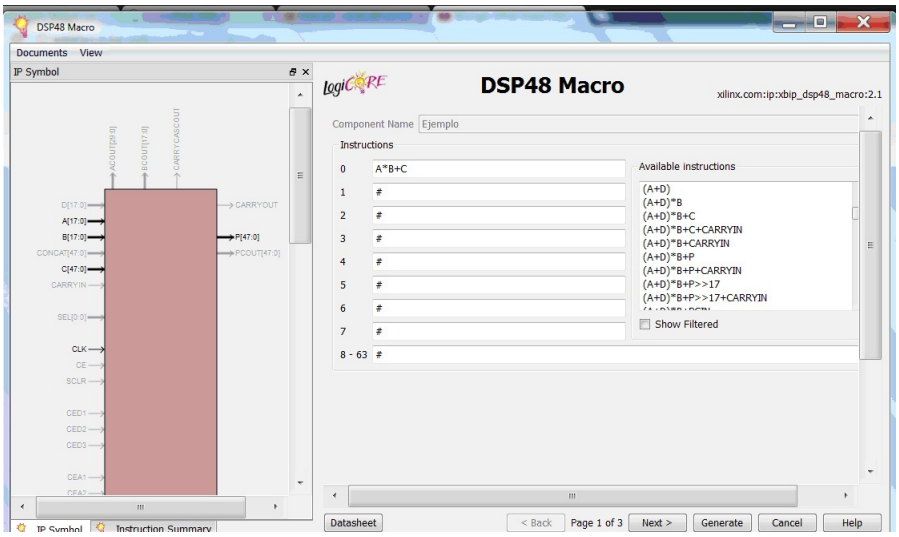

Figura 5.7: Pantalla de selección de operación del componente *DSP48Slice*

Figura 5.8: Pantalla de configuración de los registros internos de los componentes *DSP48Slice*

5.1.3. Algoritmo de Multiplicación de Montgomery

El algoritmo de multiplicación de Montgomery, es muy usado en aplicaciones criptográficas debido a su capacidad de realizar el producto sobre campos $\mathbb{F}_p$ requiriendo unicamente el cálculo de 3 multiplicaciones y algunas operaciones fuera de línea.

Dentro de las operaciones que se realizan fuera de línea está encontrar los parámetros, p', ℓ y ℓ^{-1}, tales que cumplan con la siguiente condición:

$$\ell \cdot \ell^{-1} - p \cdot p' = 1$$

Dónde p es el primo que define el campo $\mathbb{F}_p$ sobre el cual se van a realizar las operaciones y ℓ es un parámetro que debe ser primo relativo a p, es decir que cumple con la condición de que $gcd(p, \ell) = 1$.

Una vez que se ha elegido el parámetro ℓ, si se quiere realizar el producto de $a \cdot b \bmod p$, primero hay que llevar los operandos al *Dominio de Montgomery*, de la siguiente manera:

$$\bar{a} = a \cdot \ell \bmod p$$

Hasta aquí quedaron las operaciones que se hacen fuera de línea, o que se pueden considerar mínimas cuando se va a trabajar con una gran cantidad de multiplicaciones; como es en el caso de exponenciaciones, usadas para algoritmos de clave pública como RSA[33] o ElGamal[18].

Haciendo uso de los cáculos anteriores, el *Algoritmo de Montgomery para Multiplicación Modular* es el descrito en el algoritmo 5.2.

Algoritmo 5.2 Producto de Montgomery

Entrada: Los operandos $\bar{a}$, $\bar{b}$; los parámetros ℓ, p', y el primo p
Salida: $\bar{c} = \bar{a} \cdot \bar{b} \cdot \ell^{-1} \bmod p$
$\bar{c} \leftarrow \bar{a} \cdot \bar{b}$
$u \leftarrow \bar{c} \cdot p' \bmod \ell$
$\bar{c} \leftarrow (\bar{c} + u \cdot p)/\ell$
devolver $\bar{c}$

El algoritmo 5.2, da como resultado $\bar{c} = \bar{a} \cdot \bar{b} \cdot \ell^{-1} \bmod p$, que es $c = a \cdot b \bmod p$ en el *Dominio de Montgomery*, por lo que se puede usar este resultado como un nuevo valor de entrada en el producto de Montgomery sin necesidad de trasformación previa, es por ello que la conversión de los primeros dos operandos a y b es considerado precómputo. Además el algoritmo 5.2 es particularmente eficiente cuando se utiliza a $\ell = 2^n$, donde n es le número de bits del primo p. La eficiencia radica en que la forma de ℓ convierte las

operaciones módulo ℓ y de división por ℓ en mantener sólo los n primeros bits o retirarlos respectivamente. De está manera se consigue calcular el producto modular (en el *Dominio de Mongomery*) únicamente con los siguientes 3 productos:

1. $\bar{a} \cdot \bar{b}$
2. $\bar{c} \cdot p'$
3. $u \cdot p$

Siendo muy eficiente si se compara con otras reducciones, pero existen distintos métodos para optimizarlo, más adelante se mostrarán algunos ejemplos de dichas mejoras.

5.1.4. Propuesta de multiplicador Polinomial

En [19] se presenta un método de multiplicación modular, cuya aplicación está orientada a emparejamientos bilineales. Dicho método hace uso de las caracterísiticas de una famlia especial de curvas elípticas, llamadas *Curvas de Barreto-Naehrig*; que son curvas definidas sobre un campo $\mathbb{F}_p$, y del algoritmo de multiplicación polinomial presentado en [13].

La idea principal de este multiplicador es usar una representación polinomial de los operandos en el *Dominio de Montgomery*, para ahorrar operaciones que a continuación serán descritas a detalle.

Para construir una curva de Barreto-Naehrig que está definida sobre un campo primo $\mathbb{F}_p$, el primo p se debe de obtener de la evaluación del siguiente polinomio:

$$p(t) = 36t^4 + 36t^3 + 24t^2 + 6t + 1 \tag{5.1}$$

Donde $t \in \mathbb{Z}^+$, es cualquier valor que al evaluar $p(t)$ da como resultado un número primo p. Gracias a esta construcción el primo p tiene algunas características interesantes, las cuales se indican a continuación:

1. $p \bmod t = 1$
2. $p^{-1} \bmod t = 1$
3. t al ser menor que p cumple con $gcd(t, p) = 1$
4. El polinomio $p(t)$ posee coeficientes pequeños.

Además p esta construido a partir de un polinomio con base t, todos los elementos $a \in \mathbb{F}_p$ pueden ser representados como polinomios de base t, es decir:

$$a = a(t) = a_4t^4 + a_3t^3 + a_2t^2 + a_1t + a_0$$

Donde $a_3, a_2, a_1, a_0 < t$ y $a_4 < 36$, lo anterior debido a la estructura del polinomio $p(t)$. En este trabajo se usan parámetros de t de aproximadamente 64 bits, por lo que a en general tiene 256 bits, a_3, a_2, a_1 y a_0 tienen 64 bits y a_4 tiene a lo más 6 bits (el número de bits necesarios para representar 36).

Retomando el algoritmo de multiplicación de Mongomery, se necesita un parámetro ℓ para realizar las operaciones. Por motivos de eficiencia se elije ℓ como una potencia de 2, pero en realidad el procedimiento funciona con cualquier número ℓ que cumpla con la condición de $gcd(\ell, p) = 1$. De esta manera podemos elegir a $\ell = t$, para utilizar el algoritmo de Montgomery de una forma más eficiente.

Uniendo las dos ideas de representación polinomial y de elegir a $\ell = t$ se puede construir un *Dominio Polinomial de Montgomery*, donde llevaríamos a los operandos de entrada a ese dominio mediante el algoritmo 5.3. Se puede observar en dicho algoritmo que cuando se realiza el cambio al *Dominio Polinomial de Mongomery*, en vez de usar t se usa t^4, debido a que t es un parámetro muy pequeño y utilizar una potencia del mismo es equivalente.

Algoritmo 5.3 Algoritmo para conversión al *Dominio Polinomial de Mongomery*

Entrada: t, $p = 36t^4 + 36t^3 + 24t^2 + 6t + 1$,$a \in \mathbb{F}_p$
Salida: $a(t)t^4 \bmod p$

1: $\bar{a} \leftarrow at^4 \bmod p$
2: $a(t) = \sum_{i=0}^{4} a_i t^i \leftarrow 0$
3: **para** $i = 0$ to 4 **hacer**
4: $\quad a_i \leftarrow \bar{a} \bmod t$
5: $\quad \bar{a} \leftarrow \bar{a}$ div t
6: **fin para**
7: **devolver** $a(t)$

Con la elección del parámetro $\ell = t$ y el uso de la forma polinomial de los operandos se consiguen las siguientes ventajas sobre las multiplicaciones:

1. La operación $\bar{a} \cdot \bar{b}$ que es una multiplicación entre números de 256 bits se intercambia por el producto de polinomios $a(t) \cdot b(t)$, coeficientes de 64 bits, por lo que ahora se necesita un multiplicador de 64 bits en vez de uno de 256 bits que es menos manejable.

2. En la producto de $\bar{c} \cdot p'$, ahora $p' = -p^{-1} \bmod t = -1$, por lo que la operación ahora es $c(t) \cdot p'(t) = -c(t)$. Es decir, esta multiplicación ya no se realiza.

3. Realizar el producto de $u \cdot p$, se convierte en calcular $u(t) \cdot p(t)$ donde $p(t)$ es un polinomio con coeficientes pequeños, por lo que esta multiplicación se facilita mucho, al grado de que es posible realizarla con unas cuantas sumas en vez de necesitar un multiplicador.

Tomando estas ventajas y basandose en el método descrito en [13]; en [19] se presenta el algoritmo 5.4.

Algoritmo 5.4 Algoritmo de multiplicación modular propuesto en [19]

Entrada: $at^4 \bmod p = a(t) = \sum_{i=0}^{4} a_i t^i$,
$bt^4 \bmod p = b(t) = \sum_{i=0}^{4} b_i t^i$,
$p(t) = 36t^4 + 36t^3 + 24t^2 + 6t + 1$.
Salida: $a(t)b(t)t^4 \bmod p$
1: $c(t)(= \sum_{i=0}^{4} c_i t^i) \leftarrow 0$
2: **para** $i = 0$ to 4 **hacer**
3: $c(t) \leftarrow c(t) + a(t)b_i$
4: $\mu \leftarrow c_0 \ \text{div} \ t$; $\gamma \leftarrow c_0 \bmod t$
5: $g(t) \leftarrow p(t)(-\gamma)$
6: $c(t) \leftarrow (c(t) + g(t))/t + \mu$
7: **fin para**
8: **para** $i = 0$ to 3 **hacer**
9: $c_{i+1} \leftarrow c_{i+1} + (c_i \ \text{div} \ t)$; $c_i \leftarrow c_i \bmod t$
10: **fin para**
11: **devolver** $c(t)$

Este algoritmo realiza la reducción del polinomio $c(t)$ coeficiente a coeficiente. En este capítulo es presentado tal y como aparece en [19], por lo que algunas operaciones difieren con respecto al producto de Montgomery original. La primera de ellas es la forma en la que realiza la reducción módulo t. En el producto de Montgomery convencional, se realiza en una sóla operación de módulo, mientras que aquí se realiza coeficiente a coeficiente del polinomio.

Además, es necesario realizar una división por t, debido a que los polinomios con los que se está trabajando son en realidad números en base t, por lo que cuando el valor de un coeficiente a_i supere el valor de t se ha generado un acarreo que debe de ser sumado con el siguiente orden de magnitud.

Precisamente en estas operaciones de módulo t y división por t es donde podría darse una duda con respecto a la eficiencia de este algoritmo. Una forma para resolver a esta situación es haciendo una selección especial de t para que las operaciones anteriores sean más fáciles de realizar que en una división convencional. La forma especial que se busca en t es:

$$t = 2^n + s \tag{5.2}$$

Donde $n \in \mathbb{Z}^+$ y $s \in \mathbb{Z}$. De esta manera la reducción de los coeficientes c_i de $c(t)$:

$$\mu \leftarrow c_i \; div \; t; \gamma \leftarrow c_i \; mod \; t$$

pueden ser cambiadas por utilizar de forma iterativa la reducción rápida:

$$\mu \leftarrow c_i \; div \; 2^n; \gamma \leftarrow c_i \; mod \; 2^n - \mu s$$

Donde si s cumple con la condición de que $s < t^{n/2}$, y tiene un peso Hamming bajo, la reducción sólo necesita 2 iteraciones de las operaciones anteriores para completarse. Las operaciones de división y módulo por una potencia de 2 son muy eficientes, pues sólo hay que hacer desplazamientos a la izquierda en la división, y mantener los n primeros bits en la operación de módulo 2^n.

Por ejemplo, el valor de t que se propone usar en [19] es :

$$t = 2^{63} + 2^9 + 2^8 + 2^6 + 2^5 + 2^3 + 1 \text{ donde } n = 63 \text{ y } s = 2^9 + 2^8 + 2^6 + 2^5 + 2^3 + 1$$

Entre más bajo sea el peso de Hamming del parámetro t más eficiente es la implementación, es por esta razón que en este trabajo se elige:

$$t = 2^{61} - 2^{15} + 1 \text{ donde } n = 61 \text{ y } s = -2^{15} + 1$$

En el algoritmo 5.4, el producto $c(t) = a(t)b(t)$ se realiza utilizando el algoritmo tradicional de multiplicación o de libro de texto, que es de orden cuadrático.

Una mejora natural es cambiar este producto por alguna variante del algoritmo de multiplicación Karatsuba, con lo cual el algoritmo final a implementar será el Algoritmo5.5, con todas las mejoras antes mencionadas.

