Arquitectura hardware compacta para criptografía ligera de llave pública

Por:

Luis Armando Rodríguez Flores

Tesis sometida como requisito parcial
para obtener el grado de:

MAESTRÍA EN CIENCIAS EN LA ESPECIALIDAD DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica

Noviembre, 2014
Tonantzintla, Puebla

Dirigida por:

Dr. René Cumplido
Dr. Miguel Morales Sandoval

Agradecimientos

En especial quiero agradecerle a mi familia por su gran apoyo y amor. Un agradecimiento especial a mis asesores Dr. René Cumplido y Dr. Miguel Morales, por la colaboración, paciencia, apoyo y sobre todo por esa gran amistad que me brindaron, por escucharme y aconsejarme siempre. Agradezco a todos los profesores del Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE) que me impartieron algún curso o algún consejo.

Sabiendo que no existirá forma alguna de agradecer una vida de sacrificios, esfuerzos y amor, quiero que sientan que el objetivo alcanzado también es de ustedes y que la fuerza que me ayudó a conseguirlo fue su gran apoyo.

Luis Armando Rodríguez Flores.

Resumen

En la actualidad la interconexión de diversos dispositivos es más común que nunca. Cada vez se conectan más dispositivos a Internet para gran diversidad de propósitos, por ejemplo: aplicaciones médicas, militares, científicas, etc. De la misma forma que la comunicación en pequeños dispositivos con recursos limitados aumenta, también aumentan los posibles riesgos de seguridad en las comunicaciones. La información almacenada en los dispositivos podría verse comprometida, el dispositivo podría ser falsificado, o rastreado, comprometiendo la privacidad de su propietario (usuario).

La criptografía es una herramienta que ha sido utilizada por varios años para proveer servicios de seguridad informática como son la confidencialidad, autenticación, integridad y no repudio. Los algoritmos criptográficos de llave pública se basan en operaciones matemáticas como son la multiplicación y exponenciación en campos finitos $\mathbb{GF}(p)$. La mayoría de estos algoritmos criptográficos han sido desarrollados para lograr la máxima seguridad posible y pocas veces toman en cuenta en que tipo de dispositivos se van a implementar, por ejemplo, si se implementarán en dispositivos con recursos limitados.

En esta tesis se desarrollaron arquitecturas *hardware* compactas para la operación de multiplicación y exponenciación en el campo finito $\mathbb{GF}(p)$ con aplicaciones en criptografía ligera de llave pública (RSA, ElGammal, DSA, Diffie-Hellman). Se seleccionaron los dispositivos FPGAs como plataforma de experimentación debido a que ofrecen gran flexibilidad como herramientas de prototipado.

La arquitectura presentada para exponenciación modular a pesar de utilizar muy poca área, realiza una exponenciación modular en un tiempo razonablemente mejor que las implementaciones en procesadores de propósito general (software). Por lo tanto, esta arquitectura puede ser utilizada como bloque de construcción de esquemas de cifrado de llave pública (cifrado, firmas digitales, intercambio de llaves) que permitan proveer servicios de seguridad informática en sistemas embebidos, donde los recursos computacionales son generalmente reducidos.

Abstract

At present the interconnection of many devices is more common than ever. More and more devices are connected to the Internet for a range of applications, e.g. medical, military, scientific, etc. In the same way that communication among small devices with limited resources increases, security risks in those communications also increase. Information stored in those devices could be compromised, or falsified, leading to risk user's privacy.

Cryptography has been used for several years to provide computer security services such as confidentiality, authentication, integrity and non-repudiation. Public key cryptographic algorithms are based on mathematical operations such as multiplication and exponentiation in finite fields $GF(p)$. Public key cryptographic algorithms are developed to achieve maximum performance, therefore fast hardware implementations are provided at the expense of resources and/or power consumption. However, this approach cannot be applied if the targeting devices are small and with limited computational resources.

In this thesis, compact hardware architectures are developed for multiplication and exponentiation in finite field $GF(p)$ with applications in lightweight public key cryptography (RSA, ELGamal, DSA, Diffie-Hellman). Moreover, FPGAs devices are used as experimental platform to evaluate the proposed hardware architectures, taking advantage of their great flexibility as prototyping tool.

The architectures presented for multiplication and modular exponentiation, perform arithmetic operations in a reasonable better time than implementations on general purpose processors (software), despite using very little area. Therefore, these architectures can be used as basic building blocks for creating cryptographic schemes (encryption, digital signatures, key exchange) to provide information security services in embedded systems, which are characterized by having reduced computational resources.

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

Índice de algoritmos

Capítulo 1

Introducción

1.1. Motivación

Las tecnologías de la información han penetrado ampliamente en las actividades del día a día de las personas. El uso de estas tecnologías es una de las principales tendencias de la sociedad actual. La vida de una persona promedio no puede ser imaginada sin varios *gadgets*[1]. En la actualidad una gran cantidad de hogares utilizan dispositivos con un sistema operativo embebido (además de las computadoras personales habituales), que se pueden conectar a Internet e incluso se pueden interconectar en una red inalámbrica. En todas partes la gente está rodeada de una gran variedad de terminales, lectores, sensores, etc. [51], por ejemplo PDAs (*personal digital assistant* o asistente personal digital), identificación por radiofrecuencia (RFID, por sus siglas en inglés), sensores inalámbricos o WSN (por sus siglas en inglés *wireless sensor network*), tarjetas inteligentes, dispositivos para el cuidado de la salud, entre otros.

Algunos de estos sensores, tal como los RFID, no cuentan con una interfaz de usuario, operan con una batería embebida, además de que las medidas de seguridad en estos dispositivos están mal diseñadas o simplemente no existen, por ejemplo, en [65] se presentan diversos ataques a los protocolos de RFID.

Una red de sensores inalámbricos (WSN) consta de pequeños dispositivos con limitados recursos computacionales y de energía que utilizan sensores para monitorear condiciones físicas o ambientales [57]. Las redes de sensores inalámbricas se han integrado con otras ciencias como son la medicina, biología, minería, etc., al igual que en aplicaciones tecnológicas militares y civiles.

El desarrollo masivo de dispositivos ubicuos promete muchos beneficios como la reducción de los costos logísticos, una mayor granularidad de procesos, optimización

[1] Un gadget es un dispositivo que tiene un propósito y una función específica, generalmente de pequeñas proporciones, práctico y a la vez novedoso.

de las cadenas de suministro o servicios basados en localización, entre otros. Por ejemplo, la tecnología de RFID se cree que es la tecnología que permitirá el Internet de las cosas o *Internet of Things* [52]. Hay que tener en cuenta las restricciones que estos dispositivos tienen, por ejemplo, un RFID de bajo costo, puede tener entre 1000 y 10000 GE (gateway equals, compuertas equivalentes) y para los componentes de seguridad solo podría disponer del 20 % de estos GE [52].

La computación ubicua [68] se refiere a la integración de la informática en el entorno de la persona, por lo general con dispositivos invisibles al usuario. A pesar de los beneficios que estos dispositivos pueden traer, también existen algunos riesgos en la computación ubicua: muchas aplicaciones son sensibles a la seguridad, como son los sensores inalámbricos para aplicaciones militares, financieras o automotrices. Con el esparcimiento del cómputo embebido, la seguridad es un reto muy importante, por que el daño potencial de ataques maliciosos también se incrementa.

Algunos de los principales problemas en computación ubicua son:

- Alcanzar suficiente seguridad.
- Prevenir que la computación ubicua llegue a convertirse en vigilancia ubicua.

La solución de éstos problemas es lo que determinará si la computación ubicua tendrá éxito o no [58].

Un factor preocupante es que los dispositivos son generalmente utilizados en un entorno no controlado, esto quiere decir que su principal uso es en un territorio hostil, es decir, en algunos casos un adversario podría tener acceso físico o secuestrar el dispositivo. A esto se suma todo el campo de los ataques físicos posibles. En particular están los ataques llamados *side-chanel-attack* tales como SPA (*Simple power analysis*) y DPA (*Differential power analysis*)) [2]. Esta expansión de las tecnologías inteligentes hace que surjan nuevos problemas en la seguridad de los datos [51].

Hay que resaltar que una mejora en la eficiencia para realizar las operaciones aritméticas en campos finitos lleva a tener una mejora en la eficiencia de los algoritmos criptográficos. Así también, una mejora en la eficiencia de los algoritmos criptográficos repercute en una mayor eficiencia en los protocolos de seguridad informática, que a su vez proporciona una mejor eficiencia en las aplicaciones que requieren de servicios de seguridad, ver Figura 1.1. Por lo tanto, es deseable mejorar la eficiencia en las operaciones aritméticas para proveer seguridad en la capa más alta, aplicaciones, y una

[2]SPA y DPA son ataques de análisis de potencia, en los cuales los atacantes estudian el consumo de energía de un dispositivo *hardware* dedicado para criptografía con el fin de extraer de forma no invasiva las claves criptográficas y otros datos secretos.

forma de lograrlo es implementar estas operaciones que sirven de base en arquitecturas hardware.

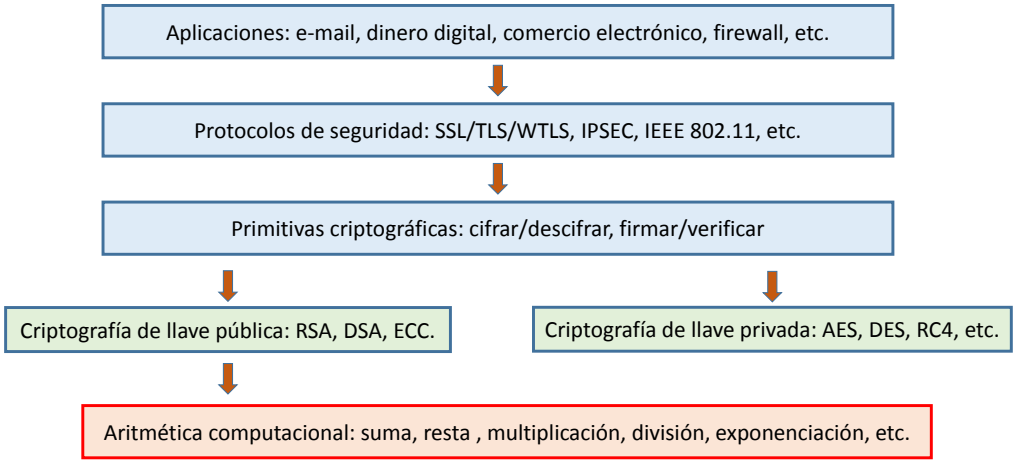


Figura 1.1: Modelo en capas para la seguridad en aplicaciones informáticas.

1.2. Planteamiento del problema

El funcionamiento de los algoritmos de criptografía de llave pública se basa fuertemente en operaciones aritméticas en campos finitos. Estas operaciones son en general consideradas costosas ya que se realizan muchas operaciones con números grandes de 160-4096 bits. Las implementaciones en software requieren múltiples instrucciones y ciclos de reloj para ejecutar las operaciones parciales, incrementando la energía total consumida. Ademas, los algoritmos criptográficos demandan altas capacidades de procesamiento. Implementaciones con un conjunto típico de instrucciones del microcontrolador o microprocesador, conduce a gran tamaño del código y alto consumo de potencia. Debido a esto, los algoritmos criptográficos de llave pública son ineficientemente implementados en software, lo que ha motivado la creación de arquitecturas *hardware* dedicadas [12,44,61,71]. Sin embargo, los algoritmos tradicionales de criptografía no son adecuados para dispositivos que cuentan con recursos limitados.

El objetivo de esta tesis es contribuir al estado del arte sobre *hardware* dedicado para criptografía ligera de llave pública. Específicamente, el problema que se aborda en esta tesis es el de diseñar arquitecturas *hardware* eficientes que ejecuten las operaciones aritméticas de bajo nivel (multiplicación y exponenciación en campos finitos) bajo las

restricciones de diseño de los sistemas embebidos, donde se requiere un consumo de energía bajo y uso de pocos recursos de área. Estas arquitecturas *hardware* permitirían acelerar el tiempo de ejecución de los esquemas criptográficos de llave pública (cifrado y firma digital), los cuales se requieren para proveer servicios de seguridad informática en las aplicaciones que se ejecutan sobre este tipo de sistemas. Aunque en la literatura existen diversos trabajos sobre arquitecturas e implementaciones en *hardware* de multiplicadores y exponenciadores en campos finitos (ver una descripción de estos en las secciones 3.1 y 3.2 del estado del arte), esos trabajos tienen como principal objetivo de diseño obtener los tiempos de ejecución más bajos, sin importar los recursos *hardware* necesarios (espacio físico). Sin embargo, esta premisa no es aplicable en el contexto de aplicaciones para sistemas embebidos que cuentan con capacidades de cómputo reducidas. En estos sistemas, el uso de mayor área de un módulo *hardware* implica mayor costo y sobre todo un mayor consumo energía. De ahí la necesidad de construir arquitecturas eficientes que puedan establecer un balance entre desempeño y recursos *hardware* usados (y por tanto, consumo de energía).

1.3. Objetivos

1.3.1. Objetivo general

- Crear arquitecturas *hardware* compactas y eficientes, especializadas para multiplicación y exponenciación en campos finitos $\mathbb{GF}(p)$, que sirvan como aceleradores *hardware* para implementar esquemas de criptografía ligera de llave pública en sistemas con recursos computacionales restringidos, como sistemas embebidos.

1.3.2. Objetivos específicos

1. Determinar los algoritmos de aritmética en campos finitos $\mathbb{GF}(p)$ que permitan una implementación *hardware* compacta.
2. Diseñar los módulos aritméticos como bloques básicos para dar soporte a la implementación *hardware* de aritmética en campos finitos $\mathbb{GF}(p)$.
3. Definir estrategias que permitan integrar módulos de bajo nivel para crear arquitecturas compactas y eficientes para aritmética en campos finitos $\mathbb{GF}(p)$.
4. Evaluar el desempeño de arquitecturas *hardware* compactas para aritmética en campos finitos $\mathbb{GF}(p)$ en esquemas de cifrado y firma digital en criptografía de llave pública.

1.4. Contribuciones y resultados alcanzados

Las principales contribuciones de esta tesis son las siguientes:

1. Modificación del algoritmo de multiplicación Montgomery iterativo que permite el diseño de una arquitectura hardware más compacta, que opera en un contexto donde se requieren múltiples operaciones de multiplicación consecutivas, tal como la exponencicación en $\mathbb{GF}(p)$.

2. Dos arquitecturas *hardware* compactas, una para la multiplicación Montgomery y otra para la exponenciación modular en el campo primo $\mathbb{GF}(p)$.

3. Un codiseño *hardware-software* que puede servir como base de desarrollo de esquemas de cifrado de llave pública (cifrado, firma digital, protocolo de intercambio de llaves), donde la operación más demandante, la exponenciación que usa múltiples multiplicaciones en $\mathbb{GF}(p)$, se ejecuta en un módulo hardware dedicado y el resto del algoritmo criptográfico se ejecuta en un microprocesador convencional.

Las arquitecturas *hardware* propuestas se describieron en el lenguaje VHDL y fueron sintetizadas para los FPGAs Spartan 3, Spartan 6, Virtex 5 y Virtex 7. Estas arquitecturas obtuvieron los mejores resultados en cuanto a área, esto es, son las más compactas reportadas en la literatura. El área utilizada por estas arquitecturas para un operando de 1024 bits es de 300, 83, 141 y 76 *slices* para los FPGAs Spartan 3, Spartan 6, Virtex 5 y Virtex 7 respectivamente, siendo en la mayoría de los casos de 5 a 20 veces más pequeñas que las reportadas en la literatura. Por lo que es altamente viable que estas arquitecturas puedan ser usadas como módulos de construcción básicos de esquemas de cifrado y firma digital en sistemas embebidos, para aplicaciones de cómputo ubicuo y móvil.

1.5. Metodología

La metodología de esta investigación consisten en:

- Estudio y análisis de algoritmos de multiplicación y exponenciación en el campo finito $\mathbb{GF}(p)$ mediante modelos software que permiten también generar vectores de pruebas.

- Bajo un enfoque iterativo e incremental, creación de bloques básicos que permiten crear arquitecturas *hardware* compactas para multiplicación y exponenciación en el campo finito $\mathbb{GF}(p)$.
- Desarrollo de arquitecturas *hardware* compactas para multiplicación y exponenciación en el campo finito $\mathbb{GF}(p)$, mediante un enfoque modular, empleando técnicas de pipeline y explotando el paralelismo.
- Creación de modelos de la arquitecturas propuestas con lenguajes de descripción de *hardware* y validar los diseños en simulación aplicando vectores de pruebas.
- Evaluación de las arquitecturas propuestas, implementando las arquitecturas *hardware* en *hardware* reconfigurable (FPGAs), aplicando el flujo de diseño que se muestra en la Figura 1.2.
- Análisis, documentación y publicación los resultados obtenidos.

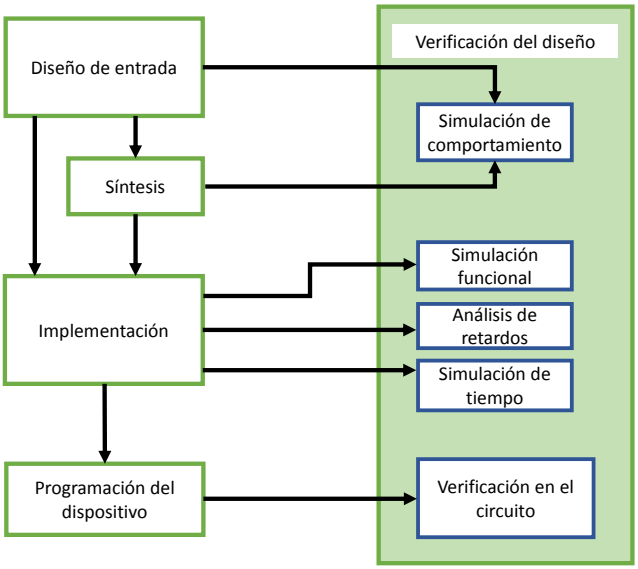


Figura 1.2: Flujo de diseño en *hardware* reconfigurable (FPGAs).

1.6. Organización de tesis

Esta tesis se organiza de la siguiente manera: en el siguiente capítulo se presenta la base matemática para la criptografía de llave pública, así como los esquemas criptográficos basados en aritmética modular (multiplicación y exponenciación). En el

capítulo 3 se presentan los trabajos más representativos para las implementaciones de multiplicación y exponenciación modular en *hardware* reconfigurable, FPGAs. En el capítulo 4 se muestra el diseño de las arquitecturas propuestas para multiplicación y exponenciación en $\mathbb{GF}(p)$, tanto en software como *hardware*. El capítulo 5 muestra los resultados de implementación de las arquitecturas propuestas en esta tesis en tecnología FPGA. Por último, en el capítulo 6 se resume este trabajo y las aportaciones de esta tesis. Las conclusiones y la dirección del trabajo futuro se presentan al final del capítulo 6.

Capítulo 2

Marco Teórico

En este capítulo se presenta una introducción de criptografía y criptografía ligera, posteriormente temas como el algoritmo de Euclides, aritmética modular, campos finitos de la forma $\mathbb{GF}(p)$ son presentados como base matemática para el entendimiento de la aritmética para criptografía de llave pública. Se presentan los algoritmos para multiplicación modular y exponenciación modular más representativos y usados en aplicaciones prácticas. Así también, se presentan los esquemas criptográficos que usan multiplicación y exponenciación en $\mathbb{GF}(p)$, tal como RSA, DH, DSA y ElGammal. Gran parte de este capítulo, en especial las secciones 2.3-2.8, se basa en [20, 24, 59]. Un lector familiarizado con los campos finitos podría saltar la mayor parte del contenido de este capítulo, sin embargo se le recomienda leer las secciones 2.8 y 2.9 las cuales tratan sobre algoritmos de multiplicación y exponenciación en el campo primo $\mathbb{GF}(p)$.

2.1. Seguridad informática

En los últimos años, hemos escuchado sobre algunos ejemplos de violación de seguridad informática, en particular desde la aparición de Internet. Información muy valiosa es a menudo el blanco principal de algunos criminales, quienes a menudo buscan robar información de las tarjetas de crédito, cuentas de banco, nombres de usuario, contraseñas e información corporativa. Otros ejemplos son simplemente pérdida de información, y que ésta no esté disponible en el momento en que se necesita.

Hay algunos fundamentos en la seguridad de la información que permiten entender mejor cuándo la información está libre de riesgo, daño o pérdida [63].

Confidencialidad: Mantiene la información libre de peligro de ser expuesta a partes no autorizadas.

Autenticación: Asegura que las partes involucradas en una transacción son quienes dicen que son. También permite saber cual es el origen de los datos.

Integridad: Permite mantener la información libre de alteraciones no autorizadas. Incluso si la información no necesita ser secreta, como con la confidencialidad, es muy importante saber si la información es correcta. La alteración de los datos involucra el borrado, la inserción o la sustitución de información.

No repudio: Evita que una entidad niegue haber estado involucrada en una acción.

Estos son los requerimientos de seguridad informática que se necesitan en prácticamente cualquier sistema informático.

Estos principios son más difíciles de satisfacer cuando se habla de dispositivos móviles y sistemas embebidos. Desafortunadamente, muchas soluciones desarrolladas para seguridad de sistemas informáticos, como son redes de computadoras o bases de datos, no son aplicables o suficientes para la seguridad en los sistemas embebidos. Por ejemplo, en muchas aplicaciones de sistemas móviles, la comunicación debe mantenerse en un nivel bajo, no se pueden transferir una gran cantidad de información, dada la naturaleza móvil de los dispositivos.

Como es de esperarse, los sistemas embebidos y los dispositivos móviles implican nuevos retos en cuanto a la seguridad informática. Es importante señalar que no hay una sola amenaza contra los sistemas de computación ubicua. Más bien, debido a la naturaleza extremadamente diversa de aplicaciones embebidas, hay una amplia gama de daños que un atacante puede hacer.

La seguridad informática en sistemas embebidos y dispositivos móviles es necesaria debido a los siguientes puntos [50]:

Riesgo potencial: Debido a la estrecha relación con el entorno físico, el riesgo involucrado en sistemas embebidos puede ser mucho mayor que el riesgo en las aplicaciones informáticas convencionales. Por ejemplo, comprometer un sistema de frenos de automóviles puede tener mucho más consecuencias que comprometer un disco duro.

Financiero: Existe un incremento en el número de las aplicaciones de cómputo pervasivo que involucra aspectos financieros, como son contenido de entretenimiento digital en casa y dispositivos móviles, servicios basados en ubicación, tarjetas inteligentes con funciones de monedero electrónico, etc.

Nuevos modelos de negocios: La televisión de pago es uno de los ejemplos establecidos de un sistema integrado con los requisitos de alta seguridad con el fin de

proteger un modelo de negocio. Así también, activación de funciones por tiempo limitado requerirán soluciones de seguridad sofisticadas.

Privacidad: Esto es ya una preocupación en los sistemas informáticos convencionales. En la computación ubicua a menudo hay un vínculo íntimo entre el usuario y el dispositivo. La privacidad incluye varios aspectos como la divulgación de la ubicación de un usuario o de su comportamiento.

Fiablidad: En muchas aplicaciones pervasivas, las manipulaciones puedes dañar la fiabilidad de un producto. Por ejemplo, las actualizaciones de software no autorizadas o maliciosas.

2.2. Criptografía

En la era actual, la necesidad de proteger información es más importante que nunca. La comunicación segura de la información sensible ya no solo le concierne a instituciones militares o gubernamentales sino también al sector empresarial, público y privado. El intercambio de información sensible sobre Internet como son transacciones bancarias, números de tarjetas de crédito y servicios de telecomunicaciones son prácticas comunes [20].

La criptografía puede usarse para proveer servicios de seguridad informática, tal como la confidencialidad, integridad, autenticación y no repudio [23]. La palabra criptografía proviene del Griego *kryptós* que significa oculto y *gráphein* que significa escribir. Ésta también se define como el arte de escribir con clave secreta o de un modo enigmático. La criptografía se encarga, por lo general con una llave secreta, de transformar texto legible en ilegible. De esta manera, un mensaje puede ser enviado sin la preocupación de que éste sea leído en caso de ser interceptado. La única persona que podrá leer el mensaje es aquella que cuente con la llave que se usó en el proceso de transformación. Al proceso de hacer un mensaje ininteligible se le llama cifrar, mientras que al proceso inverso se le llama descifrar. Una técnica conocida como criptoanálisis se enfoca en obtener el mensaje original a través de un mensaje cifrado sin la necesidad de conocer la llave. Las raíces de la criptografía pueden trazarse desde tiempos ancestrales, es casi tan antigua como la escritura.

La criptografía se divide en dos categorías importantes: criptografía de llave pública y de llave privada. Ambas categorías juegan un papel importante en las aplicaciones

Tabla 2.1: Algoritmos criptográficos más populares.

Tipo de Criptografía	Algoritmos
Criptografía de llave pública	RSA [54], ECC [30], ElGamal [19], DSA [46]
Criptografía de llave privada	AES [13], DES [60], Triple DES, Blowfish [56], Serpent [4]

de criptografía moderna [20]. Ejemplos de algoritmos criptográficos de llave pública y privada ampliamente usados en la actualidad se mencionan en la Tabla 2.1.

Criptografía simétrica y asimétrica. La criptografía de llave privada o simétrica consiste en utilizar una misma llave entre el emisor y el receptor para cifrar y descifrar información, ver Figura 2.1. Por lo tanto, el intercambio de esta llave secreta en un principio es un problema en este tipo de cifrado. Una de las principales ventajas de este tipo de criptografía es que es mucho más barata de implementar y consume menos recursos y poder computacional que la criptografía de llave pública. La criptografía de llave pública (en inglés public key cryptography) o asimétrica, usa un par de llaves para cada usuario y a través de ellas se realizan las operaciones de cifrado y descifrado, ver Figura 2.2. Una llave de este par es pública y puede ser conocida por cualquiera, la otra llave es privada y el propietario debe guardarla de modo que nadie tenga acceso a ella. La criptografía de llave pública permite proveer servicios de seguridad informática, de autenticación y no-repudio a través de firmas digitales, así como el intercambio seguro de llaves criptográficas a través de los sobres digitales.

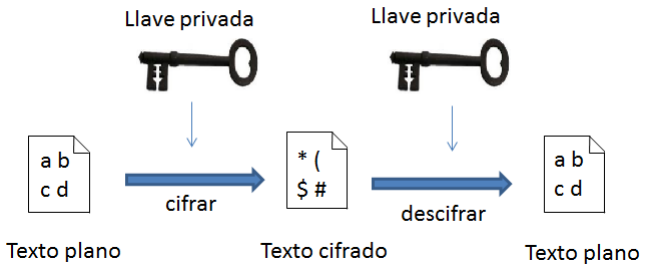


Figura 2.1: Criptografía de llave privada

2.2.1. Criptografía de llave pública

Si dos entidades, por decir Alice y Bob, requieren intercambiar un mensaje usando un cifrador simétrico, deben de ponerse de acuerdo antes de qué llave secreta k van a

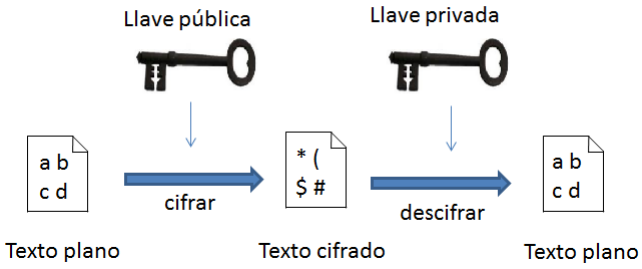


Figura 2.2: Criptografía de llave pública

utilizar. Esto es perfecto si ellos tienen la oportunidad de reunirse en secreto o si pueden comunicarse a través de un canal seguro. Pero, ¿si ellos no tiene esta oportunidad y si cualquier comunicación entre ellos es monitoreada por un adversario Eve?. ¿Es posible para Alice y Bob intercambiar una llave secreta bajo estas condiciones?. Aquí es donde aparece la brillante idea de Diffie y Hellman [16] que bajo ciertas hipótesis, esto es posible. La criptografía de llave pública es una solución a este problema, y forma una de las partes más interesantes de la criptografía moderna basada en álgebra abstracta [24].

Cifradores asimétricos

Los cifradores asimétricos usan una llave k de un espacio de llaves K para cifrar un mensaje en texto plano m de un conjunto de posibles mensajes M, y el resultado del proceso de cifrado es un texto cifrado c tomado de un espacio de posibles mensajes cifrados C. La llave k es en realidad un par de llaves, denotado como $k = (k_{priv}, k_{pub})$, k_{priv} es llamada la *llave privada* y k_{pub} la *llave pública*. Para cada llave pública k_{pub} hay una función de cifrado e correspondiente

$$e_{k_{pub}} : M \to C$$

y para cada llave privada k_{priv} hay una función de descifrado d correspondiente

$$d_{k_{priv}} : C \to M$$

Estas funciones tienen la propiedad que si el par (k_{priv}, k_{pub}) están en el espacio de llaves K, entonces la Ecuación 2.1 se cumple.

$$d_{k_{priv}}(e_{k_{pub}}(m)) = m, \text{ para todo } m \in M \tag{2.1}$$

Si un cifrador asimétrico es seguro, debe ser difícil para un adversario calcular la función de descifrado $d_{k_{priv}}(c)$, incluso si se conoce la llave pública k_{pub}. Bajo esta suposición, Alice puede enviar k_{pub} a Bob usando un canal de comunicación inseguro, y Bob puede enviarle el texto cifrado $e_{k_{pub}}(m)$ a Alice sin preocuparse de que Eve sea capaz de obtener m. Para un descifrado fácil, es necesario conocer la llave privada k_{priv}, y presumiblemente Alice es la única persona que la conoce. Informalmente se puede decir que "fácil" significa que se puede realizar el cálculo en menos de un segundo con cualquier computadora de escritorio y "difícil" que incluso con el mayor poder computacional posible se requieren años para realizar el cálculo.

Si (K, M, C, e, d) es un buen sistema criptográfico de llave pública, debe tener las siguiente propiedades:

- Para cualquier $k \in K$ y un texto plano $m \in M$, debe de ser fácil calcular el texto cifrado $e_k(m)$.
- Para cualquier $k \in K$ y un texto cifrado $c \in C$, debe de ser fácil calcular el texto plano $d_k(c)$.
- Dado uno o más textos cifrados $c_1, c_2, \ldots, c_n \in C$ cifrados usando una llave $k \in K$, debe de ser muy difícil calcular alguno de los textos planos correspondientes $d_k(c_1), \ldots, d_k(c_n)$, sin el conocimiento de k.
- Sean $(m_1, c_1), (m_2, c_2), \ldots, (m_n, c_n)$ uno o más pares de texto plano y su correspondiente texto cifrado, debe ser difícil descifrar cualquier texto cifrado c que no esta en la lista sin el conocimiento de k. Aunque esta propiedad es deseable, es mucho más difícil de alcanzar.

La primera contribución importante de Diffie y Hellman fue la definición de un Criptosistema de Llave Pública (PKC) y sus componentes asociados: funciones de un solo sentido e información de trampa o *trapdoor*. Una función de un solo sentido es una función que es fácil de calcular, pero su inversa no lo es. Los criptosistemas, conjunto de procedimientos que garantizan la seguridad de la información utilizando técnicas criptográficas, de llave pública están construidos usando funciones de un solo sentido que tienen información de trampa. La información de trampa es una pieza auxiliar de información que permite que el inverso de una función sea sencilla de calcular.

Se puede decir que la llave privada k_{priv} es *información de trampa* para la función $e_{k_{pub}}$, por que sin la información de trampa es muy difícil calcular la función inversa de $e_{k_{pub}}$, pero con la función de trampa es relativamente fácil calcular la inversa.

Tabla 2.2: Problemas matemáticos propuestos como funciones de un solo sentido.

Problema	Descripción	Criptosistema que lo utilizan
Factorización de enteros grandes [54]	Dado un número n, encontrar sus factores primos.	RSA, Rabin-Williams
Logaritmo discreto [16]	Dado un número primo p, y los número g y h, encontrar x que satisfaga $h = g^x \mod p$.	ElGamal, Diffie y Hellman, DSA
Logaritmo discreto para curvas elípticas [30, 40]	Dada una curva elíptica E y los puntos P y Q de la curva E, encontrar x que satisfaga $Q = xP$.	ECC, ECDSA.

Puede parecer sorprendente que a pesar de años de investigación, aun no se conoce la existencia de funciones de un sentido. Una prueba de la existencia de una función de un solo sentido podría simultáneamente resolver el famoso problema de $P = NP$ en teoría de la complejidad. Varios candidatos para este tipo de funciones han sido propuestos, y algunos de ellos son usados por los algoritmos de criptografía de llave pública modernos. La seguridad de los criptosistema reside en la suposición de que invertir este tipo de funciones es un problema difícil [24].

Dentro de los problemas considerados difíciles de resolver que son utilizados en criptografía de llave pública los tres más utilizados se muestran en la Tabla 2.2. Para el problema de factorización de enteros grandes y el problema de logaritmo discreto, los mejores algoritmos conocidos para resolverlos tiene una complejidad sub-exponencial, esto es, el tiempo necesario para resolver este problema es más grande que cualquier polinomio, pero sigue siendo menor que un algoritmo exponencial. Para el problema del logaritmo discreto en curvas elípticas, el mejor algoritmo conocido para resolverlo tiene una complejidad totalmente exponencial. Debido a esto, el criptosistema de curvas elípticas, que está basado en el problema del logaritmo discreto de curvas elípticas, ofrece el mismo nivel de seguridad que otros criptosistema como son RSA y ElGamal, pero con una longitud más corta en la llave utilizada.

En los esquemas criptográficos RSA, ElGamal, DSA y Diffie-Hellman una de las operaciones más importantes que se realiza es la exponenciación modular de números muy grandes. Esta operación se usa tanto para cifrar un mensaje como para descifrarlo. Los tamaños típicos de los números utilizados en estos criptosistemas son de 512 a 4096 bits.

Por ejemplo, RSA genera un texto cifrado C a partir de un texto plano M realizando una exponenciación modular:

$$C = M^e \mod n$$

De la misma forma para poder descifrar un texto cifrado C y obtener un texto plano M se realiza otra exponenciación modular:

$$M = C^d \mod n$$

Se darán más detalles de estas operaciones en el Marco Teórico en el Capítulo 2.

2.2.2. Criptografía ligera

También conocida como criptografía de bajo peso o *lightweight cryptography*, es una rama de la criptografía moderna, y se enfoca en el estudio de algoritmos criptográficos pensados para dispositivos con bajos o extremadamente bajos recursos. La criptografía ligera no determina criterios estrictos para clasificar un algoritmo criptográfico como de bajo peso, pero las características generales de los algoritmos criptográficos de bajo peso son requerimientos extremadamente bajos en cuanto a los recursos de los dispositivos móviles o embebidos [51], por ejemplo:

- Tamaño requerido para la implementación *hardware*;
- Poder computacional de microprocesadores o microcontroladores;
- Memoria de acceso aleatorio (RAM);
- Memoria de solo-lectura (ROM).

Los algoritmos de criptografía ligera están diseñados para minimizar el consumo de recursos de una implementación *hardware* como son: área, poder computacional, y consumo de energía [35]. Cada diseñador de criptografía de bajo peso debe lidiar con el balance entre seguridad, costo y desempeño. Es generalmente fácil optimizar cualquier par de estos tres objetivos, seguridad y costo, seguridad y desempeño, o costo y desempeño; sin embargo, es muy difícil optimizar los tres objetivos a la vez [18]. Las primeras primitivas exploradas fueron los cifradores de flujo (cifradores simétricos) [3, 14, 49], seguida de un amplio rango de cifradores de bloque [9, 15, 25] y posteriormente se ha expandido a la introducción de nuevas funciones hash de bajo costo [36].

En términos generales, hay tres enfoques para proporcionar servicios de criptografía en aplicaciones ligeras [52]:

1. Optimización de implementaciones de bajo costo para algoritmos estandarizados y confiables.
2. Modificar ligeramente un cifrador de confianza.
3. Diseñar nuevos sistemas de cifrado con el enfoque de tener bajos costos de implementación en *hardware*.