En resumen, el algoritmo 5.5 incorpora todas las mejoras antes mencionadas para dar como resultado una implementación eficiente. Primero recibe el polinomio $p(t)$ que tiene coeficientes pequeños, y $t = 2^n + s$ donde n y s cumplen las restricciones ya mencionadas anteriormente. A continuación en la linea 2 se realiza la multiplicación entre los polinomios $a(t)$ y $b(t)$, utilizando un multiplicador a la Karatsuba que se describe más adelante en el algoritmo 5.1. Como resultado de este se tiene el polinomio $c(t)$ el cual posee 9 coeficientes,

Algoritmo 5.5 Propuesta de Producto Modular

Entrada: $at^4 \bmod p = a(t) = \sum_{i=0}^{4} a_i t^i$,
$bt^4 \bmod p = b(t) = \sum_{i=0}^{4} b_i t^i$,
$p(t) = 36t^4 + 36t^3 + 24t^2 + 6t + 1$, $t = 2^n + s$
Salida: $a(t)b(t)t^4 \bmod p$

1: $c(t) \leftarrow 0$
2: $c(t) = Karatsuba5terminos(a(t), b(t))$
3: **para** $i = 0$ to 4 **hacer**
4: $\mu \leftarrow c_0$ div 2^n; $\gamma \leftarrow c_0 \bmod 2^n - \mu s$
5: $g(t) \leftarrow p(t)(-\gamma)$
6: $c(t) \leftarrow (c(t) + g(t))/t + \mu$
7: **fin para**
8: **para** $i = 0$ to 3 **hacer**
9: $\mu \leftarrow c_i$ div 2^n; $\gamma \leftarrow c_i \bmod 2^n - \mu s$
10: $c_{i+1} \leftarrow c_{i+1} + \mu$; $c_i \leftarrow \gamma$
11: **fin para**
12: **devolver** $\sum_{i=0}^{4} c_i t^i$

y hay que reducirlo a 5, para ello se utiliza el ciclo de las líneas 3 a 7, en dónde se utiliza la reducción rápida para obtener μ y γ. La multiplicación de la línea 5 es muy eficiente, debido a que los coeficientes de $p(t)$ son muy pequeños, lo que conlleva a usar sumadores en vez de otro multiplicador en esta etapa. En la línea 6 se puede observar una división entre t que en realidad significa descartar el coeficiente menos significativo, pues como estamos usando las ecuaciones de Montgomery, este debe de ser 0. De esta manera de este ciclo obtenemos un polinomio de 4 términos. Como paso final, se realiza el ciclo descrito en las líneas 8 a 10, que es aplicar la reducción rápida sobre todos los coeficientes del polinomio, para garantizar que estos sean menores a t. Este procedimiento puede generar un término extra, por lo que el resultado final es el polinomio $c(t)$ con 5 elementos.

5.1.5. Caracterísiticas de la Reducción Polinomial

La reducción polinomial consiste en reducir el polinomio de resultado de la multiplicación de 9 coeficientes a sólo 4. En esta subsección se detalla el hecho de que se puede reducir el polinomio, sin la necesidad de que este último esté completamente calculado.

Para demostrar lo anterior partimos del polinomio $c(t)$ que tiene la forma:

$$c(t) = c_8 t^8 + c_7 t^7 + c_6 t^6 + c_5 t^5 + c_4 t^4 + c_3 t^3 + c_2 t^2 + c_1 t + c_0$$

EL valor de t por el momento sólo determina la posición de los coeficientes c_i, por lo que podemos ver a $c(t)$ desde este punto como la lista de coeficientes C. De esta manera si se aplica la reducción que comprende la primera iteración de las líneas 4, 5 y 6 del del algoritmo 5.5, se tiene que:

$C^{(0)}$	$= C_8$	C_7	C_6	C_5	C_4	C_3	C_2	C_1	$\cancel{C_0}$
								μ_0	
					$36\gamma_0$	$36\gamma_0$	$24\gamma_0$	$6\gamma_0$	
$C^{(1)}$	$= C'_8$	C'_7	C'_6	C'_5	C'_4	C'_3	C'_2	C'_1	

Donde $\mu = C_0 \text{ div } 2^n$ y $\gamma_0 = C_0 \bmod 2^n - \mu \cdot s$, recordando que $n = 61$ y $s = -2^{15} + 1$. Además $C'_1 = C_1 + \mu_0 - 6\gamma_0$. Aplicando iterativamente este método obtenemos el total de la reducción:

$C^{(0)}$	$= C_8$	C_7	C_6	C_5	C_4	C_3	C_2	C_1	$\cancel{C_0}$
								μ_0	
					$-36\gamma_0$	$-36\gamma_0$	$-24\gamma_0$	$-6\gamma_0$	
$C^{(1)}$	$= C'_8$	C'_7	C'_6	C'_5	C'_4	C'_3	C'_2	$\cancel{C'_1}$	
							μ_1		
				$-36\gamma_1$	$-36\gamma_1$	$-24\gamma_1$	$-6\gamma_1$		
$C^{(2)}$	$= C''_8$	C''_7	C''_6	C''_5	C''_4	C''_3	$\cancel{C''_2}$		
						μ_2			
			$-36\gamma_2$	$-36\gamma_2$	$-24\gamma_2$	$-6\gamma_2$			
$C^{(3)}$	$= C'''_8$	C'''_7	C'''_6	C'''_5	C'''_4	$\cancel{C'''_3}$			
					μ_3				
		$-36\gamma_3$	$-36\gamma_3$	$-24\gamma_3$	$-6\gamma_3$				
$C^{(4)}$	$= C^{iv}_8$	C^{iv}_7	C^{iv}_6	C^{iv}_5	$\cancel{C^{iv}_4}$				
				μ_3					
	$-36\gamma_4$	$-36\gamma_4$	$-24\gamma_4$	$-6\gamma_4$					
$C^{(5)}$	$= C^{v}_8$	C^{v}_7	C^{v}_6	C^{v}_5					

El proceso anterior se puede resumir en las siguientes ecuaciones:

$$
\begin{aligned}
C_5^v &= C_5 + \mu_4 - 36\gamma_1 - 36\gamma_2 - 24\gamma_3 - 6\gamma_4 \\
C_6^v &= C_6 - 36\gamma_2 - 36\gamma_3 - 24\gamma_4 \\
C_7^v &= C_7 - 36\gamma_3 - 36\gamma_4 \\
C_8^v &= C_8 - 36\gamma_4
\end{aligned}
$$

Donde:

$$
\begin{aligned}
\mu_0 &= C_0 \text{ div } 2^n; & \gamma_0 &= C_0 \bmod 2^n - \mu_0 \cdot s \\
\mu_1 &= C_1' \text{ div } 2^n; & \gamma_1 &= C_1' \bmod 2^n - \mu_0 \cdot s \\
\mu_2 &= C_2'' \text{ div } 2^n; & \gamma_2 &= C_2'' \bmod 2^n - \mu_0 \cdot s \\
\mu_3 &= C_3''' \text{ div } 2^n; & \gamma_3 &= C_3''' \bmod 2^n - \mu_0 \cdot s \\
\mu_4 &= C_4^{iv} \text{ div } 2^n; & \gamma_4 &= C_4^{iv} \bmod 2^n - \mu_0 \cdot s
\end{aligned}
$$

Como se puede ver, cada vez que se reduce un nuevo coeficiente, éste afecta únicamente a los 4 coeficientes superiores a él, así que en primera instancia sólo se necesitan los coeficientes menos significativos en la siguiente iteración. Posteriormente, la forma en que la reducción de un coeficiente afecta a los coeficientes inmediatos superiores, es por medio de restas y sumas, las cuales se pueden ir almacenando antes de tener el coeficiente calculado.

La ventaja de esta forma de calcular la reducción es que se puede ir realizando en paralelo con la multiplicación de los coeficientes, además de que como sólo se necesitan los coeficientes menos significativos del polinomio $c(t)$, los polinomios más significativos pueden comenzarse a reducir en paralelo cuando estén listos.

5.2. Diseño de la Arquitectura

Una vez definido el algoritmo con el que opera el multiplicador, se describe la estructura de la arquitectura que realizará el trabajo. Se establece que a la entrada estarán los coeficientes de los polinomios $a(t)$ y $b(t)$, ya en el *Dominio Polinomial de Montgomery*. La arquitectura está diseñada para dividir el algoritmo en 4 partes principales, que son:

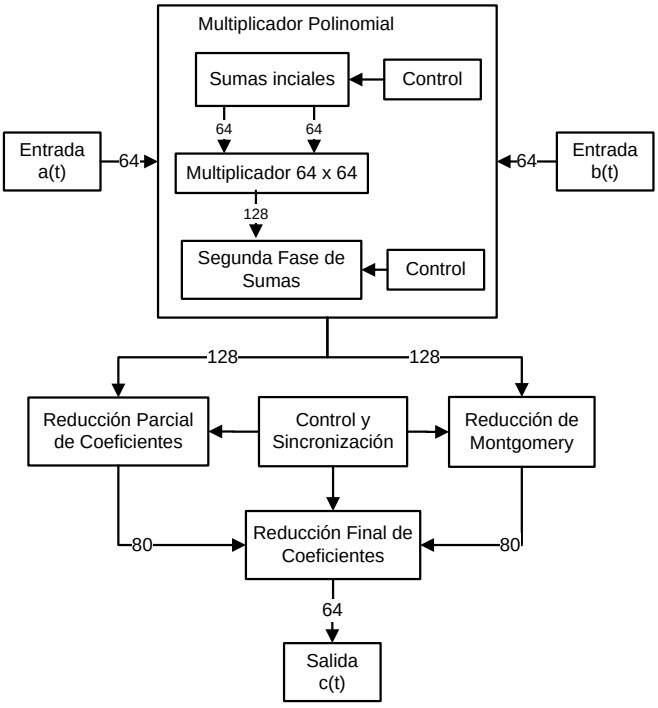

Figura 5.9: Diagrama a bloques del diseño del multiplicador $\mathbb{F}_p$

- El Multiplicador Polinomial($c(t) = a(t) \cdot b(t)$ utilizando Karatsuba de 5 elementos.
- La Reducción de Montgomery (la reducción del polinomio $c(t)$ de 9 coeficientes a uno de 4 coeficientes).
- La Reducción de Coeficientes (la realización de las operaciones de división y módulo t con $t = 2^{61} - 2^{15} + 1$).
- Una Reducción Final de Coeficientes, que se realiza a la salida.

En la figura 5.9, se aprecia la composición de la arquitectura. Se hace esta definición de los bloques con base a la forma en que fluyen los datos y cómo se van transformando. Como valores de entrada se tienen dos polinomios con 5 coeficientes, 4 de 64 bits y 1 de a lo más 6 bits, pero para efectos prácticos todos se manejan como si fueran de 64 bits. Al salir del multiplicador se tiene un nuevo polinomio con 9 coeficientes de 128 bits cada uno, en este punto hay que reducir el número de coeficientes del polinomio de 9 a 4, mediante la reducción de Montgomery. La reducción polinomial se hace junto con una

primera reducción de coeficientes de los coeficientes más significativos. Ambos módulos trabajan en paralelo, por lo que tienen una misma unidad de control y sincronización. Después de la reducción polinomial los coeficientes no necesariamente son menores a t (64 bits), así que es necesario realizar una reducción de coeficientes, la cual puede llegar a generar un quinto coeficiente. De esta manera en la salida se produce un polinomio de 5 coeficientes, cada uno de ellos menores a t, y donde el más significativo cumple con $|c_4| < 36$.

Cada una las etapas anteriores estan conectada a una unidad de control que son instrucciones en una memoria ROM que activan o desactivan los componentes, cara ROM de control tiene 15 palabras de control de una longitud dependiente del módulo que controlan.

5.2.1. Multiplicador Polinomial de 5 términos

El primero de los módulos de la arquitectura se encarga de realizar la multiplicación $c(t) = a(t) \cdot b(t)$ utilizando el algoritmo 5.1 de Karatsuba de 5 términos, que como se observado en la figura 5.9 se dividide en otros tres componentes que son:

- Entrada al multiplicador (Sumas Iniciales).
- Multiplicador 64×64 (Etapa de multiplicación).
- Sumas del multiplicador de Karatsuba (Segunda etapa de sumas).

A continuación la descripción detallada de cada uno de estos componentes, excepto del multiplicador de 64×64 pues ya se expuso en la sección 5.1.2.

Sumas iniciales

La entrada al multiplicador se puede observar en la figura 5.10, donde tenemos dos memorias de alta velocidad *DualPortRam* que almacenan los operandos de entrada con los cuales se va a alimentar al multiplicador. Entre las memorias y el multiplicador hay registros y sumadores, con el objetivo de realizar las sumas previas a cada producto, almacenar resultados para reutilizarlos y reducir la ruta crítica del diseño y así aumentar la frecuencia de trabajo. El control en este caso se enfoca en el direccionamiento de las memorias, para solicitar el valor correspondiente en cada momento, siguiendo la secuencia del algoritmo 5.1, de habilitar los multiplexores y registros, para almacenar valores futuros. Se puede observar que la arquitectura es simétrica, pues las operaciones que hay que realizar son equivalentes tanto para la entrada $a(t)$ como para $b(t)$, por lo que en lo que sigue se explica únicamente una de ellas. Se tienen los registros $SA0$, $SA1$, $SA2$ y $SA3$,

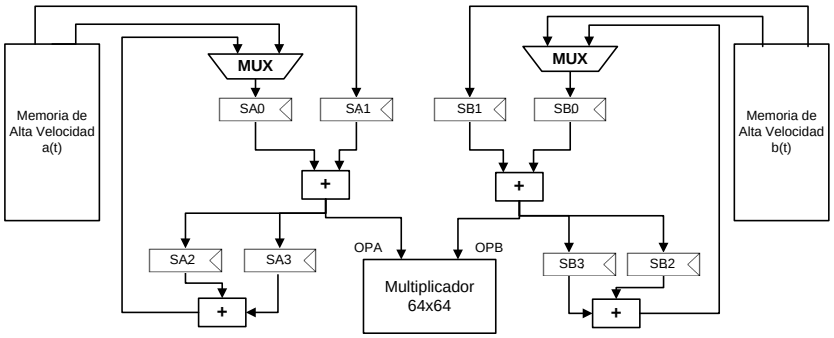

Figura 5.10: Primera etapa de la arquitectura llamada Sumas Iniciales

y dan como resultado el valor adecuado para el operando OPA del multiplicador, de esta manera a partir del algoritmo 5.1, tenemos que hay que evaluar las siguientes ecuaciones:

$OPA^{(0)}$	$=a_0$	$OPA^{(1)}$	$=a_1$
$OPA^{(2)}$	$=a_0+a_1$	$OPA^{(3)}$	$=a_2$
$OPA^{(4)}$	$=a_2+a_3$	$OPA^{(7)}$	$=a_1+a_3$
$OPA^{(8)}$	$=a_0+a_1+a_2+a_3$	$OPA^{(9)}$	$=a_4$
$OPA^{(10)}$	$=a_0+a_4$	$OPA^{(11)}$	$=a_0+a_1+a_4$
$OPA^{(12)}$	$=a_2+a_4$	$OPA^{(13)}$	$=a_2+a_3+a_4$

Para realizar estas operaciones la arquitectura funciona de la siguiente forma:

Tiempo en OPA	Operación	$SA0$	$SA1$	$SA2$	$SA3$
$OPA^{(-1)}$	$SA0$	a_0	0	$\perp$	$\perp$
$OPA^{(0)}$	$SA0$	a_1	0	$\perp$	$\perp$
$OPA^{(1)}$	$SA0+SA1$	a_1	0	$\perp$	$\perp$
$OPA^{(2)}$	$SA0$	a_2	0	a_0+a_1	$\perp$
$OPA^{(3)}$	$SA0$	a_0	a_2	a_0+a_1	$\perp$
$OPA^{(4)}$	$SA0+SA1$	a_3	0	a_0+a_1	$\perp$
$OPA^{(5)}$	$SA0$	a_2	a_3	a_0+a_1	$\perp$
$OPA^{(6)}$	$SA0+SA1$	a_1	a_3	a_0+a_1	a_2+a_3
$OPA^{(7)}$	$SA0+SA1$	$SA3+SA4$	0	a_0+a_1	a_2+a_3
$OPA^{(8)}$	$SA0$	a_4	0	a_0+a_1	a_2+a_3
$OPA^{(9)}$	$SA0$	a_0	a_4	a_0+a_1	a_2+a_3
$OPA^{(10)}$	$SA0+SA1$	$SA2$	a_1	a_0+a_4	0
$OPA^{(11)}$	$SA0+SA1$	a_2	a_4	a_0+a_4	0
$OPA^{(12)}$	$SA0+SA1$	$SA3$	a_3	a_2+a_4	0
$OPA^{(13)}$	$SA0+SA1$	0	0	0	0

Donde se puede apreciar que el registro que siempre está trabajando es el $SA0$, y ocasio-

nalmente trabaja $SA1$ para completar la suma, por último $SA2$ y $SA3$[1] sirven para almacenar valores temporales cuando el operando solicite una suma de 3 a 4 coeficientes del polinomio. El control se enfoca en enviar a las memorias la dirección del coeficiente necesario para la siguiente operación, habilitar los registros $SA2$ y $SA3$, para que se les cargue el valor del resultado de la suma, accionar si *reset* para darles un valor de 0 y de que el multiplexor **MUX** elija el valor de las suma de $SA2$ y $SA3$ cuando es necesario.

Sumas del multiplicador de Karatsuba o Segunda fase de sumas

Contruyendo un operador $SUMA(A, B, C, D) = (A + B) - (C + D)$, las operaciones de las líneas 16 a 25 del algoritmo 5.1 pueden ser reescritas de la siguiente forma:

$$\begin{aligned}
c_0 &= \text{Suma}(p_0, 0, 0, 0) \\
c_1 &= \text{Suma}(p_2, 0, p_1, p_0) \\
c_2 &= \text{Suma}(p_4, p_1, p_0, p_3) \\
S1 &= \text{Suma}(p_6, 0, p_5, p_3) \\
ACC1 &= -(c1 + S1) \\
c_3 &= \text{Suma}(ACC1, p_8, p_7, p_4) \\
ACC2 &= -p_0 + p_3 + p_5 - p_1 + p_7 \\
c_4 &= \text{Suma}(ACC2, p_{10}, p_9, 0) \\
ACC1 &= -p_1 + c1 - S1 \\
c_5 &= \text{Suma}(ACC1, p_{11}, p_{10}, 0) \\
c_6 &= \text{Suma}(p_{12}, p_5, p_3, p_9) \\
ACC2 &= -p_5 - S1 \\
c_7 &= \text{Suma}(ACC2, p_{13}, p_{12}, 0) \\
c_8 &= \text{Suma}(p_9, 0, 0, 0)
\end{aligned}$$

Por lo que se diseñó un componente especializado en realizar esta operación, que es mostrada en la figura 5.11. Las entradas que este componente recibe son directamente los resultados del multiplicador 64×64, un par de señales llamadas ACC1 y ACC2 que se explicarán más adelante y una señal de acceso a un banco de registros. El componente realiza la operación:

$$Salida = R1 + R2 - (R3 + R4)$$

Además $Ri1$ y $Ri2$ son registros intermedios, para reducir la ruta crítica del diseño y aumentar la frecuencia, a cambio de que el componente tenga una latencia de dos ciclos de reloj, que hay

[1]El símbolo $\perp$ denota que no conocemos el contenido del registro en ese momento

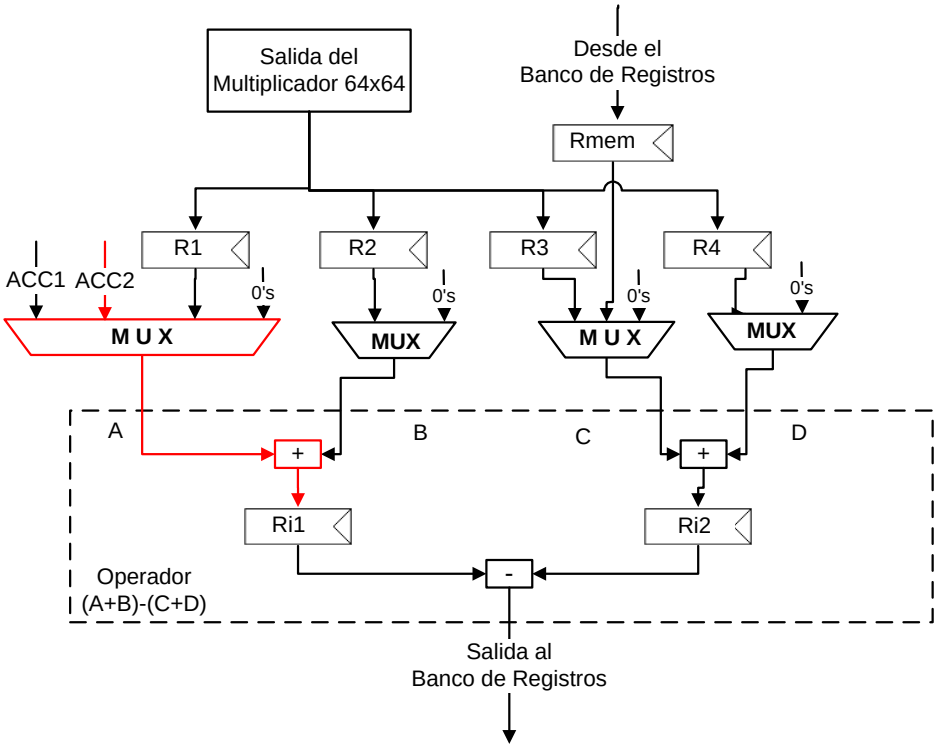

Figura 5.11: Sumadores de la arquitectura de la segunda fase de sumas

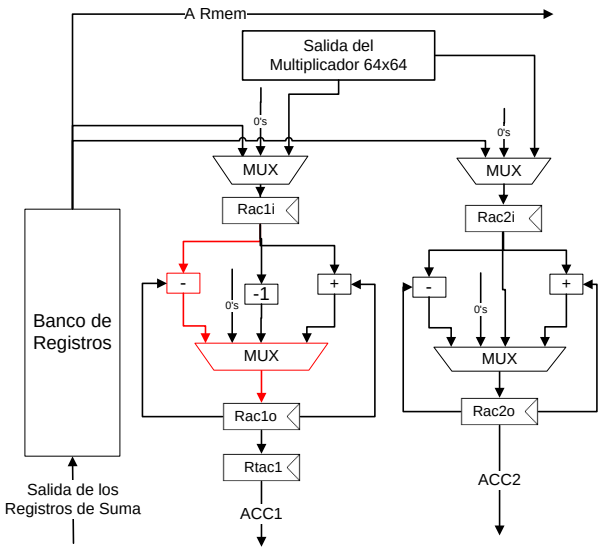

Figura 5.12: Acumuladores de la arquitectura se sumas

que tomar en cuenta en la sincronización. Los resultados van a un banco de registros, donde se almacena la salida final, estos registros también son usados como variables temporales, debido a que no son solicitados todos al mismo tiempo, si no que los componentes posteriores los van solicitando de manera secuencial. Se puede observar un registro llamado *Rmem* el cual es utilizado para tener acceso directamente al banco de registros y rescatar de forma rápida un resultado obtenido previamente. Entre los registros y los sumadores existen 4 multiplexores, los cuales permiten llevar a 0 un operando sin que se pierda el valor que hay almacenado en el registro correspondiente.

Las entradas $ACC1$ y $ACC2$ son las salidas de un par de acumuladores, mostrados en la figura 5.12, que son los encargados de servir de variables temporales, y procesar los coeficientes cuyo cálculo implica una complejidad mayor, como es el caso de los coeficientes c_3,c_4 y c_5; es decir las líneas 20, 21 y 22 del algoritmo 5.1.

Estos acumuladores trabajan en paralelo con el resto de la arquitectura, y pueden recibir valores tanto directamente del multiplicador de 64×64 o los valores ya calculados con anterioridad del banco de registros. Los valores de entrada pueden ser sumados, restados o servir de inicialización para los registros de resultado.

El acumulador con la salida de nombre $ACC1$ se compone de un multiplexor de 3 entradas 1 salida, para seleccionar si la entrada a procesar es del multiplicador 64×64, del banco de registros, o un 0. Posee también un registro de entrada $Rac1i$, donde se almacena el valor a ser procesado, para que después mediante un bloque multiplexor de 4 entradas una salida, se elija si la entrada va a ser sumada, restada, negada y asignada directamente o simplente ignorada;

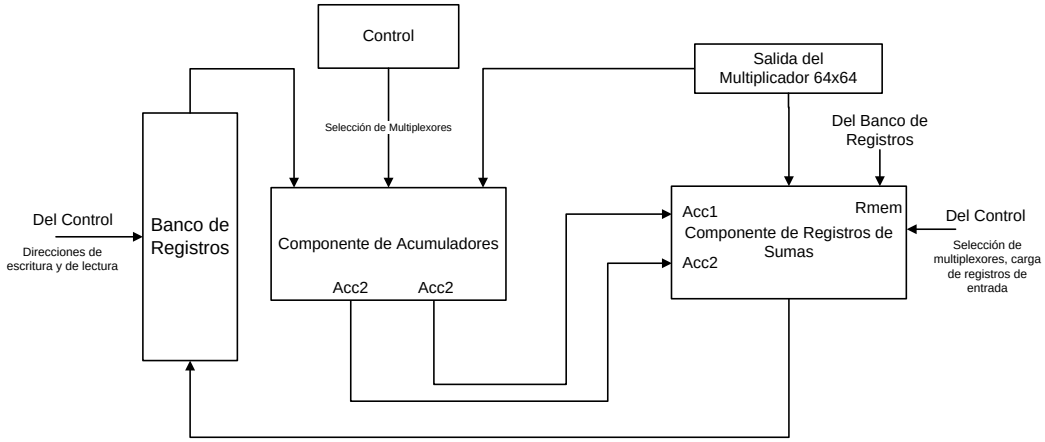

Figura 5.13: Sumas del algoritmo de Karatsuba

esto último para que se asigne un 0 al registro de salida $Rac1o$, sirviendo como inicialización. Además existe un registro nombrado como $Rtac1$, que sirve de temporal, para poder comenzar a acumular nuevos valores sin que el cálculo anterior se pierda. Por otro lado, el acumulador con la salida de nombre $ACC2$ es muy similar al de su contraparte de salida $ACC1$, pero el primero carece de un equivalente al registro temporal $Rtac1$ y no vuelve negativa la entrada al asignarla directamente al registro $Rac2o$.

En la figura 5.12 también se puede ver el banco de registros, que se compone de una memoria $DualPortRAM$ de 16 localidades de 128 bits. Una de las salidas de la memoria se utiliza como retoralimentación, y la otra salida es el medio de acceso por los módulos siguientes para obtener el resultado $c(t)$, la arquitectura completa de este módulo puede verse en la figura 5.13.

Debido a la necesidad que tenemos de mantener al multiplicador ocupado, esta arquitectura se encarga de tomar producto a producto y procesarlo, de forma que cuando el multiplicador termine de generar los 14 productos, tengamos a la salida los 9 coeficientes de 128 bits (en realidad son 8 de 128 bits y uno de 12 bits, pero todos se manejan con el mismo tamaño de palabra) correspondientes al polinomio de resultado, sin embargo se pierde un ciclo de reloj para la sincronización, por lo que en total, la arquitectura da un resultado completo del polinomio cada 15 ciclos de reloj. Cabe mencionar que en esta parte es donde se encuentra la ruta crítica de todo el diseño, la cual es la que va del el registro **Rac2o** al **Ri1**, es decir, un multiplexor de 4 entradas y un sumador, ambos de 128 bits, ofreciéndonos una frecuencia de 220 MHz. La ruta crítica de este componente esta empatada, marcada con rojo en las figuras 5.11 y 5.12.

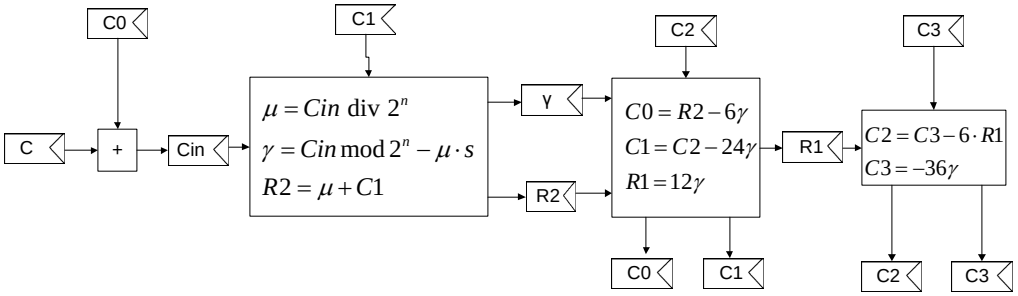

Figura 5.14: Bloques de Reducción de Mongomery para polinomios

5.2.2. Reducción Polinomial de Montgomery

Esta etapa es la encargada de reducir el polinomio de salida que tiene 9 coeficientes de 128 bits a un nuevo polinomio de sólo 4 coeficientes, basándose en el método ya descrito anteriormente en la subsección 5.1.5. Pensando en esta idea se diseñó una arquitectura cuyo propósito es calcular:

$$
\begin{aligned}
S_0 &= \mu_4 - 36\gamma_1 - 36\gamma_2 - 24\gamma_3 - 6\gamma_4 \\
S_1 &= -36\gamma_2 - 36\gamma_3 - 24\gamma_4 \\
S_2 &= -36\gamma_3 - 36\gamma_4 \\
S_3 &= -36\gamma_4
\end{aligned}
$$

Donde :

$$
\begin{aligned}
\mu_0 &= C_0 \text{ div } 2^n; & \gamma_0 &= C_0 \bmod 2^n - \mu_0 \cdot s \\
\mu_1 &= C_1' \text{ div } 2^n; & \gamma_1 &= C_1' \bmod 2^n - \mu_0 \cdot s \\
\mu_2 &= C_2'' \text{ div } 2^n; & \gamma_2 &= C_2'' \bmod 2^n - \mu_0 \cdot s \\
\mu_3 &= C_3''' \text{ div } 2^n; & \gamma_3 &= C_3''' \bmod 2^n - \mu_0 \cdot s \\
\mu_4 &= C_4^{iv} \text{ div } 2^n; & \gamma_4 &= C_4^{iv} \bmod 2^n - \mu_0 \cdot s
\end{aligned}
$$

De forma que cuando el componente termine su ejecución los resultados sean sumados con los coeficientes C_8, C_7, C_6 y C_5 originales del polinomio $c(t)$. Las operaciones anteriores se pueden observar sintetizadas en la figura 5.14, dónde cada bloque indica lo que ocurre en un ciclo de reloj, y en que registros se almacenan los resultados.