Mientras que las primitivas de criptografía ligera de llave pública están en demanda para los protocolos de manejo de llaves en pequeños dispositivos, los recursos requeridos para criptografía de llave pública son mucho más grandes que para criptografía de llave privada. Hasta el momento, no hay primitivas prometedoras que cumplan con suficiente seguridad y propiedades de peso ligero en comparación con las primitivas convencionales tales como RSA y ECC. Algunos esquemas de cifrado de clave pública (por ejemplo ECC) se pueden implementar con relativamente poco espacio, pero no pueden ejecutarse en un tiempo razonable [28].

2.3. Introducción a los campos finitos y sus aplicaciones en criptografía.

Los campos finitos juegan un rol importante en el área de la criptografía. Un gran número de algoritmos criptográficos basa fuertemente su seguridad en varias de las propiedades de estos campos. Los campos finitos se estudian principalmente en teoría de números y álgebra abstracta. En el nivel más básico, como se menciona en [24], la teoría de números es el estudio del conjunto de los números enteros

$$\cdots, -3, -2, -1, 0, 1, 2, 3, \cdots$$

denotado por el símbolo $\mathbb{Z}$. Los enteros son un ejemplo de un anillo, *ring*, debido a que si a y b son enteros estos pueden ser sumados $a + b$, restados $a - b$ y multiplicados $a \times b$ en la forma usual de la aritmética (ley conmutativa, asociativa, distributiva, etc.) y en cualquier caso el resultado será otro número entero. Esta propiedad de que al aplicar una operación a un par de elementos de un conjunto y mantenerse dentro del conjunto es una característica de los anillos, más adelante se dará la definición formal de grupos, anillos y campos.

Para mantenerse dentro de los enteros no se debe aplicar la operación de división a todos ellos. Debido a que si se dividen dos enteros no siempre se obtendrá otro entero como resultado. Por ejemplo, al dividir 5 entre 3 el resultado no es un entero, sin embargo, 4 entre 2 está dentro del conjunto $\mathbb{Z}$.

Esto conlleva a un concepto muy importante dentro de la teoría de números conocido cono divisibilidad.

Definición 1 *Sean* a *y* b *enteros con* $b \neq 0$. *Se dice que* b divide *a* a, *o que* a *es divisible por* b, *si existe un entero* c *tal que*

$$a = bc.$$

Se escribe $b \mid a$ para indicar que b divide a a, y si b no divide a a, entonces se escribe $b \nmid a$.

Hay algunas propiedades elementales sobre divisibilidad, algunas de ellas se listan a continuación.

Proposición 1 *Sean* $a, b, c \in \mathbb{Z}$.

(a) Si $a \mid 1$, *entonces* $a = \pm 1$.

(b) Cualquier $b \neq 0$ *divide a* 0.

(c) Si $a \mid b$ *y* $b \mid c$, *entonces* $a \mid c$.

(d) Si $a \mid b$ *y* $b \mid a$, *entonces* $a = \pm b$.

(e) Si $a \mid b$ *y* $a \mid c$, *entonces* $a \mid (b + c)$ *y* $a \mid (b - c)$.

Definición 2 *Un* común divisor *de dos enteros* a *y* b *es un entero positivo* d *que divide a ambos. Así el máximo común divisor de* a *y* b *es el entero más grande* d *que cumple* $d \mid a$ *y* $d \mid b$. *El* máximo común divisor *también se denota como* $\gcd(a, b)$ *(por sus siglas en inglés greatest common divisor). Si* a *y* b *son ambos* 0, *entonces el* máximo común divisor $\gcd(a, b)$ *no está definido.*

Por ejemplo,

$$\gcd(24, 16) = 8$$

y

$$\gcd(559, 301) = 43$$

Una de las formas para mostrar que el resultados del máximo común divisor de 559 y 301 es 43 puede ser listar todos los divisores positivos de 559 y de 301, y el máximo común divisor es aquel número mayor que aparezca en ambas listas.

Por ejemplo,

$$\text{Divisores de } 559 = \{1, 13, 43, 559\}.$$
$$\text{Divisores de } 301 = \{1, 7, 43, 301\}.$$

Observando las dos listas se concluye que el máximo común divisor de 559 y 301 es 43, aparte de ser éste el único común divisor mayor que uno. Para números pequeños parece ser un poco tedioso este método, lo cual significa que para números muy grandes, encontrar el *gcd* con este método resulta ser ineficiente. Por lo tanto se necesita un mejor algoritmo que permita calcular el *gcd*.

Una propiedad interesante acerca del *gcd* es que:

$$\gcd(a, b) = \gcd(a, -b) = \gcd(-a, b) = \gcd(-a, -b)$$
$$\text{En general } \gcd(a, b) = \gcd(|a|, |b|).$$

Así también, debido a que todo valor entero $a \neq 0$ divide a 0

$$\gcd(a, 0) = |a|$$

Definición 3 *Dos enteros* a *y* b *son primos relativos si su único común divisor positivo es 1. Esto es lo mismo que decir que* a *y* b *son primos relativos si* $\gcd(a, b) = 1$.

Las definiciones y proposiciones de esta sección fueron tomadas de [24].

2.4. Algoritmo de Euclides

En [24] se da la siguiente definición del algoritmo de la división y el algoritmo de Euclides.

Definición 4 *(Algoritmo de la División) Sean* a *y* b *enteros positivos. Entonces* a *dividido por* b *tiene un cociente* q *y un residuo* r*, esto significa que*

$$a = b \cdot q + r \qquad \text{con } 0 \leqslant r < b$$

Los valores de q y r están completamente determinados por los valores de a y b y son únicos.

Si ahora se desea calcular el máximo común divisor de dos números enteros a y b, se puede empezar por dividir a entre b para obtener los valores de q y r con

$$a = b \cdot q + r \qquad \text{con } 0 \leqslant r < b$$

Si ahora el número d es el común divisor de a y b es claro que entonces d debe dividir también a r (por la Proposición 1). De la misma forma si d es común divisor de b y r, entonces d también es divisor de a.

$$gcd(a, b) = gcd(b, r).$$

Repitiendo este proceso, dividiendo b entre r para obtener otro cociente y otro residuo, tenemos

$$b = r \cdot q' + r' \qquad 0 \leqslant r' < r.$$

Siguiendo el mismo razonamiento, tenemos que

$$gcd(b, r) = gcd(r, r').$$

Continuando con este proceso, el residuo será cada vez más pequeño en cada iteración, hasta que eventualmente llega a tener un valor de 0, y en este punto final, el valor de $gcd(s, 0) = s$ es igual al gcd de a y b. Este proceso, es conocido como el *algoritmo de Euclides*.

Teorema 1 *(Algoritmo de Euclides) Sean* a *y* b *enteros positivos con* $a \geqslant b$. *El Algoritmo 1 calcula* $gcd(a, b)$ *en un número finito de iteraciones.*

Algoritmo 1: Algoritmo de Euclides

Entrada: enteros positivos a y b

Salida: $gcd(a, b)$

1: $r_0 \leftarrow a$;
2: $r_1 \leftarrow b$;
3: $i \leftarrow 1$;
4: Dividir r_{i-1} entre r_i para obtener un cociente q_i y un residuo r_{i+1},

$$r_{i-1} = r_i \cdot q_i + r_{i+1} \qquad \text{con } 0 \leqslant r_{i+1} \leqslant r_i.$$

5: Si el residuo $r_{i+1} = 0$, entonces $r_i = gcd(a, b)$ y el algoritmo termina
6: De otra forma, si $r_{i+1} \geqslant 0$ entonces
7: $i \leftarrow i + 1$
8: ir a paso 4

devolver r_i

En el Algoritmo 2 se muestra un algoritmo recursivo para calcular el algoritmo de Euclides, este algoritmo hace uso del operador módulo $\bmod$, el cual aplicará poco más adelante. Por el momento es suficiente saber que el operador $\bmod$ da como resultado el residuo en una división de números enteros.

Algoritmo 2: Algoritmo de Euclides Recursivo, EUCLID(a,b)

Entrada: enteros positivos a y b
Salida: $gcd(a, b)$

```
1: if b = 0 then
2:    devolver a
3: else
4:    devolver EUCLID(b, a mod b)
5: end if
```

Otra característica muy sencilla, pero con muchas consecuencias importantes y aplicaciones acerca del máximo común divisor de dos números enteros a, b, es que siempre es posible escribir el máximo común divisor como una combinación lineal, lo cual es conocido como el algoritmo de Euclides Extendido.

Teorema 2 *(Algoritmo de Euclides Extendido). Sean* a *y* b *enteros positivos. Entonces la ecuación*

$$au + bv = gcd(a, b)$$

siempre tiene soluciones dentro de los enteros para los números u *y* v*. Y si* (u_0, v_0) *es una solución, entonces cualquier solución* u, v *tiene la forma*

$$u = u_0 + \frac{b \cdot k}{gcd(a,b)} \qquad y \qquad v = v_0 - \frac{a \cdot k}{gcd(a,b)} \qquad \text{para algún } k \in \mathbb{Z}.$$

2.5. Aritmética modular

La aritmética modular es un área muy importante dentro de las matemáticas. También es conocida como aritmética de reloj, debido a que después de alcanzar un valor máximo el siguiente valor da la vuelta, vuelve a empezar como si fuera un reloj. Esto permite ecuaciones un poco extrañas. Por ejemplo,

$$9 + 8 \equiv 5 \mod 12, \qquad 2 - 3 \equiv 11 \mod 12 \qquad \text{y} \qquad 12 + 1 \equiv 1 \mod 12$$

Aunque estas operaciones parecen extrañas, son correctas utilizando aritmética modular o de reloj. Lo que sucede es que en este tipo de aritmética después del 12, en vez de seguir el 13, como normalmente lo utilizaríamos, sigue el número 1, al igual que en un reloj. Este ejemplo a pesar de ser un ejemplo sencillo, es muy ilustrativo para mostrar lo que comúnmente se conoce como *congruencias*. La teoría de congruencias es

un método muy poderoso en teoría de números basado en la idea de aritmética de reloj.

Definición 5 *Sea* $m \geqslant 1$ *un número entero. Se dice que los enteros* a *y* b *son congruentes módulo* m *si la diferencia* $a - b$ *es divisible por* m*. Y se escribe.*

$$a \equiv b \mod m$$

para indicar que a y b son congruentes módulo m. El número m es llamado el *módulo*.

Los ejemplos de aritmética modular anteriores pueden ser escritos de la siguiente forma

$$9+8=17\equiv 5 \mod 12, \qquad 2-3=-1\equiv 11 \mod 12 \qquad \text{y} \qquad 12+1=13\equiv 1 \mod 12$$

Así también,

$$19 \not\equiv 6 \mod 11, \qquad \text{ya que } 19-6=13 \text{ no divide a } 11.$$

La razón de que las congruencias sean muy útiles es que tienen varias propiedades muy parecidas a las igualdades, como se muestra en la Proposición 2.

Proposición 2 *Sea* $m \geqslant 1$ *un número entero.*

(a) Si $a_1 \equiv a_2 \mod m$ *y* $b_1 \equiv b_2 \mod m$*, entonces*

$$a_1 \pm b_1 \equiv a_2 \pm b_2 \mod m \quad y \quad a_1 \cdot b_1 \equiv a_2 \cdot b_2 \mod m$$

(b) Sea a *un número entero. Entonces*

$$a \cdot b \equiv 1 \mod m \text{ para algún número entero } b \text{ si y solo si } \gcd(a,m)=1.$$

Si ese número entero b *existe, entonces decimos que ese entero* b *es el inverso (multiplicativo) de* a *módulo* m*, y cualquier par de inversos multiplicativos de* $a \mod m$ *son congruentes módulo* m*.*

Un ejemplo de inverso multiplicativo, se tiene $m = 13$ y $a = 4$. Si se calcula el $\gcd(a,m)$ se obtiene el resultado igual a 1. Por lo tanto el inverso de 4 módulo 13 existe. El inverso de 4 módulo 13 es 10 ya que

$$\gcd(4,13)=1$$
$$4 \cdot 10 = 40 \equiv 1 \mod 13.$$

Otro ejemplo, tomemos $m = 449$ y $a = 180$

$$gcd(449, 180) = 1$$
$$180 \cdot 227 = 40860 \equiv 1 \mod 449.$$

De esta forma se puede trabajar con fracciones ya que $\frac{1}{a} = a^{-1}$. Lo cual se puede interpretar de la siguiente forma: a^{-1} es el inverso (multiplicativo) de $a \mod m$.

Y así se puede trabajar con fracciones $\frac{a}{b}$ módulo m siempre y cuando el denominador sea primo relativo de m, o lo que es igual $gcd(b, m) = 1$. Por ejemplo, se puede calcular $\frac{11}{10} \mod 17$ primero calculando $10^{-1} \equiv 12 \mod 17$, entonces

$$\frac{11}{10} = 11 \cdot 10^{-1} \equiv 11 \cdot 12 \equiv 132 \equiv 13 \mod 17$$

En el primer ejemplo fue trivial calcular el inverso multiplicativo de $4 \mod 13$ por prueba y error, ya que es un número pequeño. En el segundo ejemplo con un poco de paciencia y esfuerzo también se puede calcular. Pero para números muy grandes, por ejemplo mayores de 100 bits, parece ser un reto calcularlo por prueba y error.

En la Proposición 2 se estableció que $a \cdot b \equiv 1 \mod m$ para algún entero b si y solo si $gcd(a, m) = 1$. Y en el teorema 2 se dijo que $au + bv = gcd(a, b)$ siempre tiene soluciones para los números u y v dentro de los números enteros.

Por lo tanto, si tenemos que

$$gcd(a, m) = 1$$

entonces, el teorema 2 dice que existe un par de números enteros u y v que satisfacen

$$au + mv = 1 = gcd(a, m).$$

Esto significa que

$$au - 1 = -mv$$

es divisible por m, entonces

$$au \equiv 1 \mod m$$

En otras palabras $b = u$ lo que es lo mismo $u = b = a^{-1} \mod m$. Existen varios algoritmos que calculan los números u y v basados en el algoritmo de Euclides.

Las definiciones y proposiciones de esta sección fueron tomadas de [24].

2.6. Grupos, anillos y campos

Los grupos, anillos y campos juegan un papel muy importante en el área de las matemáticas conocida como álgebra abstracta o álgebra moderna, así como en el campo de la criptografía. Los grupos, anillos y campos se definen formalmente en [59] de la siguiente manera:

Definición 6 *Un **Grupo** G, a veces denotado por* $\{G, \bullet\}$*, es un conjunto de elementos con una operación binaria* $\bullet$ *que asocia cada par de elementos* (a, b) *de* G *con un elemento* $(a \bullet b)$ *en* G*, donde los siguientes axiomas se cumplen:*

A1. Cerradura: Si a y b pertenecen a G , entonces $a \bullet b$ también pertenece a G.

A2. Asociatividad: $a \bullet (b \bullet c) = (a \bullet b) \bullet c)$ para todos a, b, c en G.

A3. Existencia del elemento identidad: Debe existir un elemento e en G tal que $a \bullet e = e \bullet a = a$ para toda a en G.

A4. Existencia del elemento inverso: Para cada a en G, debe existir un elemento a' en G tal que $a \bullet a' = a' \bullet a = e$.

El operador $\bullet$ es genérico y puede referirse a cualquier operación aritmética, como la suma, multiplicación, o alguna otra.

Si un **grupo** tiene un número finito de elementos se dice que es un **grupo finito**, y el orden del grupo es el número de elementos que el grupo contiene. De otra forma el grupo es un grupo infinito.

Definición 7 *Un **grupo abeliano** es un grupo que respeta los axiomas **A1-A4**, y además*

A5. Conmutatividad: $a \bullet b = b \bullet a$ para toda a, b pertenecientes a G.

Un ejemplo de un **grupo abeliano** es el conjunto de los números enteros con la operación suma. Por ejemplo, el elemento identidad es 0, ya que $a + 0 = a$, el inverso de a es $-a$, ya que $a - a = 0$, y la resta se define como $a - b = a + (-b)$.

Definición 8 *Un **Anillo o Ring**, a veces denotado por* $\{R, +, *\}$ *es un conjunto de elementos con dos operaciones binarias, llamadas adición* $+$ *y multiplicación* $*$*, donde para todo* a, b, c *en* R*, los siguientes axiomas se cumplen.*

A1-A5. R es un grupo abeliano con respecto a la operación $+$. Esto es, R satisface los axiomas A1 a A5 para la operación $+$ con el elemento identidad 0 y el inverso de a es $-a$.

M1. Cerradura bajo la multiplicación: Si a y b pertenecen a R , entonces $a * b$ también pertenece a R.

M2. Asociatividad bajo la multiplicación: $a * (b * c) = (a * b) * c$ para toda a, b, c pertenecientes a R.

M3. Ley distributiva: $(a + b) * c = a * c + b * c$ para toda a, b, c pertenecientes a R.

Definición 9 *Un **anillo conmutativo** es un anillo que además satisface el siguiente axioma.*

M4. Conmutatividad de multiplicación: $a * b = b * a$ para todo $a, b \in R$

El conjunto de los enteros (negativos, positivos y 0) bajo las operaciones usuales de la suma y la multiplicación son un ejemplo de un anillo conmutativo.

Definición 10 *Un **dominio integral** es un anillo conmutativo que además obedece los siguientes axiomas*

M5. Identidad de multiplicación: Existe un elemento 1 en R tal que $a * 1 = 1 * a = a$ para todo a perteneciente a R.

M6. No hay divisores de 0: Si a, b pertenecen a R y $a * b = 0$, entonces $a = 0$ o $b = 0$.

El conjunto de los números enteros bajo las operaciones usuales de suma y multiplicación, son un dominio integral.

Definición 11 *Un **Campo**, a veces denotado por $\{F, +, *\}$ es un conjunto de elementos F con dos operaciones binarias, llamadas* suma $+$ y multiplicación $*$*, tal que para todo a, b, c en F los siguientes axiomas se cumplen.*

A1-M6. F es un dominio integral; esto es, F satisface los axiomas A1-A5 y M1-M6.

M7. Inverso multiplicativo: Para cada a en F, excepto para 0, existe un elemento a^{-1} en F que satisface $a * a^{-1} = a^{-1} * a = 1$.