La arquitectura funciona en 3 etapas:

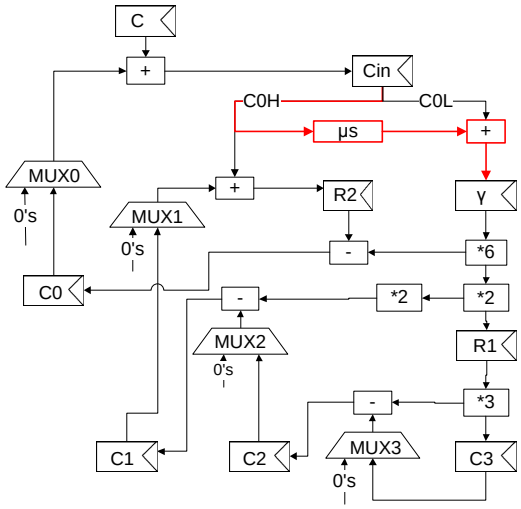

Figura 5.15: Etapa de Reducción de Mongomery para polinomios

1. Primero se realiza la suma del nuevo coeficiente de entrada, en el regustro C con el valor almacenado en el registro C0, de forma que el coeficiente se actualiza (se convierte en el C',C'', C''' o C^{iv} de las ecuaciones anteriores según sea el caso).

2. En la segunda etapa se obtiene $\mu = C$ div 2^n y se suma con el contenido anterior del registro $C1$, y este valor se almacena temporalmente en el registro $R2$, al mismo tiempo se cálcula el valor de $\gamma = C \bmod 2^n - \mu \cdot s$.

3. En la tercera etapa se calculan los nuevos valores de los registros $C0 = R2 + 6\gamma$ y $C_1 = C2 - 24\gamma$, el 24γ se calcula a partir de $6\gamma \cdot 2 \cdot 2$, además el registro $R1 = 12\gamma$.

4. Al tiempo que se hace la primera etapa, se realiza el calculo de $C2 = C3 - 36\gamma$ y $C3 = -36\gamma$, 36γ es obtenido a partir de $12\gamma \cdot 3$, se podría ver como una cuarta etapa que se realiza en paralelo con la primera.

A partir de ahí arquitectura propuesta es la que aparece en la figura 5.15 se basa en esta idea, de reducir un coeficiente a la vez.

El diseño de la figura 5.15, opera con números de 80 bits, a diferencia de la etapa anterior que trabaja con 128. Esto significa que se pueden realizar más operaciones por ciclos de reloj. La selección del parámetro t impacta el funcionamiento de este módulo en la arquitectura, definiendo su eficiencia y determinando el tamaño de los buses de datos dentro del componente.

El parámetro t usado, que es $t = 2^{61} - 2^{15} - 1$, lo que significa que $n = 61$, $s = -2^{15} - 1$, que

es un trinomio[2], convierte el producto $\mu \cdot s$ de las reducciones, en un desplazamiento de 15 bits a la izquierda y una resta; es decir:

$$\mu \cdot s \Rightarrow -\mu(<< 15) + \mu$$

Este desplazamiento de 15 bits provoca que haya elementos con una longitud de 76 bits, razón por la que los registros que se utilizan en esta etapa sean de una longitud de 80 bits; es decir hay una holgura de 4 bits para evitar errores de cálculo por desbordamiento, pues se realizan hasta 4 sumas sin reducción.

De esta manera toma 3 ciclos de reloj procesar cada coeficiente y debido a que son 5 coeficientes a reducir, la arquitectura requiere 15 ciclos de reloj parala generación de un resultado completo, además por motivo de las 3 etapas en las está dividido el componente se tiene una latencia de 3 ciclos de reloj.

El control de esta etapa se realiza mediante los multiplexores **MUX0**, **MUX1**, **MUX2**, **MUX3**, y en habilitar y deshabilitar el registro **R2**. Los multiplexores actuan en el caso cuando la arquitectura inicia con un nuevo producto, pues debido a que está construida en *pipe-line*, aunque el registro **Cin** ya tenga un valor correspondiente a la reducción de un nuevo polinomio $c(t)$, los registros inferiores no pueden ser borrados pues aún están procesando el polinomio anterior.

La ruta crítica de esta etapa es básicamente la acumulación de 3 sumas de 80 bits que está marcada en rojo en la figura 5.15, con lo que se consigue una frecuencia de 230 Mhz en el proceso de síntesis.

5.2.3. Reducción de Coeficientes

Esta etapa se encarga de la reducción de coeficientes, para que queden de un tamaño menor a t (en este caso 61 bits). La arquitectura se instancía 2 veces, una que realiza la reducción de los coeficientes más significativos del polinomio $c(t)$, es decir de C_8, C_7, C_6 y C_5, recién estos son cálculados, y trabaja en paralelo con la reducción polinomial de Montgomery, realizando su tarea en 5 ciclos de reloj (los últimos 5 ciclos de la reducción polinomial de Montgomery). El control está dado por señales que habilitan o deshabilitan el paso de los datos a través de los registros, este control es un módulo de sincronización, que se encarga de enviar a cada arquitectura el correspondiente producto para reducir, pues los coeficientes del C_0 al C_5 van a la reducción de Montgomery, y de C_6 a C_8, terminan en este componente, y de ahí pueden pasar a sumarse para pasar a la arquitectura final, que es la reducción final.

Esta reducción tiene su propio control interno, que se encarga de habilitar los registros intermedios y de salida.

[2]Entendiéndose por trinomio un número entero que tiene una representación binaria con sólo 3 bits en uno o menos uno

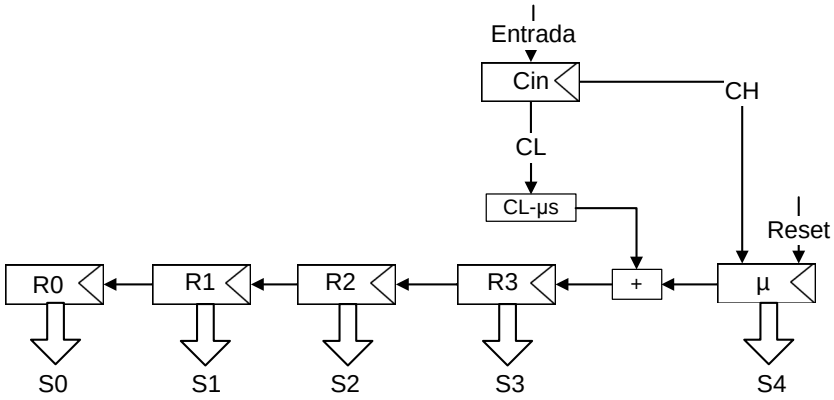

Figura 5.16: Arquitectura de las reducciones de coeficientes

La arquitectura recibe 4 coeficientes a la entrada, pero puede regresar 5, debido a que se genera un quinto coeficiente debido a las reducciones, este coeficiente cumple con la característica de que es menor a 36.

La ruta crítica en este componente son las 3 sumas que se realizan en $Cl - \mu \cdot s + \mu_{i-1}$, donde μ_{i-1} es resultado de una reducción previa, como se puede ver en el ciclo final del algoritmo 5.5.

5.3. Resultados de Implementación

De esta forma se realiza el producto polinomial completo con todo y reducción, a una frecuencia máxima de 220 Mhz con un espacio de 1983 *Slices* dónde 3238 LUT's (Cada *Slice* tiene 4 LUT's) se usaron como lógica y el resto como registros. La ruta crítica es la que va del registro **Rtac** al registro **Ri1en**, en la tercera etapa del producto polinomial de Karatsuba, con 40 ciclos de reloj de latencia y emitiendo un resultado nuevo cada 15 ciclos de reloj. Todos estos resultados fueron obtenidos usando la versión 13.2 de la herramienta ISE de Xilinx. Los resultados de implementación fueron hechos el dispositivo Virtex6 modelo XC6VLX130, en la simulación de ubicación y enrutamiento con una restricción de tiempo de 4.5 ns, y las simulaciones con la herramienta ModelSim.

En la tabla 5.2 se mencionan las características de cada una de las etapas del multiplicador en $\mathbb{F}_p$, donde como se puede observar el cuello de botella se encuentra en la **Segunda etapa de sumas**, correspondiente a las sumas que se realiza en el algoritmo de Karatsuba después de los productos. Las sumas que se realizan usan operandos de 128 bits, por lo que estos sumadores se convierten en la ruta crítica del diseño. También esta sección es muy grande debido al control tan robusto que necesita (22 señales de control) para poder ordenar las variables temporales conforme lo necesita el algoritmo 5.1.

Cuadro 5.2: Características de implementación en cada etapa

Etapa	Latencia	Frecuencia (MHz)
Entrada del Multiplicador	3	270
Multiplicador 64×64	4	313
Sumas	18	223.7
Reducción de Montgomery	18	230
Reducción de Coeficientes	15	223.7
Final	40	223.7

Capítulo 6

Arquitectura de un multiplicador en el campo $\mathbb{F}_{p^2}$

Una multiplicación eficiente en un campo $\mathbb{F}_{p^k}$, depende de la forma en que se construya la torre de campo para realizar las operaciones a partir del campo base $\mathbb{F}_p$.

Además como se describe en [3], para la construcción de la torre de campo es conveniente que la extensión tenga la forma $k = 2^a 3^b$, lo que implica que se necesitan realizar operaciones con polinomios de grado 2 y 3, en los cuales se puede utilizar el algoritmo de Karatsuba para hacer la multiplicación. Por ejemplo en [8] se decribe una torre de campo, donde se necesitan realizar una gran cantidad de operaciones en $\mathbb{F}_{p^2}$, tanto multiplicaciones como operaciones de elevar al cuadrado, siendo estos últimos particularmente eficientes.

En este capítulo se presenta la descripción de una de arquitectura eficiente para realizar la multiplicación en $\mathbb{F}_{p^2}$, que por motivo de costos, se recomienda que se utilice también para llevar a cabo la operación de elevar al cuadrado y las multiplicaciones en el campo $\mathbb{F}_p$.

6.1. El Algoritmo de Multiplicacion en $\mathbb{F}_{p^2}$

El campo $\mathbb{F}_{p^2}$ es una extensión del campo $\mathbb{F}_p$, es decir que un elemento $A \in \mathbb{F}_{p^2}$ se ve como un polinomio $A = a_0 + a_1 i$ (parte real y parte imaginaria), donde $a_0, a_1 \in \mathbb{F}_p$. Dicho campo se define a partir de un polinomio irreducible $P(u)$, que típicamente por razones de eficiencia es un binomio con la forma $P(u) = u^2 + \beta$, donde i es una solución del polinomio, es decir $P(i) = i^2 + \beta = 0$, lo que implica que $i^2 = -\beta$.

Para la construcción de una multiplicación eficiente, también se elige β con un valor muy pequeño, como por ejemplo en [8] se propone a $\beta = 5$. En base a lo anterior se construye el algoritmo 6.1 de multiplicación, que es presentado en [8], y se basa en el algoritmo de Karatsuba

para reducir el número de multiplicaciones en el campo $\mathbb{F}_p$ que se necesitan para obtener un producto en el campo $\mathbb{F}_{p^2}$.

Este multipliplicador está diseñado con el objetivo de ser usado en aplicaciones de criptografía de curvas elípticas de 128 bits de seguridad, razón por la cual los operandos tienen una longitud de 256 bits.

Algoritmo 6.1 Algoritmo de multiplicación en $\mathbb{F}_{p^2}$ presentado en [8]

Entrada: $X, Y \in \mathbb{F}_{p^2}$, donde $X = x_0 + x_1 u$ y $Y = y_0 + y_1 u$, y $u^2 = -51$
Salida: $W = X \cdot Y \in \mathbb{F}_{p^2}$
1: $s \leftarrow$ add256(x_0, x_1)
2: $t \leftarrow$ add256(y_0, y_0)
3: $d_0 \leftarrow$ mult256(s, t)
4: $d_1 \leftarrow$ mult256(x_0, y_0)
5: $d_2 \leftarrow$ mult256(x_1, y_1)
6: $d_0 \leftarrow$ sub512(d_0, d_1)
7: $d_0 \leftarrow$ sub512(d_0, d_2)
8: $w_1 \leftarrow$ mod512(d_0)
9: $d_2 \leftarrow 5d_2$
10: $d_1 \leftarrow$ sub512(d_1, d_2)
11: $w_0 \leftarrow$ mod512(d_1)
12: **devolver** $W = w_0 + w_1 u$

En el algoritmo 6.1 se puede observar que se requieren 3 multiplicaciones y 2 reducciones para efectuar el producto en $\mathbb{F}_{p^2}$. No se toma en cuenta la multiplicación de la línea 9, pues el producto por 5, al ser "101" su representación binaria, se puede sustituir por una suma y un desplazamiento a la izquierda, operaciones que son sencillas de realizar en hardware.

Por otro lado el algoritmo 6.1 sirve para elevar al cuadrado un elemento de $\mathbb{F}_{p^2}$, de forma que es un caso particular de multiplicación en el que los elementos de entrada son iguales. Esta caracterísitica permite usar un truco heredado de los números complejos conocido como el *conjugado*, dónde se ahorra una multiplicación, que unido con el uso del algoritmo de Karatsuba permite calcular el resultado haciendo uso de sólo dos multiplicaciones de 256 bits.

6.2. Diseño de la Arquitectura

A continuación únicamente se realiza el desarrollo de la arquitectura para la multiplicación, basándose en el algoritmo 6.1, pues es una operación más costosa que elevar al cuadrado, debido a que requiere una multiplicación extra, pero no se deja de remarcar que ambos módulos son importantes al diseñar una arquitectura completa para un emparejamiento.

Algoritmo 6.2 Algoritmo para elevar al cuadrado en $\mathbb{F}_{p^2}$ presentado en [8]
1
Entrada: $X \in \mathbb{F}_{p^k}$, donde $X = x_0 + x_1 u$, y $u^2 = -5$
Salida: $W = X^2 in \mathbb{F}_{p^2}$
$t \leftarrow$ add256(x_1, x_1)
$d_1 \leftarrow$ mul256(t, x_1)
$t \leftarrow$ sub256$(x_0, x, 1)$
$t \leftarrow$ add256(t, p)
$w_1 \leftarrow 5x_1$
$w_1 \leftarrow$ add256(x_0, w_1)
$d_0 \leftarrow$ mul256(t, w_1)
$w_1 \leftarrow$ mod512(d_1)
$d_1 \leftarrow$ add256(d_1, d_1)
$d_0 \leftarrow$ sub512(d_0, d_1)
$w_0 \leftarrow$ mod512(d_0)
devolver $W = w_0 + w_1 u$

A partir del algoritmo 6.1 se puede observar los componentes que se necesitan para poder realizar la multiplicación:

- Multiplicadores de 256×256 bits
- Sumadores y restadores de hasta 512 bits
- Elementos de reducción de 512 bits

Con estos componentes se desarrolla la arquitectura que se puede observar en la figura 6.1.