En esencia, un campo es un conjunto en el cual se pueden hacer sumas, restas, multiplicación y división, sin que el resultado de la operación sea un número que no pertenezca al conjunto. La división está definida como $a/b = a * (b^{-1})$. Algunos ejemplos de **campos** son el conjunto de los número reales y el conjunto de los números complejos. Note que el conjunto de los números enteros no es un campo ya que al realizar una división no siempre el resultado se mantiene dentro del conjunto de los números enteros.

2.7. Campos finitos de la forma $\mathbb{GF}(p)$

Un *campo finito* o *Galois field* es indicado como $\mathbb{GF}(q = p^m)$, con un número de elemento q, donde p es un número primo y m es un número entero positivo. En criptografía, uno de los campos finitos ampliamente usado es el campo finito con $m = 1$, llamado campo primo. Con $p = 2$, se tiene el otro campo finito usado en criptografía, llamado campo binario, $\mathbb{GF}(2^m)$. Este campo se conoce también como campo de extensión y suele también denotarse como $\mathbb{F}_{2^m}$. En este trabajo se utiliza el campo primo $\mathbb{GF}(p)$ debido a que es una operación base para algoritmos muy utilizados, como es RSA, intercambio de llaves Diffie-Hellman y DSA.

En [59] se da la siguiente definición de campo primo $\mathbb{GF}(p)$:

Definición 12 *Para un número primo p dado, se puede definir el campo finito de orden p, $\mathbb{GF}(p)$, como el conjunto de enteros $\{0, 1, 2, 3, \ldots, p-1\}$ junto con las operaciones aritméticas módulo p.*

Una observación muy interesante mostrada anteriormente, es que para todo elemento a, $a \times b \mod m \equiv 1$ para algún entero b si y solo si $gcd(a, m) = 1$. Para el caso de $\mathbb{GF}(p)$ como todas los operaciones se realizan módulo un número primo p, entonces para cualquier número a, $a \times b \equiv 1 \mod p$ siempre tiene una solución para el número b, ya que p es primo y $gcd(a, p) = 1$. Por lo tanto, cualquier elemento $a \in \mathbb{GF}(p)$ tiene un inverso multiplicativo a^{-1}.

Cabe mencionar que el inverso aditivo de a es aquel número $-a$ tal que $a + (-a) \mod p = 0$, y que el inverso multiplicativo módulo p de un número a es el número a^{-1} tal que $a \cdot a^{-1} \equiv 1 \mod p$.

En las Tablas 2.3, 2.4 y 2.5 se muestra un ejemplo de las operaciones de multiplicación, suma y los inversos multiplicativos y aditivos de un campo finito $\mathbb{GF}(7)$.

En los siguientes capítulos se mostrarán algunos otros ejemplos de este tipo de campos.

Para un estudio más detallado sobre campos finitos, se recomienda consultar [59]. En [24] se dan más detalles sobre las propiedades de los campos finitos, así como de las matemáticas para criptografía de manera más formal, explicando algunas demostraciones de las propiedades y teoremas aquí presentados. Así también en [22] se profundiza en los grupos algebraicos y los principales algoritmos en teoría de números desde un punto de vista criptográfico.

Tabla 2.3: Multiplicación en el campo primo $\mathbb{GF}(7)$.

x	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Tabla 2.4: Suma en el campo primo $\mathbb{GF}(7)$

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

Tabla 2.5: Inverso aditivo y multiplicativo en el campo primo $\mathbb{GF}(7)$

w	$-w$	w^{-1}
0	0	-
1	6	1
2	5	4
3	4	5
4	3	2
5	2	3
6	1	6

2.8. Multiplicación en el campo primo $\mathbb{GF}(p)$.

La operación de multiplicación modular en el campo primo $\mathbb{GF}(p)$ se define en [20] de la siguiente manera:

$$A \times B \mod n$$

para los enteros A, B y n. Es común asumir que A y B sean enteros positivos con $0 \leqslant A, B < n$. Para implementar esta operación existen diversos enfoques, así como varios algoritmos, entre los cuales se encuentran:

- Multiplicar y después dividir.
- Los pasos de multiplicación y reducción se ejecutan de forma intercalada.
- Multiplicación Montgomery.

En el enfoque de multiplicar y después dividir las operaciones se realizan de forma separada, esto es, primero se calcula el producto:

$$P' = A \times B.$$

Cabe mencionar que A y B son números de k-bits cada uno, entonces el producto P' sera un número de 2k-bits. Posteriormente el paso de reducción es ejecutado dividiendo P' por n; en este caso, solo interesa el residuo, no el cociente. Debido a que los pasos de multiplicación y división están separados, se puede utilizar cualquier algoritmo para multiplicar, así como cualquier algoritmo que permita acelerar el paso de reducción.

El algoritmo **clásico de multiplicación** es el que la mayoría de las personas aprenden en el colegio. Esto es, sean A y B dos números de *s-dígitos* expresados en base W:

$$A = (A_{s-1}A_{s-2}\dots A_0) = \sum_{j=0}^{s-1} A_i W^i,$$

$$B = (B_{s-1}B_{s-2}\dots B_0) = \sum_{j=0}^{s-1} B_i W^i,$$

donde los dígitos de A y B están en el rango $[0, W-1]$ En general, W puede ser cualquier número entero positivo. En cambio, para el caso de implementaciones en *hardware* por lo general se utiliza $W = 2^w$ donde w es el tamaño de la palabra o la granularidad del dispositivo, por ejemplo $w = 4$ o $w = 5$. El algoritmo clásico de multiplicar A y B genera productos parciales al multiplicar un dígito de B por el número A, los cuales posteriormente se suman para obtener el número P', de *2s-dígitos*. Sea P_{ij} el par (carry,save) obtenido al calcular el producto $A_i \times B_j$. Por ejemplo, cuando $W = 10$, y $A_i = 7$ y $B_j = 8$, entonces $P'_{ij} = (5, 6)$. Los pares P_{ij} pueden ser ordenados en una tabla de la siguiente manera:

				A_3	A_2	A_1	A_0
$\times$				B_3	B_2	B_1	B_0
				P'_{03}	P'_{02}	P'_{01}	P'_{00}
			P'_{13}	P'_{12}	P'_{11}	P'_{10}	
		P'_{23}	P'_{22}	P'_{21}	P'_{20}		
$+$	P'_{33}	P'_{32}	P'_{31}	P'_{30}			
P'_7	P'_6	P'_5	P'_4	P'_3	P'_2	P'_1	P'_0

El último registro indica la suma total de los productos parciales, y representa el producto como un número de *2s-bits*

Para el caso de ahorrar espacio, se puede utilizar un solo registro P' para almacenar los productos parciales. El valor inicial del registro P' es cero; entonces, se toma un dígito de B y se multiplica por A, y se suma al producto parcial P'. La variable P' que almacena los productos parciales al final de recorrer todos los dígitos de B tendrá

almacenado el producto final $A \times B$. En el Algoritmo 3 se muestra una implementación del algoritmo estándar para la multiplicación.

Algoritmo 3: Algoritmo estándar para la multiplicación de números enteros [20]

Entrada: enteros positivos A y B
Salida: $P' = A \times B$

1: $P'_i \leftarrow 0$ para todo $i = 0, 1, 2 \ldots 2s - 1$;
2: **for** $i = o$ to $s - 1$ **do**
3: $C \leftarrow 0$;
4: **for** $j = o$ to $s - 1$ **do**
5: $(C, S) \leftarrow P'_{ij} + A_j + B_i + C$;
6: $P'_{i+j} \leftarrow S$;
7: **end for**
8: $P'_{i+s} \leftarrow C$;
9: **end for**
devolver $(P'_{2s-1} P'_{2s-2} \ldots P'_0)$

2.8.1. Algoritmo de Karatsuba

Fue introducido por Karatsuba y Ofman para multiplicación de enteros muy grandes [27]. El algoritmo reduce el tiempo de ejecución de multiplicar dos números enteros grandes de *N-dígitos* a $\mathcal{O}(N^{\log_2 3})$, comparado con el tiempo de ejecución de $\mathcal{O}(N^2)$ del algoritmo clásico. Una multiplicación es reemplazada por 3 sumas o restas. El algoritmo de Karatsuba reduce el número de multiplicaciones tomando ventaja de dos productos parciales intermedios.

Para ilustrar el algoritmo de Karatsuba se puede observar el siguiente ejemplo, sean X y Y dos números enteros sin signo de *2k-bits*. Se pueden dividir de la siguiente manera:

$$X = 2^k \times X_1 + X_0$$
$$Y = 2^k \times Y_1 + Y_0$$

Cuatro multiplicaciones de *k-bits* y tres sumas son requeridas para calcular el producto $X \times Y$ en el algoritmo clásico de multiplicación. Ver Ecuación 2.2.

$$\begin{aligned} X \times Y &= 2^{2k} \times M_\beta + 2^k \times M_\gamma + M_\alpha \\ M_\beta &= X_1 \times Y_1 \\ M_\gamma &= X_0 \times Y_1 + X_1 \times Y_0 \\ M\alpha &= X_0 \times Y_0 \end{aligned} \tag{2.2}$$

Karatsuba observó que el término medio M_γ puede ser calculado usando los productos parciales M_α y M_β. La Ecuación 2.3 muestra el cálculo del término medio $M\gamma$ en el método de Karatsuba.

$$\begin{aligned} M_\gamma &= X_0 \times Y_1 + X_1 \times Y_0 \\ &= X_1 \times Y_1 + X_0 \times Y_0 + (X_0 - X_1)(Y_1 - Y_0) \\ &= M_\beta + M_\alpha + K_\alpha \times K_\beta \end{aligned} \tag{2.3}$$

Por lo tanto, por las Ecuaciones 2.2 y 2.3, el producto $X \times Y$ es calculado con tres *k-bits* multiplicaciones, cuatro sumas y dos restas. Entonces, el algoritmo de Karatsuba ahorra un cuarto del costo de multiplicaciones intercambiándolas por algunas sumas y restas.

2.8.2. Multiplicación y reducción intercalada

Sean A_i y B_i los i-ésimos bits de los números enteros A y B respectivamente cada uno de *k-bits*. El producto P' puede ser escrito de la siguiente forma:

$$\begin{aligned} P' &= A \times B = A \times \sum_{i=0}^{k-1} B_i 2^i = \sum_{i=0}^{k-1} (A \times B_i) 2^i \\ &= 2(\cdots 2(2(0 + A \times B_{k-1}) + A \times B_{k-2}) + \cdots) + A \times B_0 \end{aligned} \tag{2.4}$$

La Ecuación 2.4 lleva al algoritmo de multiplicación *shift-add*. El algoritmo completo lo puede observar en el Algoritmo 4. Cabe mencionar que la reducción módulo n se le aplica en cada paso al producto parcial.

Asumiendo que $A, B, P < n$, entonces

Algoritmo 4: Algoritmo de multiplicación y reducción intercalada.

Entrada: Enteros positivos $A = \sum_{i=0}^{k-1} A_i 2^i$ y $B = \sum_{i=0}^{k-1} B_i 2^i$, $n = \sum_{i=0}^{k-1} n_i 2^i$.
Salida: $P = A \times B \mod n$

1: $P \leftarrow 0$;
2: **for** $i = 0$ to $k-1$ **do**
3: $\quad P \leftarrow 2P + A \times B_{k-1-i}$;
4: $\quad P \leftarrow P \mod n$
5: **end for**
devolver P

$$P \leftarrow 2P + A \times B_j \leqslant 2(n-1) + (n-1) = 3n - 3.$$

El nuevo P está en el rango $0 \leqslant P \leqslant 3n-3$, entonces necesitarán máximo 2 restas para reducir P al rango $0 \leqslant P < n$. Se podría utilizar el Algoritmo 5 para que P tenga un valor nuevamente en el rango deseado.

Los detalles de la multiplicación y reducción intercalada pueden ser consultados en [5].

Algoritmo 5: Reducción módulo P.

Entrada: Enteros positivos P, n.
Salida: $P = P \mod n$

1: $P' \leftarrow P - n$;
2: **if** $P' \geqslant 0$ **then**
3: $\quad P \leftarrow P'$
4: **end if**
5: $P' \leftarrow P - n$;
6: **if** $P' \geqslant 0$ **then**
7: $\quad P \leftarrow P'$
8: **end if**
devolver P

2.8.3. Multiplicación Montgomery

El algoritmo Montgomery fue propuesto por el matemático Peter L. Montgomery en 1985 [42]. Desde entonces ha sido utilizado por muchos como base para diferentes implementaciones de multiplicación modular.

Sea el módulo n un número entero de k bits, esto es $2^{k-1} \leqslant n < 2^k$. El algoritmo de multiplicación Montgomery requiere que se utilice un factor r tal que r y n sean primos relativos, esto es $gcd(n, r) = 1$. Es común utilizar $r = 2^k$, por lo tanto se requiere que $gcd(n, 2^k) = 1$ y esto se logra tomando n un valor impar.

Para realizar una multiplicación Montgomery de dos números a, b es necesario que estos números se transformen a una representación diferente, es decir, hay que transformar a a y a b al dominio de Montgomery. Estas transformaciones se realizan de la siguiente forma:

$$a' = ar \mod p$$

$$b' = br \mod p$$

Dado que $gcd(r, p) = 1$ se pueden calcular los números r^{-1} y p' tal que $rr^{-1} + pp' = 1$ con $0 < r^{-1} < p$ y $0 < p' < r$. Para esto se puede utilizar el algoritmo extendido de Euclides. Para realizar una reducción Montgomery se requiere que estos valores estén previamente calculados. Mientras que a es un número normal, a' lo llamaremos un número Montgomery.

La reducción modular Montgomery recibe como entrada un número z' y devuelve como resultado $z'r^{-1} \mod P$. La reducción Montgomery se muestra en el Algoritmo 6. El producto Montgomery se puede calcular con el Algoritmo 7.

Algoritmo 6: Algoritmo para reducción Montgomery: REDUCE(z')

Entrada: z'
Salida: $z' \times r^{-1} \mod p$
1: $q \leftarrow (z' \mod r) \times p' \mod r$
2: $a \leftarrow (z' + qp)/r$
3: **if** $a > p$ **then**
4: $\quad a \leftarrow a - p$
5: **end if**
6: **devolver** a

Para realizar una multiplicación Montgomery de 2 números en el dominio de Montgomery a' y b' se puede utilizar la función MM de la siguiente forma:

$$z' = MM(a', b')$$

se puede observar que $(a'b') = (arbr) \mod p = (abr^2) \mod p$, al aplicar la función REDUCE, Algoritmo 6, a este valor se obtiene $(abr^2r^{-1}) \equiv (abr) \mod p$

Algoritmo 7: Algoritmo de multiplicación Montgomery: MM(a', b')

Entrada: $a' = a \times r \mod p$

Entrada: $b' = b \times r \mod p$

Salida: $a' \times b' \times r^{-1} \mod p$

1: $t' \leftarrow a' \times b'$

2: $z' \leftarrow \text{REDUCE}(t')$

3: **devolver** z'

lo cual da otro número en el dominio de Montgomery. Esta función MM también se puede utilizar para hacer la transformación de un número normal al dominio de Montgomery, por ejemplo:

$$a' = MM(a, r^2)$$

De igual forma se puede convertir un número en el dominio de Montgomery al dominio normal de la siguiente forma:

$$a = MM(a', 1)$$

Teniendo estas transformaciones disponibles, se puede ejecutar toda la multiplicación modular de 2 números a y b con la siguiente secuencia de instrucciones.

$$\begin{aligned} a' &= MM(a, r^2) \\ b' &= MM(b, r^2) \\ z' &= MM(a', b') \\ z &= MM(z', 1) = a \times b \mod p \end{aligned}$$

Realizar la multiplicación Montgomery es más simple y rápido que calcular $a \times b \mod p$. Lo cual involucra una división por p. Sin embargo, las conversiones involucradas de un número normal al dominio de Montgomery, la conversión de un número Montgomery a un número normal, y el cálculo del valor p' involucran tareas que consumen tiempo. Por lo tanto, utilizar el método Montgomery para realizar una sola multiplicación modular es una mala idea, ya que realizar todos los cálculos parece ser un método más costoso que utilizar un método tradicional. Es más conveniente utilizar el algoritmo de Montgomery cuando se requiere hacer varias multiplicaciones con el mismo valor p para el módulo. Por ejemplo, cuando se necesite implementar una exponenciación modular, este es el caso del criptositema RSA.

2.9. Exponenciación en el campo primo $\mathbb{GF}(p)$.

El problema de la exponenciación modular está definido como la operación $g^e \mod p$ dados los números enteros g, e y p. Normalmente g es un número entero positivo con $0 \leqslant g < p$ y e es un número entero positivo arbitrario. El método simple de calcular $g^e \mod p$ es por multiplicaciones sucesivas de g, esto es:

$$g_1 \equiv g \mod p,\ g_2 \equiv g * g_1 \mod p\ g_3 \equiv g * g_2 \mod p, \ldots$$

Es claro que $g_e \equiv g^e \mod p$. Pero si e es muy grande, este algoritmo es completamente impráctico. Por ejemplo, si $e \approx 2^{1024}$, un número de aproximadamente 1024 bits. El algoritmo simple podría tomar tanto tiempo como la edad aproximada del universo.

Otro factor importante a tomar en cuenta es que no se debe calcular primero g^e y después realizar una división para obtener $C = (g^e) \mod p$. Los resultados de las multiplicaciones parciales deben de ser reducidos en cada paso. Esto es así, debido a que la cantidad de bits para almacenar el número g^e es enorme, suponiendo que g y e sean de 1024 bits cada uno. No se tiene forma de almacenar este valor en ninguna computadora, por mucha memoria que ésta tenga.

Claramente si esta operación es muy útil, es necesario tener un algoritmo que la calcule de una manera más eficiente. Existen varias estrategias para implementar esta operación, por ejemplo: la estrategia binaria, Montgomery Powering Ladder, la técnica de ventana deslizante, etc.