Es importante notar que antes de realizar la operación de $d1 - 5d2$, a $d1$ se le suma el número $p \cdot 2^{250}$, con el fin de evitar que el resultado de la resta nunca sea negativo, pues la arquitectura aún no tiene soporte para este tipo de elementos, y las alteraciones que pudiera provocar esta operación desaparecen cuando se aplica la operación de reducción modulo p. Por otro lado, se cuidó que todas las interconexiones tuvieran un tamaño de 128 bits, que se consideraría como el tamaño del bus de datos, lo que pese a consumir una cantidad importante de recursos, permite hacer muy eficiente el diseño, pues 128 bits es el límite que imponen las operaciones que definen la ruta crítica son las sumas como se describirá más adelante.

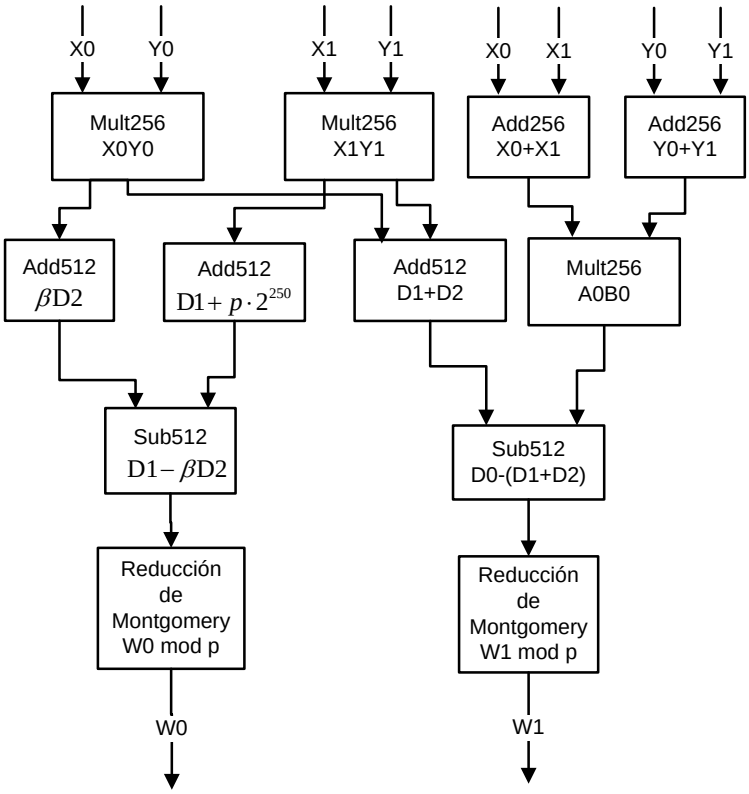

Figura 6.1: Arquitectura general del multiplicador en $\mathbb{F}_{p^2}$

6.3. Sumadores y Restadores de Números Grandes

Una de las operaciones más delicadas cuando se trabaja con implementaciones en hardware y campos finitos primos, donde el primo que define al campo es muy grande (aproximadamente 256 bits en este caso), es la suma entera, y por lo tanto la resta. Esta operación es especialmente costosa, debido a que no se puede conocer el resultado de la suma de dos bits sin conocer el acarreo generado por la suma de los bits que les antescedieron. Lo anterior implica que si se realiza una suma de 64 bits, existan al menos 64 sumadores en cascada en la ruta crítica del diseño, por lo que para el caso de sumas de 512 bits por ejemplo hay que hacer uso de distintas tecnicas para atacar esta problemática.

Para realizar las sumas y restas de números grandes en un *FPGA*, existen dos opciones principales. La primera es construir los sumadores con los elementos lógicos *LUT*'s, contenidos en los *Slices* que a su vez están incorporados en los componentes *CLB*'s, conectándolos en cascada y de esa forma obtener un sumador con el tamaño de palabra que se desee.

La segunda opcion son los sumadores dedicados de los componentes *DSPSlices*, los cuales son muy rápidos y poseen un tamaño de palabra fijo de 48 bits. Sin embargo cuando se quieren utilizar en cascada o se mezcla su funcionamiento con lógica externa a los *DSPSlices*, la frecuencia de trabajo disminuye dramáticamente, lo que obligo al diseño a optar por construir los sumadores con los elementos lógicos *LUT*'s, pues nos dan una mayor flexibilidad y velocidad cuando hablamos de sumas de más de 48 bits.

Para poder reducir los pormenores que implica la propagación del acarreo en sumas enteras de números grandes se toman dos medidas principales, la primera es la división de los operandos en palabras, para de alguna forma "cortar" la propagación del acarreo y reducir el tamaño de los buses de datos, y la segunda es el uso de la técnica conocida como *Carry Save*, para poder realizar una gran cantidad de sumas a una mayor velocidad.

El tamaño de palabra que se utilizó en esta implementación fue de 128 bits, debido a que otorga un equilibrio entre velocidad y eficiencia al realizar las sumas, pues entre más se dividan los operandos, más ciclos de reloj son necesarios para procesarlos. Por ejemplo, en esta implementación, el tamaño máximo de los operandos que se van a sumar es de 512, que si son divididos en palabras de 64 bits, se obtiene una frecuencia muy alta, más de 280 MHz, pero es muy costoso en cuanto a la cantidad de registros que se necesitan y requiere 8 ciclos de reloj para completar el resultado,lo cual no es competitivo. Por otro lado si se divide el operando en palabras de 256 bits, la frecuencia cae demasiado (apenas los 200 Mhz), y con 128 bits, se tiene un equilibrio entre la frecuencia de trabajo (250 Mhz) y los ciclos de reloj (4 ciclos de reloj para números de 512 bits), sin embargo las anteriores son velocidades ideales que pueden variar de acuerdo al ruteo de las señales así como de la lógica extra utilizada con el sumador.

La primera arquitectura para realizar sumas es la mostrada en la figura 6.2, que no es más que un sumador de 128 bits, con una retroalimentación en el acarreo, permitiendo realizar sumas consecutivas y después de 4 ciclos realizar sumas de 512 bits, por ejemplo. La ventaja de este sumador es que es muy flexible y puede utilizarse para realizar sumas de cualquier

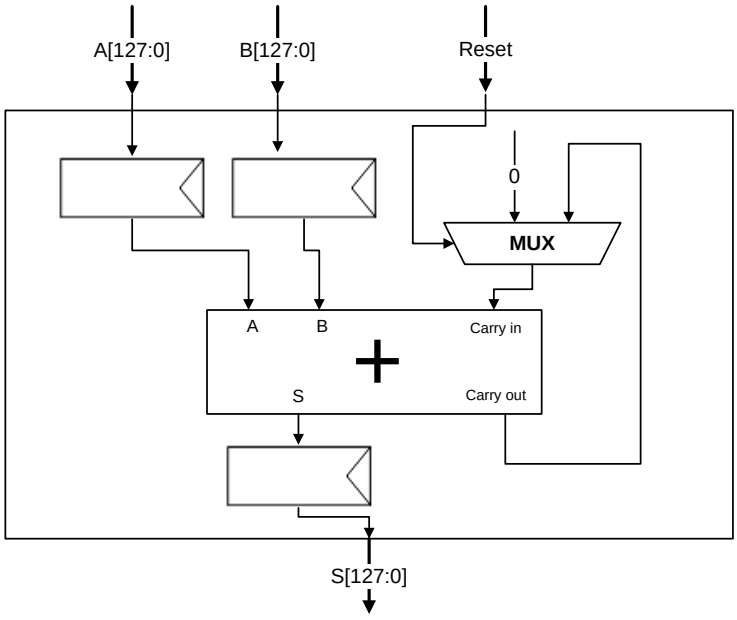

Figura 6.2: Sumador serial de 128 bits utilizado para sumas de 512 bits

tamaño, el cual determinará el tiempo necesario para que se realice una adición completa. En la retroalimentación del acarreo se puede observar que puede ser forzada a tomar un valor 0, de esta manera cuando se requiere realizar una suma con nuevos operandos se evita que los acarreos generados por resultados anteriores afecten los nuevos.

Estos sumadores se crearon utilizando los *IPCores* de la herramienta ISE de Xilinx, permitiendo implementar sumadores a la medida del diseño, y configurándolos según sea conveniente. Los sumadores en este caso de 128 bits se configuraron en un modo de trabajo **sin signo**, y con un acarreo de entrada y uno de salida, para poder retroalimentarlo, esta retroalimentación es la que podría perjudicar un poco el rendimiento del diseño, reduciendo la velocidad de 250 Mhz a un valor entre 240 y 230 dependiendo del ruteo, pero está desventaja está más que compensada por su flexibilidad y la posibilidad de usarlos en diseños basados en la técnica de *Pipe-Line*(Tubería).

Para los restadores se utiliza exactamente la misma técnica, pero con una variación en el acarreo. Primero hay que recordar cómo se compone una resta en la representación de complemento a 2, que es:

$$S = A - B = A + \bar{B} + 1 \tag{6.1}$$

Como se puede observar la resta es una suma, pero el sustraendo es negado lógicamente(es decir se invierte el valor de sus bits), y con la suma de un valor de 1, es decir, con el acarreo

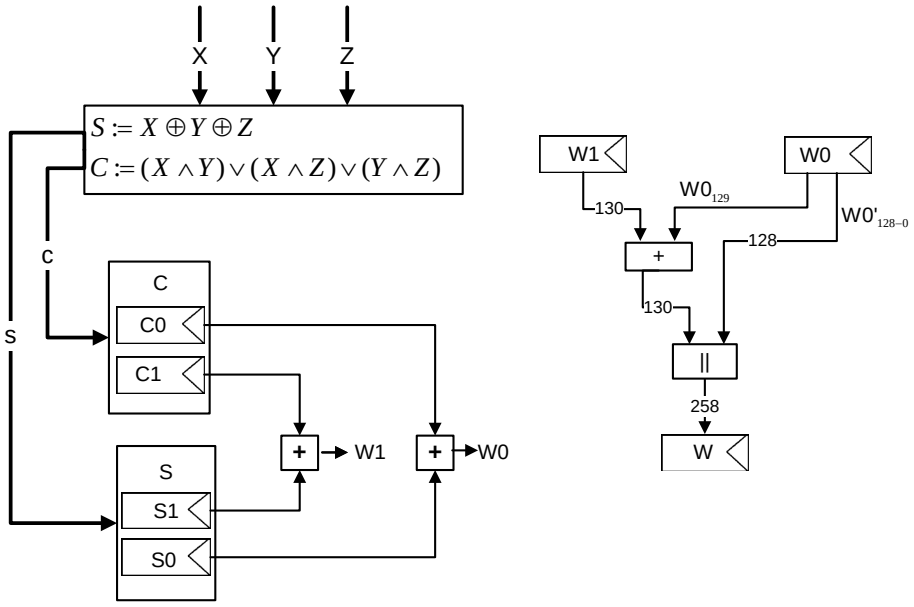

Figura 6.3: Sumador *Carry-Save* de 256 bits con 3 operandos de entrada

de entrada activado. Cuando se construye un restador utilizando la herramienta de *IPCores*, el componente resultante automáticamente niega el valor del sustraendo, pero es en la configuración del restador dónde se tiene que aplicar el acarreo de entrada con valor de 1, es por ello que un restador se puede ver igual que un sumador, pero con un valor de 1 en vez de 0 en el *reset* de la retroalimentación.

Los sumadores seriales, como se les ha llamado en este trabajo a los sumadores descritos anteriormente, son muy útiles y eficientes cuando es necesario realizar una suma con operandos de gran longitud, pues reducen la ruta crítica al precio de realizar la operación en varios ciclos de reloj, sin embargo cuando son una gran cantidad de sumas, es necesario encontrar otro método que sustituya o complemente a la suma serial.

El método que se propone usar en este trabajo es el conocido como *Carry-Save Adder*, o sumadores sin acarreo, el cual es una solución para realizar más de una suma con un sólo sumador. Dicho método ya se explicó brevemente en el la sección 4.1.1 del capítulo 4, pero retomaremos su funcionamiento para observar sus ventajas más a detalle.

La técnica de *Carry-Save* en el caso más simple permite realizar 2 sumas con un poco más del costo de una suma, teniendo su base en realizar la suma en dos pasos, el primero es calcular las operaciones lógicas para los acarreos C y para la suma sin acarreo S como se describe a continuación:

$$S = X \oplus Y \oplus Z$$
$$C = (X \wedge Y) \vee (Y \wedge Z) \vee (Z \wedge X)$$

Las cuales son operaciones de tiempo constante con un retardo mínimo, pues los bits no tienen dependencias entre sí, y el segundo paso es realizar la suma de estos resultados parciales.

En la arquitectura de la figura 6.3 se muestra cómo se realiza la suma de 3 operandos, cada uno de ellos con un tamaño de 256 bits. Se realiza el cálculo de la suma, almacenándo los acarreos en C mientras que la suma sin acarreo se guarda en S, estos resultados tienen una longitud de 257 bits cada uno.

Posteriormente S y C son procesados en 1 palabra de 128 bits y una palabra de 129 bits, donde las palabras de S son sumadas con las palabras de C, obteniendo 1 palabra de 129 bits y otra de 130 bits, nombradas como $W0$ y $W1$, respectivamente.

Después es necesario unir estas 2 palabras para obtener el resultado final, de forma que se propague el acarreo que generó la operación de suma entera. Para realizar esto la palabra $W1$ es sumada con el valor del bit más significativo de $W0$.

Es importante remarcar el hecho de que $W1$ es el resultado de la suma de dos números de 129 bits, los cuales podrían tener cada uno un valor máximo de $2^{129} - 1$, es decir, la palabra $W1$ puede tener un valor máximo de $2(2^{129} - 1) = 2^{130} - 2$, número al cual si se le suma un 1, que es el valor más grande posible del acarreo de $W0$, el resultado es $2^{130} - 1$, por lo que el resultado sigue siendo de a lo más 130 bits.

A continuación se concatena el resultado de la suma anterior con los primeros 128 bits de $W0$ para obtener $W = (W1 + W0^{129})||(W0^{128-0})$, que es el resultado final W con una longitud de 258 bits es el resultado de la suma de X,Y y Z.