2.9.1. Reducción del número de multiplicaciones en la exponenciación

En este método se requiere de la expansión binaria del exponente, la cual se puede partir en grupos de d-bits cada uno. Entonces se precalcula y se obtiene $M^w \mod n$ solo para estos w que aparecen en la expansión binaria. Considerando el exponente k de 16 bits y $d = 4$

$$\underline{1011}\ \underline{0011}\ \underline{0111}\ \underline{1000}$$

lo cual implica que solo se necesita calcular $M^w \mod n$ para los valores de $w = 3, 7, 8, 11$. Los cálculos para $w = 3, 7, 8, 11$ pueden ser obtenidos secuencialmente de la siguiente forma:

$$\begin{aligned} M^2 &= M \times M \\ M^3 &= M^2 \times M \\ M^4 &= M^2 \times M^2 \\ M^7 &= M^3 \times M^4 \\ M^8 &= M^4 \times M^4 \\ M^{11} &= M^8 \times M^3 \end{aligned}$$

lo cual requiere solo de 6 multiplicaciones. El método *m-ario* que requiere calcular todos los valores posibles de w requiere de $16 - 2 = 14$ multiplicaciones. Mientras que en el mejor de los casos podría no requerirse ningún cálculo, y utilizar solo el valor de M. Este caso es cuando el exponente tiene una forma:

$$\underline{0001}\ \underline{0001}\ \underline{0001}\ \underline{0001}$$

esto solo pasa en casos muy raros.

En general, se necesita calcular $M^w \mod n$ para todos $w = w_0, w_1, \ldots, w_{p-1}$. Si la expansión del conjunto $\{w_i | i = 0, 1, \ldots, p-1\}$ son los valores $\{2, 3, \ldots, 2^d - 1\}$, entonces, no hay ningún ahorro al utilizar este método. Sin embargo, si la expansión es un subconjunto de los valores $\{2, 3, \ldots, 2^d - 1\}$, entonces se pueden ahorrar algunas multiplicaciones al utilizar este método. Un algoritmo para calcular cualquiera de los valores del exponente p es llamado algoritmo cadena de adición vectorial (*vectorial addition chain*) y en el caso de $p = 1$ algoritmo cadena de adición (*addition chain*). Desafortunadamente, el problema de obtener una cadena de adición de longitud minimal es un problema *NP-completo* [17].

2.9.2. Método de ventana deslizante

Los métodos *m-ary* descomponen los bits del exponente en palabras de *d-bits*. La probabilidad de que una palabra de longitud d sea cero es igual a 2^d, asumiendo que los bits 0 y 1 tienen la misma probabilidad de aparecer en la cadena. En el Algoritmo 8 se muestra el algoritmo de ventana deslizante. En la línea 6 del algoritmo se evita la multiplicación cuando se encuentra una ventana igual a cero. Si d crece demasiado, la probabilidad de que se tenga que realizar una multiplicación también se hace más grande. Por lo tanto, el número de multiplicaciones se incrementa en medida en que d disminuye. El algoritmo de ventana deslizante provee un compromiso, permitiendo

Algoritmo 8: Método de ventana deslizante [31].

Entrada: M, e, n

Salida: $C = M^e \mod n$

1: Calcular y almacenar $M^w \mod n$ para todos $w = 3, 5, 7, \ldots, 2^d - 1$.
2: Descomponer el exponente e en ventanas de ceros y no ceros F_i de longitud $L(F_i)$ para $i = 0, 1, 2, \ldots, p - 1$.
3: $C \leftarrow M^{F_{k-1}} \mod n$
4: **for** $i = p - 2$ **downto** 0 **do**
5: $\quad C \leftarrow C^{2^{L(F_i)}} \mod n$
6: $\quad$ **if** $F_i \neq 0$ **then**
7: $\quad\quad C \leftarrow C \times M^{F_i} \mod n$
8: $\quad$ **end if**
9: **end for**
devolver C;

palabras de cero y no-cero de longitud variable; esta estrategia permite incrementar el número promedio de palabras igual a cero, usando valores relativamente largos de d.

El algoritmo de exponenciación de ventana deslizante primero descompone el exponente e en palabras (ventanas) F_i cero y no-cero de longitud $L(F_i)$. El número de ventanas p puede no ser igual al número de bits del exponente dividido entre d. En general, no se requiere que el tamaño de las ventanas sea de la misma longitud. Se debe tomar d como el tamaño de la ventana más larga, esto es $d = max(L(F_i))$ para $i = 0, 1, \ldots, k - 1$.

Existen algunas variantes del algoritmo de ventana deslizante, como es el algoritmo de Ventana no-cero de longitud constante. Para ver más a detalle estos algoritmos, se recomienda al lector consultar [31].

2.9.3. Estrategia binaria

En el método binario se escanean los bits del exponente, existen dos variantes del escaneo del exponente conocidos cono *left to right* y *right to left* (izquierda a derecha y derecha a izquierda). Una operación de elevación al cuadrado es ejecutada en cada paso, y dependiendo de la forma de recorrer el exponente se ejecuta una multiplicación secuencial o paralela. El lector puede consultar el libro de Knuth [29] sección 4.6 para más información.

El algoritmo binario de izquierda a derecha se puede observar en el Algoritmo 9; de la misma forma, el algoritmo de derecha a izquierda puede observarse en el

Algoritmo 10.

Algoritmo 9: Exponenciación binaria de izquierda a derecha o LR (left-to-right).

Entrada: M, e, n
Salida: $C = M^e \mod n$

```
1: if e_h == 1 then
2:    C ← M;
3: else
4:    C ← 1;
5: end if
6: for i = h − 2 to 0 do
7:    C ← C × C mod n;
8:    if e_i == 1 then
9:       C ← C × M mod n;
10:   end if
11: end fordevolver C;
```

En el algoritmo de izquierda a derecha las multiplicaciones que se realizan en cada paso, esto es dentro del ciclo *for*, son totalmente secuenciales. Esto implica que estas multiplicaciones se deben calcular una después de la otra. Una ventaja de este enfoque al implementarla en *hardware*, es que se puede implementar el algoritmo con un solo multiplicador el cual se utiliza primero para calcular $C \leftarrow C \times C \mod n$, y una vez teniendo este valor se puede utilizar para calcular $C \leftarrow C \times M \mod n$.

En el algoritmo de derecha a izquierda las multiplicaciones pueden ejecutarse totalmente en paralelo, ya que el valor de una, no depende de la otra. Esto implica que se debe de tener un registro más que en el algoritmo de izquierda a derecha. Debido a que las multiplicaciones se pueden ejecutar en paralelo, se debe tomar en cuenta que se necesitan 2 multiplicadores en vez de 1. El tiempo de ejecución de este algoritmo es mejor que el algoritmo de izquierda a derecha.

Para un exponente e de *k-bits* con $e_{k-1} = 1$, el algoritmo binario requiere:

- $k - 1$ elevaciones al cuadrado, donde k es el número de bits en la expansion binaria de e
- $H(e) - 1$ multiplicaciones, donde $H(e)$ es la medida Hamming (el número de unos que hay en la expansión binaria) de e.

.

Algoritmo 10: Exponenciación binaria de derecha a izquierda o RL (right-to-left).

Entrada: M, e, n
Salida: $C = M^e \mod n$

```
1: C ← 1;
2: P ← M;
3: for i = 0 to h − 2 do
4:    if e_i == 1 then
5:       C ← C × P mod n
6:    end if
7:    P ← P × P mod n
8: end for
9: if e_{h−1} == 1 then
10:   C ← C × P mod n
11: end ifdevolver C;
```

2.9.4. Montgomery Powering Ladder

Originalmente el algoritmo Montgomery Ladder [41] fue propuesto como una forma de acelerar la multiplicación escalar en el contexto de las curvas elípticas. Este algoritmo ha sido aplicado con diferentes propósitos. Joe y Yen [26] proponen una contra medida para SPA(*simple power analysis*), el cual es un ataque de análisis de consumo de energía, basada en el algoritmo Montgomery Ladder [41], conocido como Montgomery powering ladder. Este algoritmo no tiene saltos condicionales ni operaciones de relleno. Debido a esto es resistente a ciertos tipos de ataques como SPA.

Considerando el problema original de calcular $y = g^k$ en un *grupo abeliano* $\mathbb{G}$, con entradas g y k. Sea $\sum_{i=0}^{t-1} k_i 2^i$ la expansion binaria del exponente k. El algoritmo *Montgomery Powering Ladder* depende de la siguiente observación. Primero se define:

$$L_j = \sum_{i=j}^{t-1} k_i 2^{i-j}, \qquad H_j = L_j + 1,$$

se tiene que:

$$L_j = 2L_{j+1} + k_j = L_{j+1} + H_{j+1} + k_j - 1 = 2H_{j+1} + k_j - 2. \tag{2.5}$$

y por lo tanto se obtiene

$$(L_j, H_j) = \begin{cases} (2L_{j+1}, L_{j+1} + H_{j+1}) & \text{if } k_j = 0, \\ (L_{j+1} + H_{j+1}, 2H_{j+1}) & \text{if } k_j = 1. \end{cases} \tag{2.6}$$

En cada iteración se puede utilizar un registro, sea R_0, para almacenar el valor de g^{L_i} y un segundo registro, sea R_1 es utilizado para almacenar el valor de G_{H_j}. La Ecuación 2.6 implica que:

$$(g^{L_j}, g^{H_j}) = \begin{cases} ((g^{L_{j+1}})^2, g^{L_{j+1}} \times g^{H_{j+1}}) & \text{if } k_j = 0, \\ (g^{L_{j+1}} \times g^{H_{j+1}}, (g^{H_{j+1}})^2) & \text{if } k_j = 1, \end{cases} \tag{2.7}$$

Recordando que $L_0 = k$, esto lleva a un elegante algoritmo para evaluar $y = g^k$, *Montgomery Ladder* [26]. El algoritmo *Montgomery Powering Ladder* se muestra en el Algoritmo 11.

Algoritmo 11: Algoritmo Montgomery Powering Ladder [26]

Entrada: $m, d = (d_{k-1}, \cdots, d_0)$ //exponente
Salida: $C = m^d$

```
1:  R0 ← 1;
2:  R1 ← m;
3:  for i = k − 1 downto 0 do
4:    if d_i == 1 then
5:      R0 ← R0 × R1;
6:      R1 ← R1 × R1;
7:    else
8:      R0 ← R0 × R0;
9:      R1 ← R1 × R0;
10:   end if
11: end for
    devolver R0;
```

2.10. Criptosistemas basados en exponenciación en $\mathbb{GF}(p)$.

En esta sección se presentan algunos sistemas criptográficos que tiene como su principal operación aritmética la exponenciación modular en campos finitos $\mathbb{GF}(p)$.

2.10.1. Intercambio de llaves Diffie-Hellman

Este algoritmo [16] resuelve el problema de intercambio de llaves, por lo general para ser utilizado en un criptosistema de llave privada. Alice y Bob quieren intercambiar una llave secreta para ser utilizada bajo un esquema de criptografía de llave privada pero no tienen un canal seguro por el cual comunicarse. Existe un adversario Eve que siempre está a la escucha de cualquier información que ellos intercambian. ¿Cómo es posible que Alice y Bob puedan intercambiar una llave sin que Eve pueda interceptarla?

El primer paso para Alice y Bob es escoger un número primo grande p y un número distinto de cero $g \mod p$. Entonces, Alice y Bob hacen los valores de p y g públicos, esto es, cualquier persona puede tener acceso a ellos, incluso Eve. El siguiente paso para Alice es escoger un número entero a que no debe revelar a nadie. De la misma forma Bob debe selecciona un número entero b que no debe revelar a nadie. Entonces Bob y Alice deben de utilizar sus números secretos para calcular

$$\underbrace{A \equiv g^a \mod p}_{\text{Alice calcula esto}} \quad \text{y} \quad \underbrace{B \equiv g^b \mod p}_{\text{Bob calcula esto}}.$$

Ellos intercambian sus valores calculados, Alice le manda A a Bob y Bob le manda B a Alice. Teniendo en cuenta que Eve intercepta estos valores, ella conoce los valores de A y B ya que fueron enviados por un canal inseguro. Finalmente Alice y Bob usan sus números secretos para calcular

$$\underbrace{A' \equiv B^a \mod p}_{\text{Alice calcula esto}} \quad \text{y} \quad \underbrace{B' \equiv A^b \mod p}_{\text{Bob calcula esto}}.$$

Los valores que ellos han calculado, A' y B', son los mismo ya que

$$A' \equiv B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \equiv B' \mod p \tag{2.8}$$

Este valor que ellos han calculado es la llave secreta que pueden utilizar en un criptosistema de llave privada.

Eve se encuentra ante el problema conocido como DHP (*Diffie-Hellman Problem*). El DHP es el problema de calcular el valor de $g^{ab} \mod p$ dados los valores $g^a \mod p$ y $g^b \mod p$. En la Tabla 2.6 se resumen los pasos a seguir en el algoritmo para intercambio de llaves Diffie-Hellman.

Tabla 2.6: Intercambio de llaves Diffie-Hellman.

Alice	**Bob**
Creación de parámetros públicos	
Escoger un número primo grande p y un número entero no cero $g \mod p$	
Cálculos privados	
Escoge un número entero a en secreto Calcular $A \equiv g^a \mod p$	Escoge un número entero b en secreto Calcular $B \equiv g^b \mod p$
Intercambio de valores calculados	
Alice manda A a Bob $\rightarrow A$ $B \leftarrow$ Bob manda B a Bob	
Cálculos privados	
Calcular el número $B^a \mod p$	Calcular el número $A^b \mod p$
Llave secreta	
El valor de la llave secreta es $B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \mod p$	

2.10.2. RSA

El algoritmo conocido como RSA es un sistema criptográfico de llave pública que fue desarrollado en 1977 [54]. RSA es el primer algoritmo de llave pública, así como el más utilizado. Este algoritmo sirve tanto para cifrar un mensaje como para firmar un mensaje digitalmente. La seguridad de este algoritmo radica en el problema de la factorización de números enteros.

Los mensajes se representan mediante números, y el funcionamiento se basa en el producto de dos números primos grandes elegidos al azar y mantenidos en secreto. Actualmente estos números primos son del orden de 2^{1024} ó 2^{2048}, y se prevé que su tamaño crezca con el aumento de la capacidad de cálculo de las computadoras.

Como en todo sistema de clave pública, cada usuario posee dos claves de cifrado: una pública y otra privada. Cuando se quiere enviar un mensaje, el emisor busca la clave pública del receptor, cifra su mensaje con esa clave, y una vez que el mensaje cifrado llega al receptor, éste se ocupa de descifrarlo usando su clave privada.

La seguridad de RSA depende de los siguientes puntos:

- **Configuración.** Sean p y q dos números primos grandes, sea $N = p \times q$, y sea e y c enteros.
- **Problema.** Resolver la congruencia $x^e \equiv c \mod N$ para la variable x.

- **Fácil.** Quien conoce los valores p y q, puede fácilmente resolver el problema para x.

- **Difícil.** Quien no conoce los valores de p y q, no puede encontrar fácilmente el valor de x.

- **Dicotomía.** Resolver $x^e \equiv c \mod N$ es fácil para la persona que tiene cierta información extra, pero es extremadamente difícil para todas las demás personas.

Supongamos que Bob y Alice tienen el usual problema de intercambiar información sensible sobre un canal inseguro. El algoritmo de criptografía de llave pública RSA se resume en la Tabla 2.7. La llave secreta de Bob es el par de números primos largos p y q. Su llave privada (N, e) consiste del producto $N = pq$ y un exponente para cifrar e que es primo relativo de $(p-1)(q-1)$. Alice puede tomar su texto plano y convertirlo a un número entero m entre 1 y N. Ella cifra m de acuerdo a la Ecuación 2.9.

$$c \equiv m^e \mod N. \tag{2.9}$$

El entero c es el texto cifrado, el cual se envía a Bob. Para Bob es muy sencillo resolver la congruencia $x^e \equiv c \mod N$ y recobrar el mensaje m de Alice, debido a que Bob conoce la factorización de $N = pq$. Cualquier otra persona puede interceptar el mensaje cifrado c, pero a no ser que conozca cómo factorizar N, tendrá una difícil tarea tratando de resolver $x^e \equiv c \mod N$.

La cantidad N y e que forman la llave pública de Bob son llamados, respectivamente, el *módulo* y el *exponente de cifrado*. El número d que usa Bob para descifrar los mensajes de Alice, esto es el número d que satisface:

$$ed \equiv 1 \mod (p-1)(q-1), \tag{2.10}$$

es llamado el *exponente de descifrado*. Es claro que el proceso de cifrado puede ser más eficiente si el exponente de cifrado e es más pequeño y similarmente el proceso de descifrado es más eficiente si el exponente de descifrado d es pequeño. Por supuesto, Bob no puede escoger ambos pequeños, ya que uno de ellos es seleccionado y el otro es determinado por la congruencia 2.10. Es importante tomar en cuenta algunos aspectos a la hora de implementar RSA, así como los valores seleccionados, ya que existen varios ataques al criptosistema RSA cuando éste se implementa de forma directa y sin considerar aspectos críticos de seguridad como los que previamente se han mencionado. En [6, 10, 11, 69] se describen algunos detalles de ataques a RSA.

Tabla 2.7: Criptosistema de llave pública RSA.

Bob	**Alice**
Creación de llaves	
Escoger primos p y q en secreto. Escoger exponente e con $\gcd(e,(p-1)(q-1))=1$ Publicar $N=pq \mod n$	
Cifrar	
	Escoger un texto plano m. Usar la llave pública de Bob (N,e) para calcular $c \equiv m^e \mod N$, Enviar el texto cifrado c a Bob.
Descifrar	
Calcular el número d que satisface $ed \equiv 1 \mod (p-1)(q-1)$. Calcular $m' \equiv c^d \mod N$, Entonces m' es igual al texto plano m.	

2.10.3. DSA

Una firma digital es un mecanismo criptográfico que permite al receptor de un mensaje firmado digitalmente determinar la entidad originadora de dicho mensaje, y confirmar que el mensaje no ha sido alterado desde que fue firmado por el originador. **Digital Signature Algorithm**, en español **Algoritmo de Firma Digital**, es un algoritmo como su nombre lo indica para realizar firmas digitales y no para cifrar la información. Este algoritmo fue introducido en el FIPS 186-2 Digital Signature Standard (DSS) [53]. El algoritmo DSA se resume en los siguientes pasos.

Creación de llaves

- Escoger un número primo p de L bits, donde $512 \leqslant L \leqslant 1024$ y L es divisible por 64.
- Elegir un número primo q de 160 bits, tal que $p-1 = qz$, donde z es algún número natural.
- Elegir h, donde $1 < h < p-1$ tal que $g = h^z(\mod p) > 1$.
- Elegir x de forma aleatoria, donde $1 < x < q-1$.