De esta manera el sumador presentado en la figura 6.3 realiza la suma de 3 operandos con una frecuencia de 250 Mhz, el punto débil de la propuesta es la cantidad de registros necesarios para ir acumulando los resultados parciales, y la latencia debido al uso de la técnica de pipe-line para que se obtenga un nuevo resultado de la suma cada ciclo de reloj. El diseño de la figura 6.3 tiene una latencia de 4 ciclos de reloj y consume 3 registros extras para unir los operandos.

Una característica interesante de estos sumadores es que se puede generalizar, por ejemplo si se quiere realizar una suma de 4 operandos, se generan los valores S y C para 3 de ellos, y al hacer la unión, se puede volver a aplicar la técnica *Carry-Save* para unir S, C y el cuarto operando, y al final se une con una sola suma.

Sin embargo hay que tener cuidado, pues con cada operando que se agrega los buses deben de crecer 1 bit, por lo que tal vez no siempre sea conveniente esta solución, pero si hay que realizar muchas sumas en paralelo, se pueden realizar a una muy buena velocidad, pues es el costo de una suma normal, más algunas operaciones lógicas independientes entre sí.

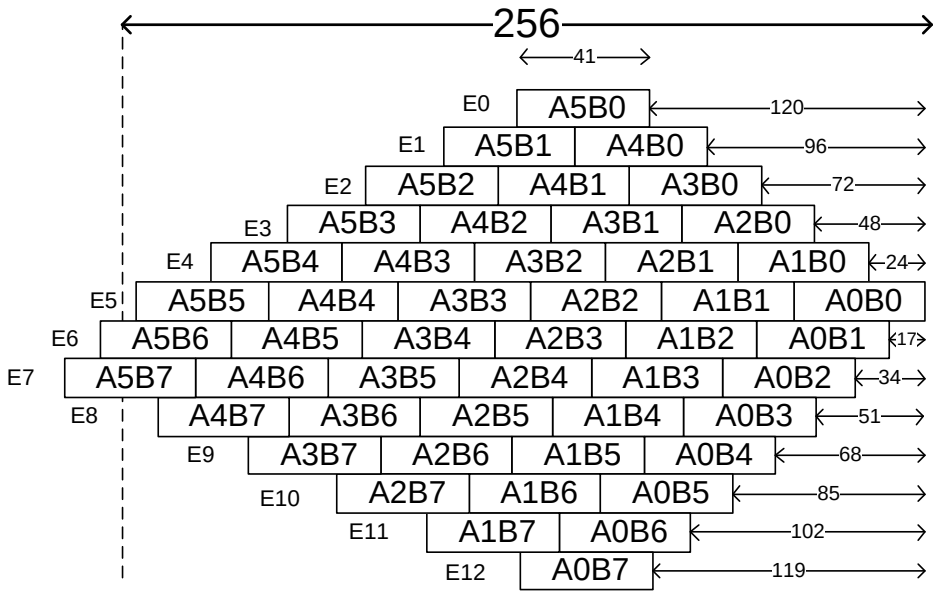

Figura 6.4: Estructura de los subproductos para el multiplicador de 128 × 128

6.4. El multiplicador de 256×256

Para realizar los productos en el campo $\mathbb{F}_p$ se construyó un multiplicador de 256 bits. La primera idea para esta construcción fue usar el método descrito en la sección 5.1.2, utilizando los *DSP48Slices*, sin embargo, al hacer el diseño preeliminar se determinó que se necesitarían 165 *DSP48Slices*, además de que al concatenar los resultados la propagación del acarreo comienza a disminuir la frecuencia de trabajo, por lo que este método se considero demasiado costoso. Por otro lado, igualmente basándose en el método descrito en la sección 5.1.2, se contruyó un multiplicador de 128x128 bits, que tiene una latencia de 7 ciclos de reloj, pero arroja un resultado nuevo cada ciclo, y consume 48 *DSP48Slices*, lo que hace más viable su implementación.

6.4.1. Multiplicador 128×128

A diferencia del multiplicador de 64×64 bits, este diseño de 128×128 bits no hace uso intensivo de los registros y sumadores de los *DSP48Slices*, si no que únicamente utiliza los multiplicadores asimétricos, pues las sumas y el almacenamiento de subproductos se hacen de manera externa, debido a que los resultados son muy grandes para los registros internos de estos componentes empotrados.

En la figura 6.4 se puede observar como son concatenados y ordenados los 48 subproductos para realizar la suma y obtener el resultado de la multiplicación, el diamante consta de 13 niveles

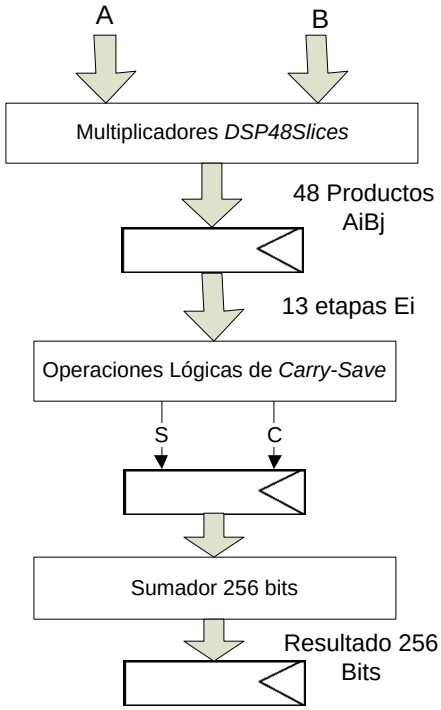

Figura 6.5: Arquitectura por bloques del multiplicador de 128×128

E_i, lo que implica que se requieren 12 sumas de 256 bits para obtener el resultado. Para efectuar estas sumas se utilizó un sumador *Carry-Save* como el descrito en la sección anterior, haciendo uso del truco de usar la representación de suma y acarreo sucesivamente, de manera que sólo se requiera hacer una suma al después de las operaciones lógicas, de esta manera se obtiene una arquitectura como la mostrada en la figura 6.5.

En el primer bloque de la figura se utilizan los 48 *DSP48Slices* para realizar los 48 productos en paralelo, después se concatenan en registros para obtener los 13 niveles E_i del diamante. Una vez con estas etapas, se realizan las siguientes operaciones lógicas basadas en las ecuaciones de los sumadores *Carry-Save*:

$$
\begin{aligned}
S_{00} &= E_0 \oplus E_1 \oplus E_2 \\
C_{00} &= (E_0 \wedge E_1) \vee (E_1 \wedge E_2) \vee (E_2 \wedge E_0) \\
S_{01} &= E_3 \oplus E_4 \oplus E_5 \\
C_{01} &= (E_3 \wedge E_4) \vee (E_4 \wedge E_5) \vee (E_5 \wedge E_3) \\
S_{02} &= E_6 \oplus E_7 \oplus E_8 \\
C_{02} &= (E_6 \wedge E_7) \vee (E_7 \wedge E_8) \vee (E_8 \wedge E_6) \\
S_{03} &= E_9 \oplus E_{10} \oplus E_{11} \\
C_{03} &= (E_9 \wedge E_{10}) \vee (E_10 \wedge E_11) \vee (E_{11} \wedge E_9) \\
C_{00} &= C_{00} <<_1 \\
C_{01} &= C_{01} <<_1 \\
C_{02} &= C_{02} <<_1 \\
C_{03} &= C_{03} <<_1 \\
S_{10} &= S_{00} \oplus S_{01} \oplus S_{02} \\
C_{10} &= (S_{00} \wedge S_{01}) \vee (S_{01} \wedge S_{02}) \vee (S_{02} \wedge S_{00}) \\
S_{11} &= C_{00} \oplus C_{01} \oplus C_{02} \\
C_{11} &= (C_{00} \wedge C_{01}) \vee (C_{01} \wedge C_{02}) \vee (C_{02} \wedge C_{00}) \\
S_{12} &= S_{03} \oplus C_{03} \oplus E_{13} \\
C_{12} &= (S_{03} \wedge C_{03}) \vee (C_{03} \wedge E_{13}) \vee (E_{13} \wedge S_{03}) \\
C_{10} &= C_{10} <<_1 \\
C_{11} &= C_{11} <<_1 \\
C_{12} &= C_{12} <<_1 \\
S_{20} &= S_{10} \oplus S_{11} \oplus S_{12} \\
C_{20} &= (S_{10} \wedge S_{11}) \vee (S_{11} \wedge S_{02}) \vee (S_{12} \wedge S_{10}) \\
S_{21} &= C_{10} \oplus C_{11} \oplus C_{12} \\
C_{21} &= (C_{10} \wedge C_{11}) \vee (C_{11} \wedge C_{02}) \vee (C_{12} \wedge C_{10}) \\
C_{20} &= C_{20} <<_1 \\
C_{21} &= C_{21} <<_1 \\
S_{30} &= S_{20} \oplus C_{20} \oplus S_{21} \\
C_{30} &= (S_{20} \wedge C_{20}) \vee (C_{20} \wedge S_{21}) \vee (S_{21} \wedge S_{20}) \\
C_{30} &= C_{30} <<_1 \\
S &= S_{30} \oplus C_{30} \oplus C_{21} \\
C &= (S_{30} \wedge C_{30}) \vee (C_{30} \wedge C_{21}) \vee (C_{21} \wedge S_{30})
\end{aligned}
$$

obteniendose dos números S y C de 256 bits, estas operaciones a pesar de ser númerosas son muy rápidas debido a que no hay interdependencia entre los los bits de los operandos. Además se sabe que el resultado no será mayor a 256 bits, implicando que las sumas no generaran acarreos mas allá de eso, por lo que el bus de datos no crece.

De esta manera, se puede obtener un nuevo producto después de 5 ciclos de reloj a una frecuencia de 230 Mhz, pero utilizando la técnica de *pipe-line*, cada ciclo de reloj se obtiene un nuevo producto.

6.4.2. Arquitectura del multiplicador 256×256

Tomando como base principal el multiplicador de 128×128 y un sumador de elementos de 128 bits, se construye la arquitectura vista en la figura 6.6, la cual es capaz de procesar un nuevo producto de 256 bits cada 4 ciclos de reloj.

La arquitectura de la figura 6.6 realiza su tarea mediante el algoritmo de *libro de texto*, entonces para realizar un producto de 256×256 necesita de 4 productos de 128×128, y los registros $R1$, $R2$, $R3$, $R4$, $R5$ y $R6$ se encargan de organizar los subproductos para que sean sumados, y los registros $R4$,$R5$ y $R6$ tienen señales de reset para ïnyectar"0's cuando resulte conveniente.

Si llamamos S_0,S_1,S_2 y S_3 a los 4 subproductos de salida del multiplicador de 128×128, y siendo S_0^h y S_0^l los 128 bits más significativos y menos significativos de S_0 respectivamente, entonces para calcular el resultado final necesitamos hacer las siguientes sumas:

$$
\begin{aligned}
W_0 &= 0 + S_0^l + 0 \\
W_1 &= S_0^h + S_1^l + S_2^l \\
W_2 &= S_1^h + S_2^h + S_3^l \\
W_3 &= 0 + S_3^h + 0
\end{aligned}
$$

Donde W_0,W_1,W_2 y W_3 son palabras del operando de salida, y como se puede ver cada una puede ser calculada con una suma de 3 operandos de 128 bits, de manera que al final es necesario concatenar estas palabras y propagar los acarreos resultantes de las sumas. De esta tarea se encarga la parte final de la arquitectura. Además existe un multiplexor de "reset", para evitar que algún acarreo obtenido al terminar de concatenar las 4 palabras de una suma completa provoque errores en los resultados futuros.

Así tenemos un multiplicador de 256×256, que nos entrega un resultado completo cada 4 ciclos de reloj, y tiene una latencia de 12 ciclos de reloj. Una caracterísitica de diseño es que cada vez que tiene una nueva palabra del operando de salida procesada ya puede ser utilizada por los módulos siguientes, es decir que el multiplicador está emitiendo resultados parciales cada ciclo de reloj.

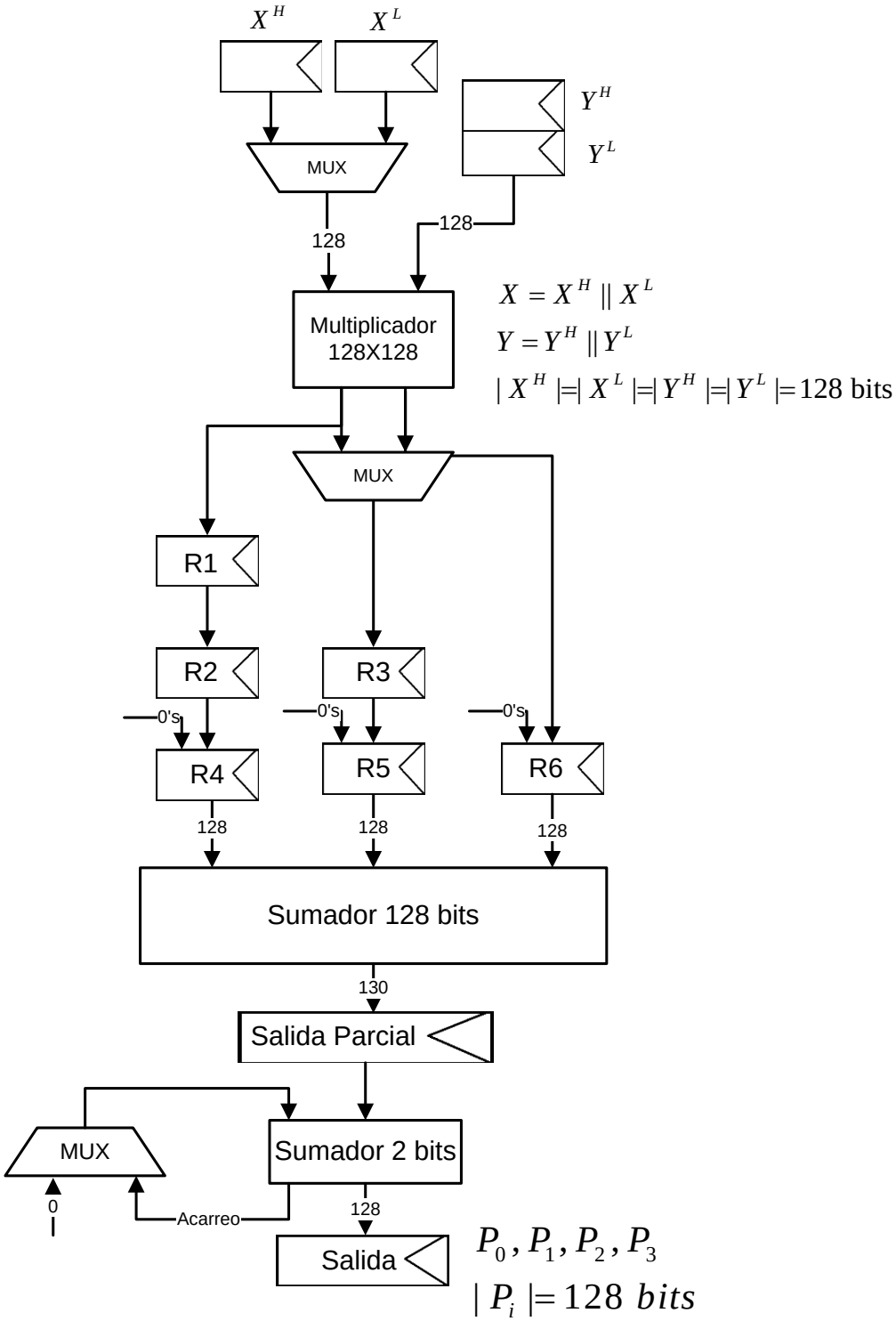

Figura 6.6: Arquitectura del multiplicador de 256 bits

6.5. Construcción de la Reducción de Montgomery

Ya con los dos módulos descritos anteriormente se puede construir el último elemento necesario en la arquitectura, que es el componente de reducción de Montgomery, para poder realizar las reducciones finales de los elementos de 512 bits a 256 bits que estén en $\mathbb{F}_p$. Para realizar la reducción se utiliza el algoritmo de Montgomery ya descrito en la seccion 4.2.6 del capítulo 4, en la cual se describe que se necesitan los parámetros que cumplan con la condición:

$$\ell \cdot \ell^{-1} - pp' = 1 \tag{6.2}$$

Donde en el caso de este diseño, $\ell = 2^{250}$ y p es el primo obtenido al evaluar el polinomio:

$$p(t) = 36t^4 + 36t^3 + 24t^2 + 6t + 1 \tag{6.3}$$

En este caso de las pruebas para validar el diseño se asignó al parámetro t el valor de $t = 2^{61} - 2^{15} + 1$, pero puede utilizarse cualquier parámetro y de hecho cualquier valor de p, siempre y cuando tenga menos de 256 bits.