- Calcular $y = g^x(\mod p)$.

Los datos públicos son y, q, g y p. La llave privada es x.

Firma

- Elegir un número aleatorio k, donde $1 < k < q$.
- Calcular $r = (g^k \mod p) \mod q$.
- Calcular $s = k^{-1}(H(m) + r \times x) \mod q$, donde $H(m)$ es la función hash aplicada al mensaje m.
- La firma es el par (r, s).

Si r ó s es cero, se vuelve a repetir el procedimiento.

Verificación

- Calcular $w = (s)^{-1}(\mod q)$.
- Calcular $u1 = H(m) \times w(\mod q)$.
- Calcular $u2 = r \times w(\mod q)$.
- Calcular $v = [g^{u1} \times y^{u2} \mod p] \mod q$.
- La firma es válida si $v = r$.

2.11. Sumario

En este capítulo se presentó una pequeña introducción sobre los campos finitos, en especial sobre el campo primo $GF(p)$. Se explicó la aritmética modular en campos finitos así como algunas de sus propiedades. También se mostraron algunos algoritmos importantes como es el algoritmo de Euclides, con el cual se puede calcular el máximo común divisor de dos números enteros. Se mostraron las operaciones en campos finitos más utilizadas en varios criptosistemas, las cuales son la multiplicación y exponenciación modular. Toda esta teoría de campos finitos es la base para muchos de los criptosistemas más utilizados actualmente. Por lo cual, un estudio y entendimiento básico de los campos finitos es necesaria para poder entender varios criptosistemas de llave pública. Se mostraron algunos de los algoritmos criptográficos de llave pública que hacen uso de la multiplicación y exponenciación en el campo primo $GF(p)$ dentro de los cuales se encuentra RSA, DAS, etc.

En el siguiente capítulo se muestran algunos de los trabajos más relevantes en cuanto a la multiplicación y exponenciacion modular en campos finitos.

Capítulo 3

Estado del arte

En este capítulo se presentan los trabajos más relevantes de arquitecturas *hardware* para la multiplicación y exponenciación modular sobre el campo primo $\mathbb{GF}(p)$. Para la multiplicación modular, los trabajos presentados son arquitecturas que implementan la multiplicación Montgomery con diferentes enfoques. Para la exponenciación modular, los trabajos presentados muestran diferentes enfoques, como son: Montgomery Powering Ladder y la estrategia binaria de izquierda a derecha y de derecha a izquierda.

3.1. Algoritmos de multiplicación en $\mathbb{GF}(p)$

El algoritmo Montgomery ha sido implementado para diferentes dispositivos, tanto en *hardware* como en software. Así también, se le han ido haciendo algunas modificaciones que permiten implementaciones acorde a las necesidades. Al igual que una multiplicación se puede implementar de diferentes formas, Koc et al. [32] proponen la implementación del algoritmo Montgomery de diferentes maneras. Estos algoritmos están basados en dos factores principales, el primero es: si la multiplicación y la reducción están juntas o separadas. El segundo factor es la forma general de la multiplicación y la reducción. Una forma es escaneando el operando (Operand Scanning) y otra escaneando el producto (Product Scanning). Las cinco técnicas desarrolladas reciben los nombres de:

- Separated Operand Scanning (SOS)
- Coarsely Integrated Operand Scanning (CIOS)
- Finely Integrated Operand Scanning (FIOS)
- Finely Integrated Product Scanning (FIPS)

- Coarsely Integrated Hybrid Scanning (CIHS).

Estos algoritmos rompen la multiplicación modular en una serie iterativa de sumas y multiplicaciones de palabras. El tamaño de las palabras o dígitos puede ser escogido de forma arbitraria, idealmente que aproveche al máximo las propiedades de la tecnología a utilizar. Koc concluye que CIOS proporciona el mejor rendimiento para aplicaciones software, mostrando implementaciones para ensamblador y C bajo el Sistema Operativo Linux. En el Algoritmo 12 se muestra el algoritmo CIOS, el cual es el más eficiente en estas implementaciones.

Algoritmo 12: Multiplicación Montgomery CIOS (Coarsely Integrated Operand Scanning)

Entrada: a_j, b_j, n_j: Palabras del operando y módulo (w bit cada una), donde $a = (a_{s-1}, \ldots, a_1, a_0)$; n_0^{-1} = inverso multiplicativo de n_0

Salida: $t = a \times b \times 2^{-k} \mod n$, donde $k = \lceil \log_2 n \rceil$

1: **for** $i \leftarrow 0$ **to** $s-1$ **do**
2: $C \leftarrow 0$;
3: **for** $j \leftarrow 0$ **to** $s-1$ **do**
4: $\{C, S\} \leftarrow t_j + a_j \times b_i + C$;
5: $t_j \leftarrow S$;
6: **end for**
7: $\{C, S\} \leftarrow t_s + C$;
8: $t_s \leftarrow S$;
9: $t_{s+1} \leftarrow C$;
10: $C \leftarrow 0$;
11: $m \leftarrow t_0 \times (-n_0^{-1}) \mod 2^w$;
12: $\{C, S\} \leftarrow t_0 + n_0 \times m$;
13: **for** $j \leftarrow 1$ **to** $s-1$ **do**
14: $\{C, S\} \leftarrow t_j + n_j \times m + C$;
15: $t_{j-1} \leftarrow S$;
16: **end for**
17: $\{C, S\} \leftarrow t_s + C$;
18: $t_{s-1} \leftarrow S$
19: $t_s \leftarrow t_{s+1} + C$
20: **end for**
devolver t;

McIvor et. al [38] al estudiar estos mismos algoritmos para FPGAs, también llega a la conclusión de que CIOS provee los mejores resultados de tiempo para estos dispositivos. En su trabajo se estudian los algoritmos SOS, CIOS y FIOS, ya que estos son los más prometedores de acuerdo al trabajo de Koc [32]. Los multiplicadores 18×18 y la lógica *carry look-ahead* embebidos dentro de los FPGAs de Xilinx Virtex 2 Pro son utilizados para ejecutar las multiplicaciones y sumas requeridas por estos algoritmos.

Posteriormente, Ersin et al. [48] muestran una implementación del algoritmo CIOS Montgomery con un enfoque de ser una implementación compacta, para FPGAs pequeños. Su diseño utiliza los bloques de multiplicadores dedicados y los bloques de memoria integrados en los FPGAs como unidad de almacenamiento para los operandos. Este diseño permite ajustar el número de multiplicadores, el tamaño de las palabras utilizadas en los multiplicadores, y el número de palabras, todo esto para satisfacer los requerimientos del sistema, como son los recursos disponibles y restricciones de tiempo. La arquitectura utiliza una técnica de *pipeline* que permite operaciones concurrentes en los multiplicadores conectados. En la Figura 3.1 se muestra la arquitectura de un elemento de procesamiento o PE (*Processing Elemen*), que se utilizan para ir realizando la multiplicación y la reducción. Se puede escoger el número de PEs dependiendo de las necesidades, logrando una arquitectura final compacta o rápida, o un balance entre ambas. El número de ciclos de reloj necesarios para calcular una multiplicación modular depende del número de PEs que se han configurado.

Para evitar una larga propagación del bit de acarreo durante la etapa de adición en el algoritmo Montgomery, varias técnicas han sido propuestas como son arquitecturas de arreglos sistólicos [7, 47, 66] y *carry-save adders* (CSA) [34, 37].

En [72] se presenta una arquitectura de CSA que ejecuta la conversión del resultado de formato CSA a formato normal reutilizando lógica del multiplicador. Como consecuencia, el *datapath* del multiplicador Montgomery es muy simple. La implementación resultante muestra claramente que la arquitectura es eficiente: requiere poca área y tiene un buen rendimiento. Para convertir el formato CSA al formato tradicional se utiliza el Algoritmo 13. El algoritmo Montgomery con CSA se muestra en el Algoritmo 14

La arquitectura que implementa el algoritmo se muestra en la Figura 3.2. Como se puede observar, los beneficios de esta arquitectura es que no se necesita la propagación del *carry*, y por lo tanto es posible realizar una suma más rápido. La misma arquitectura se utiliza para hacer la conversión de CSA a una representación normal y por lo tanto se logra implementar en un área pequeña.

Algoritmo 13: Algoritmo de adición con CSA (Carry Save Adder)

Entrada: X, Y

Salida: $P = X + Y$

1: $SUM \leftarrow X$;
2: $CARRY \leftarrow Y$;
3: **while** $CARRY \neq 0$ **do do**
4: $\quad SUM, CARRY \leftarrow SUM + CARRY$;
5: **end while**
devolver $P = SUM$;

Algoritmo 14: Multiplicación Montgomery con CSA (Carry Save Adder)

Entrada: X, Y, M, con $0 \leqslant X, Y < 2M$ y $2^{n-1} < M < 2^n$

Salida: $P = X \times Y \times 2^{-(n+2)} \mod M$

1: $SUM \leftarrow 0$;
2: $CARRY \leftarrow 0$;
3: **for** $i = 0$ **to** $n + 1$ **do**
4: $\quad SUM, CARRY \leftarrow SUM + CARRY + X_i \times Y$
5: $\quad SUM, CARRY \leftarrow SUM + CARRY + SUM_0 \times M$
6: $\quad SUM \leftarrow SUM/2$
7: $\quad CARRY \leftarrow CARRY/2$
8: **end for**
9: **while** $CARRY \neq 0$ **do do**
10: $\quad SUM, CARRY \leftarrow SUM + CARRY$;
11: **end while**
devolver $P = SUM$;

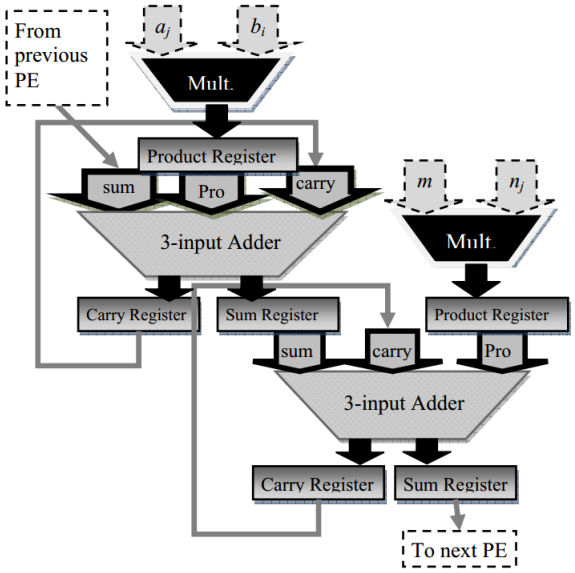


Figura 3.1: Estructura de un elemento de procesamiento (PE) en la arquitectura del multiplicador reportada en [48]

En [43] se propone una arquitectura *hardware* para multiplicación en el campo primo $\mathbb{GF}(p)$ usando el algoritmo de Montgomery. El multiplicador es parametrizable, permitiendo la evaluación de diferentes tamaños de operandos, así como diferentes tamaños de dígitos de la forma 2^β. El diseño utiliza solo tres multiplicadores de $k \times k$ y tres sumadores de 2k-bits. La organización del *hardware* del multiplicador maximiza el uso de los multiplicadores procesando iterativamente el multiplicando, multiplicador y el módulo. Ya que el diseño es parametrizable, permite un estudio del balance entre área-desempeño, de tal manera que se pueda encontrar la mejor implementación de acuerdo a los requerimientos de área-tiempo. El algoritmo que se presenta es una modificación del algoritmo propuesto por Walter [67], el cual es un algoritmo que calcula la multiplicación Montgomery en forma iterativa y en tiempo constante, ya que no necesita la resta final que se realiza en el algoritmo Montgomery original. El algoritmo que se presenta en [43] se muestra en el Algoritmo 15.

En la Tabla 3.1 se resumen los resultados de síntesis de cada uno de los trabajos analizados en esta sección.

3.2. Algoritmos de exponenciación en $\mathbb{GF}(p)$

En [48] se propone un exponenciador modular utilizando el algoritmo *Montgomery Powering Ladder*, este exponenciador utiliza 2 multiplicadores en paralelo mostrados en

Algoritmo 15: Algoritmo de multiplicación Montgomery iterativo

Entrada: integers $X = \sum_{i=0}^{n} X_i \beta^i, Y = \sum_{i=0}^{n} Y_i \beta^i, M = \sum_{i=0}^{n} M_i \beta^i$, con $0 < X, Y < 2 \times M, R = \beta^{n+1}$ con $gcd(M, \beta) = 1$, y $M' = -M^{-1} \mod \beta$

Salida: $A = \sum_{i=0}^{n} a_i \beta^i = X \times Y \times R^{-1} \mod M$

1: $A \leftarrow 0$;
2: **for** $i \leftarrow 0$ to n **do**
3: $\quad c_0 \leftarrow 0$
4: $\quad$ **for** $j \leftarrow 0$ to n **do**
5: $\quad\quad s_j \leftarrow [a_0 + X_j \times Y_i]$ /*a_0 toma cada valor de a_j*/
6: $\quad\quad$ **if** $j = 0$ **then**
7: $\quad\quad\quad q_i \leftarrow (s_j \times M') \mod \beta$
8: $\quad\quad$ **end if**
9: $\quad\quad r_j \leftarrow q_i \times M_j$
10: $\quad\quad \{c_{j+1}, t_{6j}\} \leftarrow s_j + r_j + c_j$
11: $\quad\quad A \leftarrow SHR(A)$
12: $\quad\quad a_n \leftarrow t_{6j}$
13: $\quad$ **end for**
14: $\quad A \leftarrow SHR(A)$ /*$A \leftarrow A/\beta$*/
15: $\quad a_n \leftarrow c_{n+1}$
16: **end for**
devolver A;

Tabla 3.1: Implementaciones compactas del algoritmo Montgomery.

Paper	bits	FPGA	Tiempo (μs)	ciclos	*slices*	Frec. (MHz)	Mbps	Mbps/*slices*
[48]	1020	Spartan-3E500	7.62	907	1553	119	133.80	0.086
[43]	1024(k=16)	Spartan-3E500	39.41	4288	147	108.8	25.98	0.177
[43]	1024(k=32)	Spartan-3E500	15.63	1120	327	71.65	65.48	0.200
[43]	1024(k=16)	xc5vlx50	10.93	4288	94	102.42	24.45	0.260
[33]	1024	Virtex II V2-600	8.72	1056	3390	121	117.00	0.035
[72]	1024	XC2V3000	9.28	1031	4512	114.2	113.40	0.025

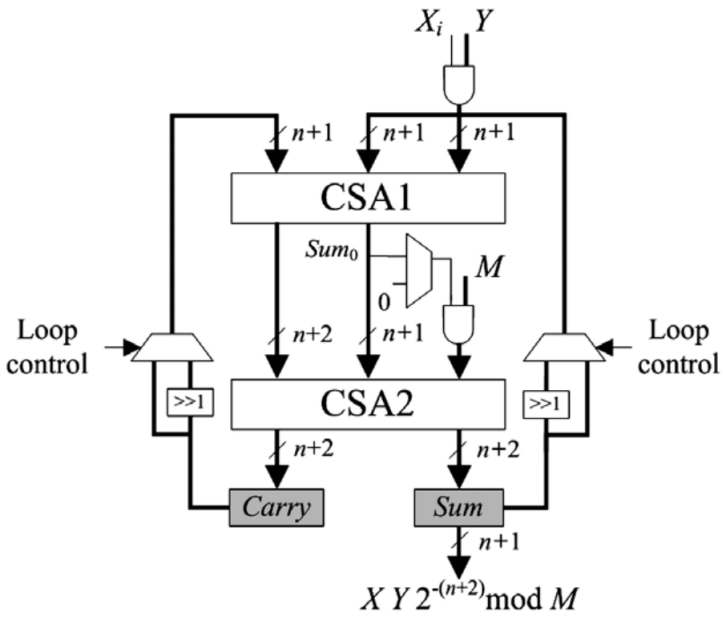


Figura 3.2: Multiplicación Montgomery con CSA (Carry Save Adder) reportada en [72]

la sección anterior de la misma referencia. Estos multiplicadores fueron configurados para utilizar 5 PE, mostradas en la sección anterior, ver Figura 3.1.

En [39] se presenta una implementación del algoritmo Montgomery propuesta por Alexandre Tenca y Cetin Koc [64]. El algoritmo MWRkMM recorre los operandos Y y M palabra a palabra y el operando X dígito a dígito, realizando la reducción un dígito a la vez. La implementación del algoritmo MWRkMM se basa en el uso de arreglos sistólicos de PEs como se discute en [64] en combinación con registros para los operandos y registros de corrimiento. Cada PE consiste de un multiplicador y una unidad de adición. Para la exponenciación modular se utiliza el algoritmo de exponenciación de derecha a izquierda, Algoritmo 10.

Timo Alho et al. en [1] proponen una implementación compacta para FPGAs de un exponenciador modular adecuado para aplicaciones criptográficas. Para la multiplicación modular se utiliza el algoritmo de Montgomery y para la exponenciación modular el algoritmo de estrategia binaria. Para lograr una área compacta y tener una arquitectura simple se utiliza el algoritmo de estrategia binaria de izquierda a derecha. Esto es por que solo se requiere un registro para almacenar los resultados temporales, en vez de dos, como en la estrategia de derecha a izquierda. La desventaja de usar el algoritmo de izquierda a derecha es su tiempo de ejecución, sin embargo solo se utiliza un multiplicador en vez de dos, logrando una menor área que en el otro algoritmo. Para la multiplicación Montgomery, los enteros A, B y el precalculado

M' son descompuestos en operandos de k-bits. El multiplicador modular consiste de una máquina de estados y un *datapath* para realizar los cálculos. Las entradas de la arquitectura X, M y E son almacenadas en las memorias DPRAMs integradas en el FPGA como dígitos de k-bits. El FPGA utilizado para probar la arquitectura es un Altera Stratix EP1S40. En esta arquitectura se logra un buen desempeño a la vez que se consume poca área del dispositivo.