La reducción de Montgomery, plasmada en el algoritmo 6.3, requiere de 2 multiplicaciones de 256 bits, pero como la multiplicación de $W \cdot p'$, es reducida módulo ℓ, y en este caso es 2^{250}, la parte alta del resultado se descarta (véase la Subsección 4.2.6, por lo que únicamente hay que calcular 3 multiplicaciones de 128 bits.

Algoritmo 6.3 Reducción de Montgomery

Entrada: $W = \bar{X} \cdot \bar{Y}$,los parámetros ℓ, p', y el primo p
Salida: $\bar{W} = W \cdot \ell^{-1} \mod p$

```
u ← W · p' mód ℓ
W̄ ← (W + u · p)/ℓ
si W̄ > p entonces
    W̄ ← W̄ − p
fin si
devolver W̄
```

Así con 2 multiplicadores, uno de 128 bits y otro de 256 bits construimos la arquitectura de la figura 6.7, que se encarga de realizar la reducción de Montgomery de un número de 512 bits a uno de 256 bits.

EL operando W, que en general es el resultado de una multiplicación de dos operandos de 256 bits, se divide en palabras de 128 bits W_0,W_1,W_2 y W_3, para ser reducido. Estas palabras primero son almacenadas en una memoria *FIFO*[1], debido a que se van a requerir al final en las

[1]De las siglas en ingles de *First In First Out*, el primero en entrar es el primero en salir, en español

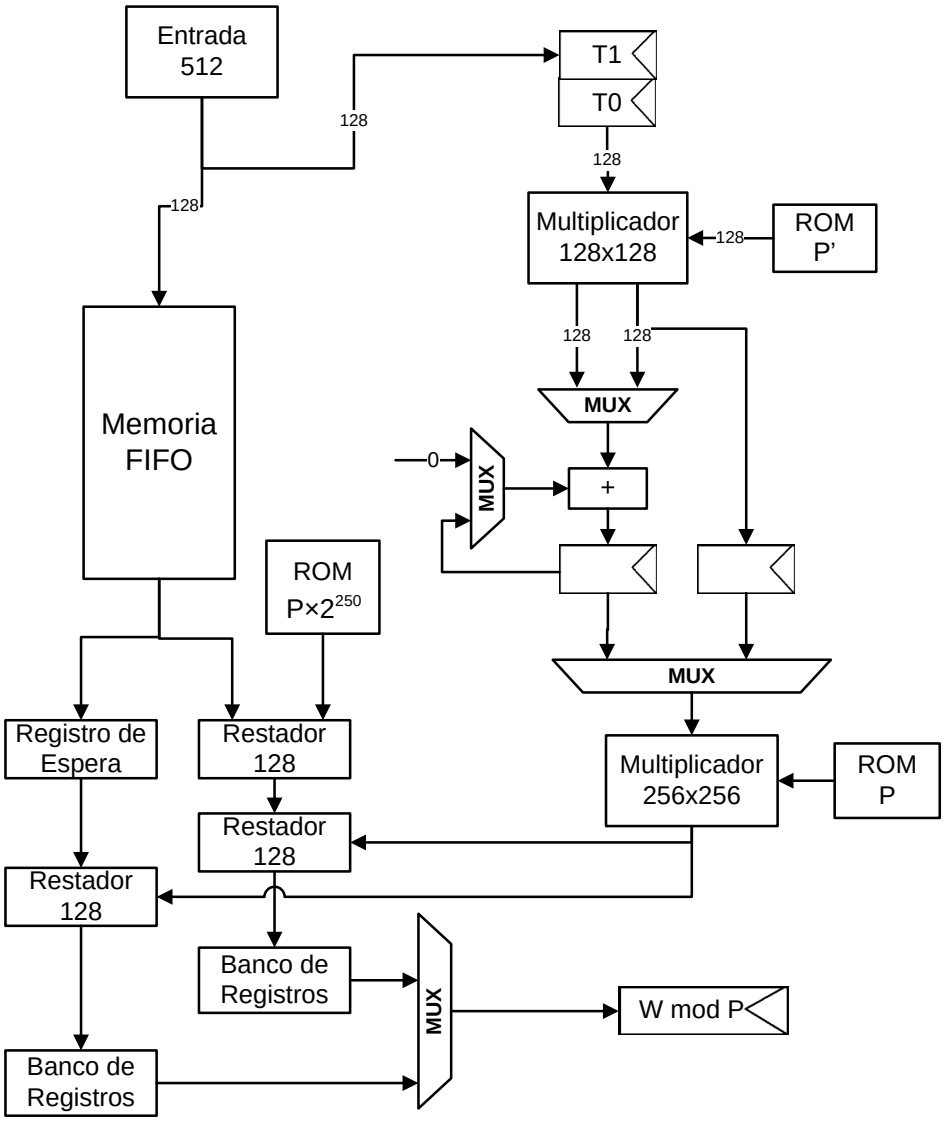

Figura 6.7: Módulo de reducción de Montgomery para números de 512 bits

operaciones de multiplicación. Las palabras W_0 y W_1 pasan y se almacenan en los registros $T1$ y $T0$ respectivamente, como variables temporales para enviarlas al multiplicador de 128 bits, cuando este así lo necesite.

Los subproductos que necesitan ser realizados son:

$$S_0 = W_0 \cdot P'_0$$
$$S_1 = W_0 \cdot P'_1$$
$$S_2 = W_1 \cdot P'_0$$

Donde P'_1y P'_0 son las partes alta y baja del número p' respectivamente, y está almacenado en palabras de 128 bits en una memoria ROM, donde los valores quedan almacenados en orden P_0,P_1 y P_0, para hacer más sencilla la multiplicación con los elementos de 128 bits. Para obtener u del algoritmo 6.3, se necesitan realizar las siguientes sumas:

$$u = (S_0^h + S_1^l + S_2^l)||S_0^h \tag{6.4}$$

Estos subproductos son unidos mediante una etapa de un par de registros y un acumulador que está entre los dos multiplicadores, para que el resultado pueda entrar ahora al multiplicador de 256 bits. Mediante un multiplexor se elige qué resultados entraran al acumulador para obtener la parte alta de u, y del resultado se tomarán los bits correspondientes a la parte alta del resultado dividido entre ℓ^2.

Como se puede ver en el algoritmo 6.3, al final del proceso se requiere restarle el valor de p al resultado si éste es mayor a dicho valor. Esta operación aunque pudiera parecer sencilla, como se realiza por medio de una condición, afecta la sincronicación del diseño, además de que aumenta ciclos de latencia para obtener el resultado final. Para dar solución a este problema la resta se aplica antes, sobre los valores que están almacenados en la memoria, de forma que se bifurcan los resultados, y a los dos se les resta el valor que de salida del multiplicador de 256×256, por lo que tienen dos resultados, que van a un *buffer* de almacenamiento temporal, una vez que se efectuan las dos restas, si el resultado al que se le aplicó la resta previamente es positivo, se emite a la salida, si por el contrario es negativo, el resultado de salida es el que no se le aplico resta previa.

Hay que notar que como la resta se realiza antes de la división entre ℓ el valor que se resta a los valores en la memoria es $p \cdot \ell$, o en este caso $p \cdot 2^{250}$. Además hay dos razones para tener el buffer de salida, la primera es que hasta que no se terminan de hacer las restas, no se puede saber cual es el resultado que se debe emitir, y la segunda es que como la división puede que no sea exactamente entre 2^{256}, implica que los bits más significativos de la segunda palabra pueden ser parte del resultado final, por lo que se necesita realizar un reacomodo de palabras.

[2]Es muy importante si se va a utilizar un primo de otra longitud hay que modificar esta parte para extraer la cantidad adecuada de bits, de lo contrario la reducción no funcionará

Este módulo trabaja a 235 Mhz, debido a todos los componentes que se incorporaron, y entrega un resultado reducido cada 4 ciclos de reloj, en 2 palabras de 128 bits, y 2 palabras con 0's.

6.6. Nota de un multiplicador en $\mathbb{F}_p$

Esta sección nació con el objetivo de realizar una nueva propuesta en base a lo desarrollado en el capítulo 5, dónde se describe un multiplicador para para el campo $\mathbb{F}_p$, pero realizando un cambiando de enfoque proponiendo utilizar más recursos y así obtener mayor velocida, explotando la cantidad de componentes *DSP48Slices* de la familia **Virtex 6**. Sin embargo como se ha observado a lo largo de esta sección, el resultado aunque efectivo, es un poco costoso en terminos de recursos, y al realizar productos en el campo $\mathbb{F}_{p^2}$ es difícil compararlo con otras implementaciones. Por lo anterior, una alternativa que se encuentra a medio camino, entre velocidad y precio, es construir un multiplicador $\mathbb{F}_p$, con los componentes desarrollados a lo largo de este capítulo, es decir, con un multiplicador de 256×256 y un componente de reducción de Montgomery, con lo que se obtiene un multiplicador que da un nuevo producto cada 4 ciclos de reloj, pero requiriendo únicamente 144 multiplicadores *DSP48Slices*, muchos más que el diseño del capítulo 5, pero menos de la mitad que los necesarios para el multiplicador del presente capítulo, a una frecuencia de 235 Mhz, y con una latencia aproximada de 35 ciclos de reloj.

6.7. Resultados

Cada elemento del multiplicador tiene su propio control interno si lo necesita, y todos son orquestados por un control superior que se encarga de activar los multiplicadores y los elementos de reducción, y aplicar la señal de reset a los sumadores y restadores cuando es necesario. Este control se encuentra implementado en forma de una memoria ROM, con las palabras de control que salen hacia los multiplexores de los componentes, y hacia los registros, cargando datos y habilitando el flujo cuando sea necesario.

En total se emite un nuevo producto en $\mathbb{F}_{p^2}$ cada 4 ciclos de reloj en 4 palabras de 128 bits, por dos puertos, y el diseño tiene una latencia de 50 ciclos de reloj, con una frecuencia máxima de 235 Mhz.

La ruta crítica son los sumadores de 128 bits con los distintos elementos lógicos que los rodean, pues no se puede fijar un sumador como el que determina de forma definitiva el máximo retardo de la señal.

Por otro lado, en términos de espacio, la memoria de las reducciones de Montgomery es el elemento más costoso, debido a la gran cantidad de registros que se utilizan para construir la memoria *FIFO*, y el uso de la técnica de *pipe-line*. Son necesarios 7 multiplicadores de 128×128 para obtener los coeficientes del polinomio $w_0 + w_1 u$, y cada multiplicador hace uso de 48

DSP48Slices, por lo que en total se requieren 336 *DSP48Slices*, y el modelo de *FPGA* con el que se realizaron las pruebas de implementación fue una **XC6VLX130T**, que contiene 488 *DSP48Slices*. Este diseño resulta bastante rápido si se requieren hacer una gran cantidad de productos independientes entre sí, además de que no depende de un tipo especial de primo p para trabajar, a diferencia del diseño de multiplicador $\mathbb{F}_p$, sin embargo debido la latencia tan alta que posee, puede no ser conveniente cuando los productos son dependientes entre sí, pues el costo verdadero de los productos hace que se vuelva menos atractiva la arquitectura presentada.

Capítulo 7

Resultados y Conclusiones

En este trabajo se exploraron distintas opciones para mejorar la multiplicación de operandos de entrada de 256 bits el campo finito primo $\mathbb{F}_p$, en su extensión cuadrática $\mathbb{F}_{p^2}$, mediante las ventajas de la implementación en hardware. Se buscó utilizar los principales componentes incorporados en los nuevos modelos de hardware reconfigurable, como son los *DSP48Slices* y las memorias *DualPortBRAM*, los cuales tienen mucho potencial en aplicaciones criptográficas como se señala en [24]. Los enfoques en $\mathbb{F}_p$ y $\mathbb{F}_{p^2}$ fueron debido a que son las operaciones muy utilizadas en emparejamientos bilineales con 128 bits de seguridad.

7.1. Resultados

A continuación se resumen las principales características las arquitecturas desarrolladas a lo largo de esta tesis en la tabla 7.1.

Cuadro 7.1: Comparación entre implementaciones presentadas

Multiplicador	Ciclos de trabajo	Latencia	*No. DSP48Slices*	Frecuencia(Mhz)
$\mathbb{F}_p$ capítulo 5	15	40 ciclos	12	220
$\mathbb{F}_{p^2}$ capítulo 6	4	45 ciclos	336	235
Propuesta $\mathbb{F}_p$ capítulo 6	4	35 ciclos	144	235

Como se puede observar en el resultado final se ofrecen 3 propuestas, la del capítulo 5, que es un poco lenta, pero muy ligera, y la presentada en el capítulo 4.13, que por el contrario es mucho más rápida pero ofreciendo un tiempo de trabajo muy pequeño (casi 8 veces menor) y utilizando una gran cantidad de *DSPSlices*. Además como solución intermedia se propone el posible uso de los componentes del multiplicador en $\mathbb{F}_{p^2}$ para un multiplicador en $\mathbb{F}_{\text{ı}}$, que es un poco costoso pero igual muy eficiente.