Gustavo D. Sutter et al. [62] proponen una arquitectura para multiplicación y exponenciación modular para RSA basada en un enfoque dígito serial. La arquitectura propuesta utiliza el algoritmo de Montgomery para realizar la multiplicación. Se muestran implementaciones del algoritmo de estrategia binaria tanto de derecha a izquierda, como de izquierda a derecha. La arquitectura presentada tiene las siguientes características principales:

- Uso de un enfoque dígito serial para la multiplicación Montgomery.
- Conversión de una representación CSA a multiplicaciones intermedias usando *carry-skip-addition*. Esto permite reducir el camino crítico aunque con un área pequeña y una penalización en la velocidad.
- Cálculo previo del cociente en cada iteración de Montgomery con el fin de acelerar la frecuencia de operación.

Los resultados se presentan para un FPGA de Xilinx, el Virtex 5.

En [70] se propone una arquitectura para exponenciación modular, basada en un enfoque de multiplicando común en la multiplicación modular Montgomery. Junto con la exponenciación modular Montgomery Powering Ladder, una rápida, compacta y simétrica arquitectura es propuesta para su implementación en *hardware*. La arquitectura se compone de un grupo de elementos de procesamiento a lo largo de la línea central y dos grupos simétricos de unidades de acumulación en los dos lados. Los elementos centrales realizan reducciones modulares, mientras que las unidades simétricas en ambos lados acumulan los resultados de la multiplicación modular. Debido a la arquitectura simétrica y al algoritmo de exponenciación Montgomery Powering Ladder, la exponenciación modular es inmune a fallos y ataques de energía simples.

En [55] se propone la implementación de una arquitectura *hardware* para multiplicación y exponenciación modular basada en FPGAs. Se utilizan dos de los principales algoritmo de multiplicación, el algoritmo Montgomery y el algoritmo de Karatsuba. Para la exponenciación modular se utiliza el algoritmo de exponenciación binaria de izquierda a derecha, aprovechando el paralelismo en las multiplicaciones. Los resultados se sintetizan en un FPGA Xilinx Virtex 7. También se presenta una implementación

de evaluación de la arquitectura, en lo que se conoce como SoC *(System-on-chip)*. Se utiliza una tarjeta de la familia Xilinx Zynq 7000, la cual combina un dual-core ARM Cortex-A9 MPCore con la tecnología de Xilinx de 28nm de lógica programable.

En la Tabla 3.2 se muestran los resultados que cada uno de estos trabajos obtuvieron en su implementación en FPGAs. Los resultados que se muestran son área, *throughput* y eficiencia. El trabajo que reporta un exponenciador con la menor área es [55] con $w = 16$, logrando una implementación que utiliza 358 *slices* en un FPGA Virtex 7. Esta arquitectura también logra la mejor eficiencia con 0.286 kbps/*slice*. En cuanto a *throughput*, el trabajo que tiene el mejor resultado es [62] con 744.6 kbps en un FPGA Virtex 5. Sin embargo, utiliza un área de 7303 *slices* y logra una eficiencia de 0.102 kbps/*slice*. En esta tesis se plantea como meta diseñar un exponenciador que ocupe menos recursos de área que los reportados en el estado del arte ([55]) sin reducir la eficiencia.

Tabla 3.2: Resultados del estado del arte para trabajos de exponenciación modular

Trabajo	Método	Tecnología	Área (*slices*)	Periodo (ns)	#ciclos	avg Cyc (x 1000)	avg T (ms)	Thrg (Kbps)	(Thrg/*slices*)
[48]	MPL[1]	Spartan3	3899	8.4	$(s(14+s)/p) \times size$	946	7.95	128.84	0.033
[39]	RLE[2]	Virtex 2	12791	11.97	-	319	3.82	268.06	0.021
[1]	LRE	Altera EP1s40	341 LEs	5.05	$(n+3)(n+4)(l+p)$	5550	28.03	36.52	0.107
[62](d=2)	MSB[3]	Virtex 5	7303	2.6	$(k/2+w2) \times (ke+2)$	529	1.38	744.6	0.102
[62](d=4)	LSB[4]	Virtex 5	6217	4.5	$(k/4+w4) \times (2 \times ke+3)$	397	1.79	572.5	0.092
[62](d=2)	LSB	Virtex 5	4060	2.6	$(k/2+w2) \times (2 \times ke+3)$	793	2.03	503.6	0.124
[70]	MPL	Virtex 5	3218	2.89	$N(N+N/w+g)$	1097	3.18	322.01	0.100
[55](w=16)	RLE	Virtex7	385	2.13	$2n((d+1)(d+1)+(l+1))$	-	9.28	110.34	0.286
[55](w=64)	RLE	Virtex 7	3046	4.97	$2n((d+1)(d+1)+(l+1))$	-	1.52	673.18	0.221

[1]Montgomery Powering Ladder
[2]Right to Left Exponentation
[3]Most Significant Bit
[4]Left to Right Exponentiation

3.3. Sumario

En este capítulo se presentaron los trabajos más relevantes en cuanto a arquitecturas *hardware* para operaciones aritméticas en campos finitos. En especial, operaciones de multiplicación y exponenciación en el campo primo $\mathbb{GF}(p)$. Estas operaciones son muy utilizadas en la criptografía de llave pública, en algoritmos como es RSA. Se mostraron diversas implementaciones del algoritmo de Montgomery, como son implementaciones con arreglos sistólicos, algoritmos iterativos, algoritmos que hacen uso de *carry save adders* (CSA) para agilizar las sumas y restas parciales. Los trabajos presentados tienen un enfoque de implementación en FPGAs. La gran mayoría de estas implementaciones, a pesar de que algunas tienen el enfoque de ser compactos, hacen un gran uso de los recursos del FPGAs. El análisis de estas implementaciones proporciona una métrica para validar las arquitecturas propuestas y poder compararlas.

En el siguiente capítulo se presentan las arquitecturas propuestas en esta tesis para las implementaciones de multiplicación y exponenciación en el campo primo $\mathbb{GF}(p)$ con el enfoque de ser lo más compactas posible.

Capítulo 4

Arquitecturas compactas para multiplicación y exponenciación en $\mathbb{GF}(p)$

A fin de atacar el problema planteado, en este capítulo se presenta el diseño y construcción de dos arquitecturas hardware que implementan las operaciones más demandantes en esquemas de cifrado de llave pública: multiplicación y exponenciación en $\mathbb{GF}(p)$. Se describe la arquitectura de un nuevo multiplicador en el campo $\mathbb{GF}(p)$. Utilizando este multiplicador como base, se describe una arquitectura *hardware* para exponenciación modular en campos finitos $\mathbb{GF}(p)$. La principal meta de diseño en las arquitecturas *hardware* propuestas es que estas sean ligeras, es decir que ocupen la menor cantidad de área posible. Uno de los principales enfoques para lograr arquitecturas *hardware* compactas de multiplicación y exponenciación es realizar implementaciones dígito-dígito y almacenar los resultados parciales en bloques de memorias.

4.1. Multiplicador Montgomery

El algoritmo y arquitectura presentada en esta sección se basan en un trabajo previo, [43]. En [43] se presenta un multiplicador modular compacto para $\mathbb{GF}(p)$ basado en el algoritmo de Montgomery iterativo [67].

4.1.1. Multiplicación Montgomery Iterativa

Cualquier número a puede expresarse en alguna base β como:

$$a = \sum_{i=0}^{m-1} \alpha_i \beta^i, \alpha_i \in \{0, 1, \ldots, \beta-2, \beta-1\}. \tag{4.1}$$

Es común que se utilice $\beta = 2^k$, con $k = 1, 2, 4, 8, 16, 32$ o 64, como representación en las computadoras. Sean $x = \sum_{i=0}^{m-1} x_i \beta^i$ y $y = \sum_{i=0}^{m-1} y_i \beta^i$. Sea $r = \beta^m$. En una

multiplicación Montgomery $MM(x, y) = x \times y \times r^{-1}$, el resultado de la operación será $A = x \times y \times \beta^{-m} = \sum_{i=0}^{m-1} a_i \beta^i$.

$$\begin{aligned} A &= x \times y \times \beta^{-m} \\ &= \left(\sum_{i=0}^{m-1} x_i \beta^i \right) \times y \times \beta^{-m} \\ &= \left(\sum_{i=0}^{m-1} x_i \beta^i \times y \right) \times \beta^{-m} \\ &= (x_0 \beta^0 \times y) \times \beta^{-m} + \cdots + \\ &\quad (x_i \beta^i \times y) \times \beta^{-m} + \cdots + \\ &\quad (x_{m-1} \beta^{m-1} \times y) \times \beta^{-m} \\ &= \sum_{i=0}^{m-1} \text{REDUCE}(x_i \beta^i \times y). \end{aligned} \tag{4.2}$$

Cabe mencionar que la suma acumulada de cada término de la sumatoria de la Ecuación 4.2 implicaría nuevamente una reducción del término actual a reducir con el valor previamente calculado de A. En la función $\text{REDUCE}(x_i \beta^i \times y)$, los cálculos serían los siguientes.

$$\begin{aligned} q &= \left((x_i \beta^i \times y) \mod \beta^m \right) \times p' \mod \beta^m \\ &= ((x_i \times y) \mod \beta^m) \times p' \mod \beta^m \\ &= ((x_i \times y) \mod \beta) \times p' \mod \beta \qquad (4.3) \\ a &= \left((x_i \beta^i \times y) + q \times p \right) / \beta^m \\ &= ((x_i \times y) + q \times p) / \beta \qquad (4.4) \end{aligned}$$

De esta forma, A se calcula en m iteraciones, siendo A_i el valor de A en la iteración i. Los cálculos que se hacen en la función REDUCE son:

$$q = (A_i + x_i \times y) \times p' \mod \beta \tag{4.5}$$

$$a = ([A_{i-1} + x_i \times y] + q \times p) / \beta. \tag{4.6}$$

Cálculo de p'

Como se mencionó anteriormente, para realizar una multiplicación Montgomery se requiere que los valores p' y r^{-1} sean previamente calculados, con $r \times r^{-1} - p \times p' = 1($

mod β^m). De esta igualdad se desprende que $-p \times p' \mod \beta = 1$, por lo tanto p' es el inverso módulo β de $-p$ en base β, es decir $p' \mod \beta = (-p)^{-1}_{\beta}$. También, $-p = \beta - p_0$, siendo p_0 el primer dígito de p. Si suponemos que la base $\beta = 2^k$, para algún número entero positivo k y que $p_0 = \beta - 1$. Entonces:

$$\begin{aligned} p' \mod \beta &= (-p)^{-1}_{\beta} \\ &= (\beta - p_0)^{-1}_{\beta} \\ &= 1. \end{aligned} \tag{4.7}$$

Debido a esto, se puede sustituir la Ecuación 4.5 por:

$$\begin{aligned} q &= (A_{i-1} + x_i \times y) \times p' \mod \beta \\ &= (A_{i-1} + x_i \times y) \mod \beta \\ &= ([a_0 + x_i \times y_0] \mod \beta). \end{aligned} \tag{4.8}$$

Al realizarse una operación módulo β, la Ecuación 4.8 solo involucra el primer dígito de $(a_i + x_i \times y)$ y por lo tanto es el único que se utiliza en los cálculos. De esta forma, el algoritmo para calcular una multiplicación Montgomery de forma iterativa se define como se muestra en el Algoritmo 16.

Algoritmo 16: Algoritmo iterativo para multiplicación Montgomery: MM(x, y)

Entrada: $a' = a \times \beta^m \mod p = \sum_{i=0}^{m-1} x_i \beta^i$
Entrada: $b' = b \times \beta^m \mod p = \sum_{i=0}^{m-1} y_i \beta^i$
Salida: $z' = a \times b' \times \beta^{-m} \mod p = \sum_{i=0}^{m-1} a_i \beta^i$

1: $A_0 \leftarrow 0$;
2: **for** $i \leftarrow 0$ to $m-1$ **do**
3: $\quad q \leftarrow (a_0 + x_i \times y_0) \mod \beta$
4: $\quad A_{i+1} \leftarrow ([A_i + x_i \times y] + q \times p)/\beta$
5: **end for**
6: **devolver** A_i;

Cabe mencionar que el Algoritmo 16 aun requiere de la condición final del algoritmo Montgomery original, el cual puede requerir una resta extra a los pasos mostrados. Sin embargo, en [67] Walter presenta una modificación del Algoritmo 16 en la cual no es necesaria la resta final, se requiere de una iteración extra pero ejecuta la multiplicación Montgomery en tiempo constante. La única condición necesaria para

el algoritmo propuesto por Walter es que $0 \leqslant X, Y < 2 * M$. El algoritmo propuesto por Walter se muestra en el Algoritmo 17.

Algoritmo 17: Algoritmo iterativo para multiplicación Montgomery: MM(x, y), sin resta final

Entrada: enteros x y y, con $0 \leqslant x, y < 2 \times M$, $R = \beta^{n+1}$ con $gcd(M, \beta) = 1$
Entrada: $M' = -M^{-1} \mod \beta$
Salida: $A = x \times y \times R^{-1} \mod M = \sum_{i=0}^{n} a_i \beta^i$

1: $A_0 \leftarrow 0$;
2: **for** $i \leftarrow 0$ to n **do**
3: $\quad q_i \leftarrow (a_0 + x_0 \times y_i) \times M' \mod \beta$
4: $\quad A_{i+1} \leftarrow ([A_i + x \times y_i] + q_i \times M)/\beta$
5: **end for**
6: **devolver** A_i;

4.1.2. Algoritmo de multiplicación Montgomery

Las líneas 3 y 4 del Algoritmo 17 pueden escribirse de la siguiente forma:

$$t_1 = x \times y_i \tag{4.9}$$
$$t_2 = a_0 + x_0 \times y_i \tag{4.10}$$
$$q_i = t_2 \times M' \mod \beta \tag{4.11}$$
$$t_4 = q_i \times M = (r_n r_{n-1} \dots r_1 r_0) \tag{4.12}$$
$$t_5 = A_i + t_1 = (s_n s_{n-1} \dots s_1 s_0) \tag{4.13}$$
$$t_6 = t_5 + t_4 \tag{4.14}$$
$$A_{i+1} = t_6/\beta \tag{4.15}$$

Usando una representación polinomial de x en base β, la operación $x \times y_i$, puede escribirse como:

$$\begin{aligned} x \times y_i &= \left(\sum_{j=0}^{n} \beta^j x_j \right) \times y_i \\ &= \sum_{j=0}^{n} \beta^j (x_j \times y_i) \\ &= \beta^n (x_n \times y_i) + \cdots + \beta (x_1 \times y_i) + (y_0 \times y_i). \end{aligned} \tag{4.16}$$

La Ecuación 4.16 puede reemplazar a la Ecuación 4.9. M. Morales y A. Díaz en [43] al implementar una arquitectura compacta para las ecuaciones 4.9 a 4.16, explotando paralelismo, reusando operaciones y removiendo recursos de memoria innecesarios formulan un nuevo algoritmo para calcular una multiplicación Montgomery procesando dígitos de k-bit de X, Y y M a la vez, ver Algoritmo 18. El algoritmo que se presenta es una modificación del algoritmo propuesto por Walter [67], Algoritmo 17, el cual es un algoritmo que calcula la multiplicación Montgomery en forma iterativa y en tiempo constante, ya que no necesita la resta final que se realiza en el algoritmo Montgomery original, Algoritmo 7.

Algoritmo 18: Algoritmo de multiplicación Montgomery iterativo dígito-dígito

Entrada: integers $X = \sum_{i=0}^{n} X_i \beta^i, Y = \sum_{i=0}^{n} Y_i \beta^i, M = \sum_{i=0}^{n} M_i \beta^i$, con $0 < X, Y < 2 \times M, R = \beta^{n+1}$ con $gcd(M, \beta) = 1$, y $M' = -M^{-1} \mod \beta$

Salida: $A = \sum_{i=0}^{n} a_i \beta^i = X \times Y \times R^{-1} \mod M$

1: $A \leftarrow 0$;
2: **for** $i \leftarrow 0$ to n **do**
3: $\quad c_0 \leftarrow 0$
4: $\quad$ **for** $j \leftarrow 0$ to n **do**
5: $\quad\quad s_j \leftarrow [a_0 + X_j \times Y_i]$ /*a_0 toma cada valor de a_j*/
6: $\quad\quad$ **if** $j = 0$ **then**
7: $\quad\quad\quad q_i \leftarrow (s_j \times M') \mod \beta$
8: $\quad\quad$ **end if**
9: $\quad\quad r_j \leftarrow q_i \times M_j$
10: $\quad\quad \{c_{j+1}, t_{6j}\} \leftarrow s_j + r_j + c_j$
11: $\quad\quad A \leftarrow SHR(A)$
12: $\quad\quad a_n \leftarrow t_{6j}$
13: $\quad$ **end for**
14: $\quad A \leftarrow SHR(A)$ /*$A \leftarrow A/\beta$*/
15: $\quad a_n \leftarrow c_{n+1}$
16: **end for**
devolver A;

El algoritmo calcula una multiplicación Montgomery por dígitos de la forma 2^β para los operandos. Cabe mencionar que el tamaño del dígito es parameterizable y se debe seleccionar previamente. De esta forma el algoritmo trabaja realizando multiplicaciones del tamaño del dígito seleccionado. Se asume que el resultado final está almacenado en un registro de corrimiento en el cual se realizará la operación de

corrimiento hacia la derecha de 1 dígito. El algoritmo no está fijo para un tamaño de operando, también se debe seleccionar previamente el tamaño de los operandos. Dentro de los tamaños de dígito típicamente usados están $2, 4, 6, 8, 16, 32$ y 64 bits. Los tamaños de operando comunes en operaciones criptográficas de llave pública en el campo primo $\mathbb{GF}(p)$ son de $512, 1024, 2048$ y actualmente de 4096 siendo estos los que se utilizarán para esta investigación debido a que son los que actualmente proporcionan la seguridad suficiente en algoritmos como RSA.

Al ser el Algoritmo 18 parameterizable permite un estudio de distintas configuraciones tanto de dígito como de operando. Esto permite realizar experimentos que lleven a implementar arquitecturas compactas, rápidas, o con un balance área-tiempo.

Debido a que prácticamente el algoritmo consiste de dos ciclos anidados, desde 0 hasta n, siendo n el número total de dígitos que contienen los operandos, la complejidad del Algoritmo 18 es de orden $\mathcal{O}(n^2)$.