Otro punto importante a tratar es la escalabilidad, pues el diseño del capitulo 5 es un poco más díficil de escalar que la arquitectura para $\mathbb{F}_{p^2}$, debido a que el primero usa el algoritmo de Karatsuba, y si existen una mayor cantidad de productos, hay que rediseñar el control y la **Segunda Fase de Sumas**, otra solución sería incrementar el tamaño de los registros, pero aquello impacataría de forma negativa sobre la frecuencia máxima de trabajo. Mientras tanto por otro lado el diseño de $\mathbb{F}_{p^2}$ es más flexible, y al usar el método de *libro de texto* para las multiplicaciones, un aumento en el tamaño de los multiplicadores no afectaría demasiado el control, aunque si la latencia que de por sí ya es muy grande. Lo anterior es una cuestión vital a resolver si se quiere subir de nivel de seguridad, pues ese paso requeriría operandos con una mayor cantidad de bits.

Posterioremente en las tablas 7.2 y 7.3 se pueden observar una comparación entre el diseño para campos $\mathbb{F}_p$ del capítulo 5 desarrollado en este trabajo con otros diseños actuales. El multiplicador en $\mathbb{F}_p$ se diseñó originalmente con el objetivo de ser implementado en un dispositivo de la familia **Virtex5**, donde el número de *DSP48Slices* es reducido, mientras que los diseños orientados a los dispositivos de la familia *Virtex 6* utilizan los componentes *DSP48Slices* de una manera más agresiva, pues hay bastantes disponibles en esta familia.

Cuadro 7.2: Comparación con implementaciones recientes (**A**)

Diseño	Ciclos por producto	Latencia	Frecuencia
[19]	23	0	204 Mhz
[20]	5	25	210 Mhz
[46]	15	8	250 Mhz
$\mathbb{F}_p$ capítulo 5	15	40	223.7 Mhz
$\mathbb{F}_{p^2}$ capítulo 6	4	45	235 Mhz
$\mathbb{F}_p$ capítulo 6	4	35	235 Mhz

Cuadro 7.3: Comparación con implementaciones recientes (**B**)

Diseño	Dispositivo	Area	*DSP48Slices*
[19]	ASIC	183 kGates	-
[20]	Virtex 6	4014 Slices	46
[46]	Virtex 6	-	36
$\mathbb{F}_p$ capítulo 5	Virtex 6	1983 Slices	12
$\mathbb{F}_{p^2}$ capítulo 6	Virtex 6	8 754 Slices	336
$\mathbb{F}_p$ capítulo 6	Virtex 6	-	144

A primera vista puede parecer un poco lento frente a otros diseños más nuevos, pero gracias

al uso del algoritmo de Karatsuba, se ahorran muchos recursos tanto *DSP48Slices* como *Slices* comunes, siendo el resultado la mitad de pequeño que el mas ligero de sus competidores, por lo que puede resultar una opción viable para dispositivos limitados.

Por otro lado para el multiplicador en $\mathbb{F}_{p^2}$ se tomo una estrategia totalmente contraria, buscando sobre todo velocidad y explotando la mayor cantidad de *DSP48Slices* disponibles. De esta manera se consiguió que tuviera una frecuencia de trabajo máxima de 235 Mhz, y el diseño presentado puede ofrecer un nuevo resultado cada 4 ciclos de reloj, con una latencia de 45 ciclos de reloj, lo que puede resultar inconveniente con operaciones cuyos productos sean muy dependientes entre si, pues hay que esperar un poco más de 12 productos para por cada resultado, pero si se hace una buena planeación para sortear esta desventaja, resulta un diseño muy competitivo pues realiza el quivalente a 3 multiplicaciones en $\mathbb{F}_p$ cada 4 ciclos de reloj, muy superior a los diseños que compiten directamnte con el multiplicador del capítulo 5.

7.2. Conclusiones

A través del desarrollo de estas arquitecturas se puedemos obtener las siguientes conclusiones:

- En las aplicaciones con números grandes, resulta conveniente utilizar los *DSP48Slices* para realizar las multiplicaciones, en especial y basándose en el método de multiplicación visto en [38], con el cual se pueden obtener resultados muy eficientes, de esta forma algoritmo de **Suma de enteros** que se utilice definirá la eficiencia del diseño. Lo anterior debido a que la frecuencia de trabajo dependerá directamente de cómo sea la propagación del acarreo en la lógica entera.

- La técnica de *pipe-line* resultó muy efectiva en este tipo de aplicaciones, siempre y cuando no exista una diferencia demasiado alta entre la latencia y el tiempo de trabajo de cada resultado o se realice una buena planificación de las operaciones. Lo anterior debido a que una latencia demasiado alta puede perjudicar el desempeño de los algortimos cuyos productos son dependientes entre sí, debido a que hay que esperar todos los ciclos de latencia para obtener un nuevo producto, en caso de existir dependencias.

- En una implementación sobre hardware, siempre es más conveniente un control sencillo, debido a que es más rápido y fácil de programar, pues no hay demasiadas líneas de control ni multiplexores que reduzcan la frecuencia de trabajo. Además, hay que tener cuidado con algoritmos que requieren el uso de una gran cantidad de valores temporales, pues no es conveniente almacenarlas en registros, si no usar una memoria, que en general es más lenta. Debido a las razones anteriores, en ocasiones no es conveniente utilizar el algoritmo de Karatsuba, pues entre más complejo se vuelve requiere una mayor cantidad de variables temporales y sumadores. Otra ventaja de un control sencillo es que es más escalable y no

es necesario realizar modificaciones muy importantes cuando se incluyan nuevos elementos o cuando se cambie de dispositivo.

- El algoritmo de Montgomery para enteros puede superar en velocidad a su versión para polinomios planteada por [19], gracias a los multiplicadores *DSP48Slices*, pero requieren de mucha más área debido a las etapas de *pipe-line*, además que al no usar un número primo p con características especiales, hay que realizar 3 multiplicaciones en vez de sólo una, lo cual lo hace muy costoso en área.
- La propuesta de multiplicador en $\mathbb{F}_{p^2}$ consigue compensar su tamaño con su velocidad, al poder producir un nuevo producto cada 4 ciclos de reloj, sin embargo aún necesita mejoras consistentes en los sumadores, para poder reducir la cantidad de registros en *pipe-line* y así reducir su latencia.
- Durante el proceso de diseño, se pudo comprobar que realizar directamente un multiplicador de 256 bits utilizando la técnica mostrada en [38] resulta demasiado costosa por el momento, debido al tamaño de los sumadores y la cantidad de *DSP48Slices* necesarios, aunque en un dispositivo futuro como una **Virtex 7** puede resultar interesante explorar esta implementación.

7.3. Trabajo a futuro

Para poder mejorar el desempeño de las arquitecturas desarrolladas proponemos las siguientes ideas para un trabajo a futuro:

- Buscar e implementar más técnicas de adición para números enteros grandes y así reducir el retardo que causan los mismos en la propagación de las señales.
- Comenzar a trabajar en multiplicadores más grandes, para aplicaciones con mayor nivel de seguridad.
- Usar de forma más agresiva los componentes *DSP48Slices*, no sólo configurados en forma de multiplicador, si no también realizar diseños basados en los registros y sumadores internos que poseen.
- Buscar nuevas técnicas de multiplicación y valores para el primo p, que puedan hacer las reducciones más eficientes.
- Implementar de forma completa y realizar modificaciones al multiplicador de $\mathbb{F}_{p^2}$ para poder encontrar una relación adecuada entre la latencia, y el tiempo de trabajo de cada producto, de forma que sea más práctico cuando se requiere hacer productos con mucha dependencia entre sí.

Bibliografía

[1] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[2] D. F. Aranha, K. Karabina, P. Longa, C. H. Gebotys, and J. López. Faster explicit formulas for computing pairings over ordinary curves. In *Proceedings of the 30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology*, EUROCRYPT'11, pages 48–68, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] S. Baktir and B. Sunar. Optimal tower fields. *Computers, IEEE Transactions on*, 53(10):1231–1243, oct. 2004.

[4] P. S. L. M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography - SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer, 2006.

[5] J.-L. Beuchat, N. Brisebarre, J. Detrey, E. Okamoto, M. Shirase, and T. Takagi. Algorithms and arithmetic operators for computing the ηt pairing in characteristic three. *IEEE Trans. Comput.*, 57:1454–1468, November 2008.

[6] J.-L. Beuchat, J. Detrey, N. Estibals, E. Okamoto, and F. Rodriguez-Henriquez. Hardware accelerator for the Tate pairing in characteristic three based on Karatsuba-Ofman multipliers. Cryptology ePrint Archive, Report 2009/122, 2009.

[7] J.-L. Beuchat, J. Detrey, N. Estibals, E. Okamoto, and F. Rodríguez-Henríquez. Fast architectures for the ηt pairing over small-characteristic supersingular elliptic curves. *IEEE Transactions on Computers*, 99, 2010.

[8] J.-L. Beuchat, J. E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya. High-speed software implementation of the optimal ate pairing over barreto-naehrig curves. In M. Joye, A. Miyaji, and A. Otsuka, editors, *Pairing-Based Cryptography - Pairing 2010*, volume 6487 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2010.

[9] J.-L. Beuchat, E. López-Trejo, L. Martínez-Ramos, S. Mitsunari, and F. Rodríguez-Henríquez. Multi-core implementation of the Tate pairing over supersingular elliptic curves. In *Proceedings of the 8th International Conference on Cryptology and Network Security*, CANS '09, pages 413–432, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques*, EUROCRYPT'03, pages 416–432, Berlin, Heidelberg, 2003. Springer-Verlag.

[11] S. Brown and J. Rose. Fpga and cpld architectures: A tutorial. In *IEEE Design & Test of Computers*, pages 42–57, 1996.

[12] Ç. Kaya Koç, T. Acar, and B. J. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *Micro, IEEE*, 16(3):26–33, June 1996.

[13] J. Chung and M. Anwar Hasan. Montgomery reduction algorithm for modular multiplication using low-weight polynomial form integers. In *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, pages 230–239, june 2007.

[14] D. R. Coelho. *The VHDL Handbook*. Kluwer Academic Publishers, Norwell, MA, USA, 1989.

[15] A. M. D. Hankerson and S. Vanstone. Guide to elliptic curve cryptography, 2004.

[16] W. Diffie and M. E. Hellman. New directions in cryptography, 1976.

[17] R. Dutta, R. Barua, and P. Sarkar. Pairing-based cryptographic protocols : A survey. Cryptology ePrint Archive, Report 2004/064, 2004. http://eprint.iacr.org/.

[18] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[19] J. Fan, F. Vercauteren, and I. Verbauwhede. Faster F_p-arithmetic for cryptographic pairings on Barreto-Naehrig curves. In *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 240–253. Springer-Verlag, 2009.

[20] J. Fan, F. Vercauteren, and I. Verbauwhede. Efficient hardware implementation of F_p-arithmetic for pairing-friendly curves. *Computers, IEEE Transactions on*, PP(99):1, 2011.

[21] D. Freeman, M. Scott, and E. Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23:224–280, 2010. 10.1007/s00145-009-9048-z.

[22] P. Grabher, J. GroBschädl, and D. Page. On software parallel implementation of cryptographic pairings. In R. Avanzi, L. Keliher, and F. Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *Lecture Notes in Computer Science*, pages 35–50. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-04159-4 3.

[23] M. Graphics. Modelsim-advace simulation and debugging.

[24] T. Güneysu. Utilizing hard cores of modern FPGA devices for high-performance cryptography. *Journal of Cryptographic Engineering*, 1:37–55, 2011.

[25] A. Joux. A one round protocol for tripartite diffie-hellman. *Journal of Cryptology*, 17:263–276, 2004. 10.1007/s00145-004-0312-y.

[26] D. Kammler, D. Zhang, P. Schwabe, H. Scharwächter, M. Langenberg, D. Auras, G. Ascheid, and R. Mathar. Designing an ASIP for cryptographic pairings over Barreto-Naehrig curves. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 254–271. Springer, 2009.

[27] N. Koblitz. Elliptic curve cryptosystems. In *Mathematics of Computation 48*, pages 209–209, 1987.

[28] N. Koblitz and A. Menezes. Pairing-based cryptography at high security levels. In *Proceedings of Cryptography and Coding 2005, volume 3796 of LNCS*, pages 13–36. Springer-Verlag, 2005.

[29] A. Menezes, S. Vanstone, and T. Okamoto. Reducing elliptic curve logarithms to logarithms in a finite field. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 80–89, New York, NY, USA, 1991. ACM.

[30] A. J. Menezes, P. C. V. Oorschot, S. A. Vanstone, and R. L. Rivest. Handbook of applied cryptography, 1997.

[31] V. S. Miller. Use of elliptic curves in cryptography. In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[32] P. Montgomery. Five, six, and seven-term *K*aratsuba-like formulae. *Computers, IEEE Transactions on*, 54(3):362–369, 2005.

[33] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[34] F. Rodríguez-Henríquez, N. A. Saqib, A. Díaz-Pérez, and C. K. Koc. *Cryptographic Algorithms on Reconfigurable Hardware (Signals and Communication Technology)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[35] A. Shamir. Identity-based cryptosystems and signature schemes. In G. Blakley and D. Chaum, editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer Berlin / Heidelberg, 1985. 10.1007/3-540-39568-7 5.

[36] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Science/Engineering/Math, first edition, July 2004.

[37] V. Shoup. *A computational introduction to number theory and algebra*. Cambridge University Press, New York, NY, USA, 2005.

[38] S. Srinath and K. Compton. Automatic generation of high-performance multipliers for FPGAs with asymmetric multiplier blocks. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 51–58, New York, NY, USA, 2010. ACM.

[39] L. C. Washington. *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Chapman & Hall/CRC, 2 edition, 2008.

[40] Xilinx. FPGA families.

[41] Xilinx. Ise webpack design software.

[42] Xilinx. Spartan 3, 2008.

[43] Xilinx. Isim user guide, September 2009.

[44] Xilinx. *Virtex 6 FPGA Configurable Logic Blocks User Guide*. Xilinx, September 2009.

[45] Xilinx. *Virtex 6 FPGA DSP48E1 Slice User Guide*. Xilinx, February 2011.

[46] G. X. Yao, J. Fan, R. C. Cheung, and I. Verbauwhede. A high speed pairing coprocessor using rns and lazy reduction. Cryptology ePrint Archive, Report 2011/258, 2011. http://eprint.iacr.org/.