Opciones de implementación

La arquitectura propuesta en [43] se muestra en la Figura 4.1 y los enfoques para implementarla fueron los siguientes:

- **Registros de corrimiento.** En este enfoque se utilizaron registros de corrimiento circular para almacenar los operandos de entrada y el módulo, X, Y y M. El registro A, donde se almacena el resultado, se implementó con un registro de corrimiento a la derecha, utilizando como valor de relleno la parte baja o alta, dependiendo, de la suma $(s_j + r_j + c_j)$. En este enfoque también son necesarios 3 multiplicadores para implementar las operaciones de multiplicación de $k \times k$ bits.

- **BRAMs.** En este enfoque los registros de corrimiento para los operandos y el módulo, X, Y y M, fueron reemplazados por bloques de memoria. El registro de corrimiento donde se almacena el resultado permanece igual que en el enfoque anterior. Se requiere un poco de lógica adicional para manejar las direcciones de las memorias, así como para el acceso a éstas.

- **2 Multiplicadores.** Se modificó la arquitectura 2 para que solo utilice dos multiplicadores. Ya que el multiplicador *MUL2* solo se utiliza una vez al inicio de cada iteración i, se decidió utilizar solo un multiplicador para calcular r_j y q_i utilizando un multiplexor para las entradas de *MUL2* y *MUL3*.

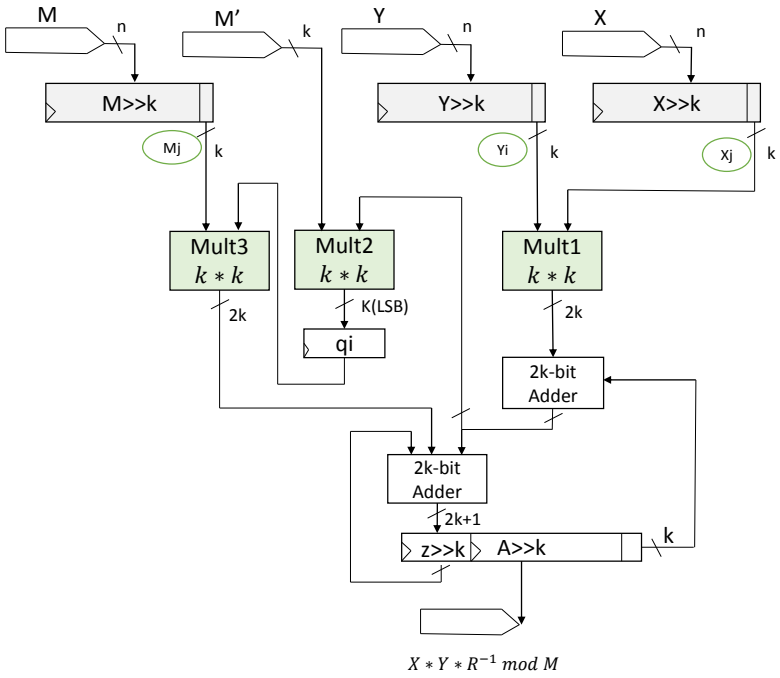


Figura 4.1: Multiplicador Montgomery compacto propuesto en [43].

En esta tesis se propone una nueva arquitectura *hardware* para la exponenciación en campos finitos $\mathbb{GF}(p)$. La operación de exponenciación se lleva a cabo a través de una serie de multiplicaciones sucesivas. Por lo tanto, para la operación de multiplicación se tomó como base la arquitectura del multiplicador Montgomery con BRAMs propuesta en [43]. Sin embargo, debido a que en la arquitectura propuesta en [43] se almacena el resultado en un registro de corrimiento y esto no permite que este valor se vuelva a utilizar inmediatamente como operando de entrada en la multiplicación siguiente, se hace una modificación a la arquitectura original, incorporando ahora una cuarta memoria BRAM para almacenar el resultado final y así reutilizar este valor como dato de entrada en la siguiente multiplicación sin requerir de lógica adicional.

Un primer enfoque propuesto es agregar un bloque de memoria a la arquitectura mostrada en la Figura 4.1 para guardar el resultado final, sin eliminar el registro de corrimiento, Figura 4.2. La ventaja de este enfoque es que no se tiene que modificar nada a la arquitectura original, simplemente el mismo resultado que se va almacenando en el registro de corrimiento se debe de ir almacenando en la memoria BRAM donde se quiere almacenar el resultado, al final de la ejecución tendremos el resultado final en la memoria BRAM. El problema es que tendremos duplicado este valor, utilizando espacio tanto del registro de corrimiento como de la BRAM utilizando un área mayor.

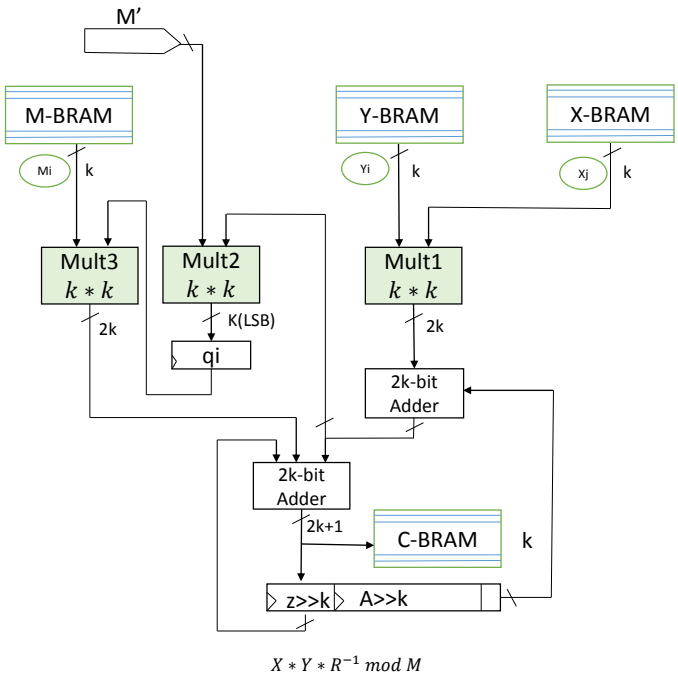


Figura 4.2: Multiplicador Montgomery compacto con 4 BRAMs.

Modificación del algoritmo

Para poder eliminar el registro de corrimiento y dejar únicamente el bloque de memoria adicional para el resultado lo primero que se debe de hacer es analizar las operaciones que requiere el Algoritmo 18. Como se puede observar en la línea 11 y línea 14 del Algoritmo 18, se requiere de un corrimiento circular de un dígito descartando este dígito o lo que es lo mismo descartando k bits, siendo k el número de bits que contiene cada dígito. Al ser descartados estos k bits menos significativos el registro se rellena con $0's$ en los k bits más significativos. Posteriormente en las líneas que siguen a los corrimientos de bits, líneas 12 y 15, se le asigna valor a los k bits más significativos.

La operación de corrimiento de bits no se puede implementar en una memoria BRAM por lo tanto hay que modificar esta operación utilizando índices tanto de lectura como de escritura en el registro donde se almacena el resultado final, al igual que se hace con los operandos de entrada.

Una operación equivalente al par de líneas $11-12$ y $14-15$, el corrimiento a la derecha de k bits y posteriormente la asignación a los k bits más significativos, es sobreescribir los bits que se desean descartar.

Hay que tener cuidado en el índice que se desea descartar. Lo primero que hay que observar es que en cada ciclo *for* interno se realizan n corrimientos, lo cual equivale a realizar n sobreescrituras, una por cada dígito, siendo n el número de dígitos de k bits en cada uno de los operandos. Posteriormente al finalizar el ciclo *for* interno se realiza otro corrimiento de k bits. Ya que estas operaciones están en el ciclo *for* externo, esto se realiza n veces.

Para poder eliminar esta operación, corrimiento de bits, se debe asignar un índice que indique en qué posición hay que sobreescribir el valor del dígito, simulando el corrimiento de bits y la escritura.

Otro aspecto importante es que en la línea 5 del Algoritmo 18 siempre se utiliza el valor de a_0. Este valor, a_0, son los k bits menos significativos del registro donde se almacena el resultado, prácticamente, es el mismo valor que se descarta al realizar el corrimiento de bits. Por lo tanto, se puede utilizar el mismo índice para leer este valor, y posteriormente escribir en esta posición, de esta forma obtendremos el valor de lo que anteriormente era a_0 y al escribir el nuevo valor en esta posición reemplazará a las operaciones de corrimiento de k bits menos significativos y a la escritura de k bits más significativos. Ya que el índice no siempre empieza en el mismo valor se necesita la operación módulo para que el índice recorra en forma circular el registro donde se almacena el resultado.

Este nuevo enfoque de procesamiento da como resultado un nuevo algoritmo de multiplicación Montgomery mostrado en el Algoritmo 19.

4.2. Arquitectura del multiplicador Montgomery

En esta sección se presentan las arquitecturas del multiplicador Montgomery propuestas. Estas arquitecturas están basadas en el nuevo enfoque del algoritmo Montgomery iterativo 19. Lo que hace diferentes a estas arquitecturas es que en la arquitectura 2 se busca reducir el número de ciclos de reloj que se requieren para realizar la multiplicación Montgomery en la arquitectura 1.

4.2.1. Arquitectura 1

Una mejora que se considera muy importante a la arquitectura del multiplicador Montgomery previamente presentada en [43] es poder prescindir del registro de corrimiento que se utiliza para guardar el resultado final, Figura 4.2. El registro de corrimiento se puede ver como una memoria. El comportamiento que se necesita para esta memoria es una que permita una lectura asíncrona y una escritura síncrona, o en

Algoritmo 19: Nuevo algoritmo de multiplicación Montgomery iterativo, sin corrimiento de bits

Entrada: integers $X = \sum_{i=0}^{n} X_i \beta^i, Y = \sum_{i=0}^{n} Y_i \beta^i, M = \sum_{i=0}^{n} M_i \beta^i$, con $0 < X, Y < 2 \times M, R = \beta^{n+1}$ con $gcd(M, \beta) = 1$, y $M' = -M^{-1} \mod \beta$

Salida: $A = \sum_{i=0}^{n} a_i \beta^i = X \times Y \times R^{-1} \mod M$

1: $indice \leftarrow 0$
2: $A \leftarrow 0$;
3: **for** $i \leftarrow 0$ to n **do**
4: $\quad c_0 \leftarrow 0$
5: $\quad$ **for** $j \leftarrow 0$ to n **do**
6: $\quad\quad s_j \leftarrow [A_{\{indice \mod n\}} + X_j \times Y_i]$
7: $\quad\quad$ **if** $j = 0$ **then**
8: $\quad\quad\quad q_i \leftarrow (s_j \times M') \mod \beta$
9: $\quad\quad$ **end if**
10: $\quad\quad r_j \leftarrow q_i \times M_j$
11: $\quad\quad \{c_{j+1}, t_{6j}\} \leftarrow s_j + r_j + c_j$
12: $\quad\quad A_{\{indice \mod n\}} \leftarrow t_{6j}$
13: $\quad\quad indice \leftarrow indice + 1$
14: $\quad$ **end for**
15: $\quad A_{\{indice \mod n\}} \leftarrow t_{6j}$
16: $\quad indice \leftarrow indice + 1$
17: **end for**
devolver A;

otro caso un bloque de memoria BRAM con 2 puertos. Esto se debe a que al final de cada ciclo de reloj se desea escribir en una localidad de memoria, y al incrementar la dirección de memoria se requiere obtener inmediatamente el valor correspondiente. Esto daría el mismo comportamiento que el registro de corrimiento, escribiendo un resultado y obteniendo el nuevo valor de $a0$.

En este trabajo se propone eliminar el registro de corrimiento de la arquitectura de la Figura 4.2 y dejar solamente los bloques de memoria necesarios tanto para los operandos de entrada como para el resultado final. Para lo cual ya se tiene preparado un algoritmo con índices para poder manejar el resultado final, Algoritmo 19. Para esto se utilizó un bloque de memoria de 2 puertos, debido a que se necesita trabajar 2 direcciones de memoria diferentes, una para la escritura y otra para ir realizando la lectura de lo que sería equivalente a $a0$, Figura 4.3. Se puede observar que se debe de ir escribiendo y leyendo el valor de una misma localidad de memoria, debido a que la lectura tiene una latencia de 1 ciclo de reloj se debe de leer ese valor un ciclo de reloj antes. Por lo tanto, se necesita que el puerto b de la nueva BRAM esté siempre una dirección de memoria adelante que la dirección de escritura, puerto a.

Al eliminar el registro de corrimiento y almacenar el resultado final en un bloque de memoria permite que este resultado sea reutilizable, esta memoria puede ser utilizada en operaciones posteriores para aprovechar este resultado sin necesidad de moverlo a otro lado, como por ejemplo cuando se tenga que usar esta arquitectura del multiplicador en una operación de exponenciación.

La nueva arquitectura del multiplicador Montgomery mostrada en la Figura 4.3 requiere de 3 multiplicadores y 2 sumadores, un registro para almacenar el valor de q_i y 4 BRAMs (Bloques de memoria RAM). Las memorias almacenarán los valores de los operandos, X, Y, así como del módulo M y el resultado A.

Latencia de la arquitectura

Toda la lógica del *for* interno en el algoritmo 19 solamente requiere de lógica combinacional, esto permite ejecutar estas instrucciones en un solo ciclo de reloj, requiriendo d iteraciones para terminar todo el *for* interno, donde d es el número de dígitos de k bits que contienen los operandos de entrada. Sin embargo, en esta implementación el cálculo de q_i así como su almacenamiento en un registro para sus posteriores usos requiere un ciclo de ejecución extra entes del inicio de cada ciclo *for* interno, prácticamente se ejecuta 2 veces la primera iteración del ciclo *for* interno, la primera solo es para calcular el valor de q_i, y en la segunda se realizan los demás cálculos. También se requiere un ciclo de reloj extra para realizar la división por β

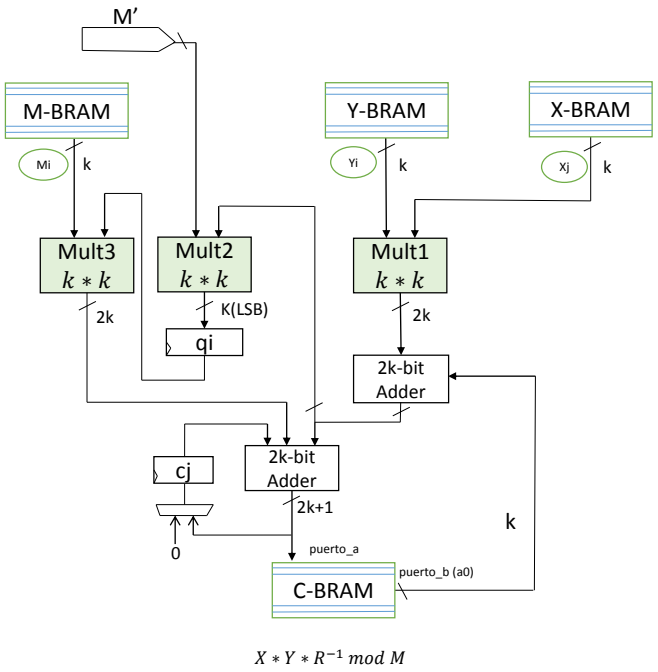


Figura 4.3: Multiplicador Montgomery compacto con BRAMs (Version 2)

después de realizar todo el ciclo *for* interno. Ya que el ciclo *for* externo se ejecuta d veces, esto da un tiempo de ejecución final de $d \times (d+2)$ ciclos de reloj para obtener el resultado final, donde d es la cantidad de dígitos que contienen los operandos. Por ejemplo si se toman operandos de 1024 bits y un tamaño de dígito de $k = 8$ bits, se tendrá $d = 1024/8 = 128$ y el número de ciclos para realizar la multiplicación requerirá $d \times (d+2) = 128 \times (128+2) = 16640$ ciclos de reloj.

Módulo de control

Se utiliza una Máquina de Estados Finitos (FSM) para el control de la arquitectura, así como para el control de las direcciones de las memorias BRAMs. Se tiene un estado para ir llenando los bloques de memorias con los datos de entrada, otro estado para cargar q_i e inicializar algunas banderas, otro estado para ejecutar cada iteración del ciclo *for* interno, así como para incrementar la dirección de memoria utilizada en cada iteración y otro estado en el que se realiza la división al final de cada ciclo *for* interno. Este estado también evalúa el final del ciclo *for* externo con lo cual se obtiene el resultado final.

En la Figura 4.4 se muestra un grafo dirigido que representa la máquina de estados finitos del multiplicador Montgomery propuesto en esta tesis, los estados son los

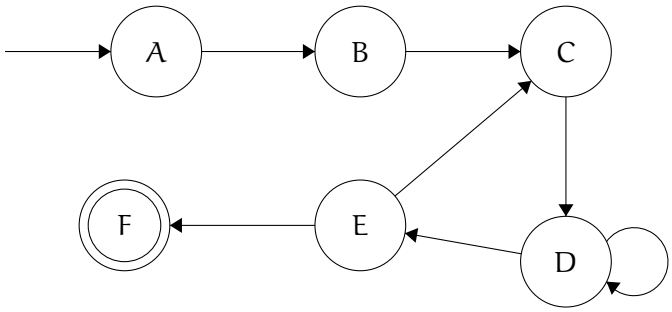


Figura 4.4: Máquina de estados finitos para el control del multiplicador Montgomery.

siguientes:

- A, representa el estado en el que la arquitectura se encuentra en modo *reset*. Todas las señales son inicializas en este estado.
- B, en este estado se llenan los bloques de memoria con los valores de Y, X y M.
- C, en este estado se almacena el valor de q_i en un registro, para uso futuro.
- D, cuando la arquitectura se encuentra en este estado, representa la ejecucion del ciclo *for* interno del Algoritmo 18.
- E, en este estado se realiza la división que le sigue al ciclo *for* interno. En caso de ser necesario, se regresa al estado C, para realizar otra iteración. En caso de haberse realizado todos los cálculos necesarios, se pasa al estado F.
- F, este es el estado final, en este momento, el resultado se encuentra en el bloque de memoria del resultado, el cual será entregado de k en k bits en cada ciclo de reloj.

4.2.2. Arquitectura 2

Una mejora para la arquitectura 1 mostrada en la Figura 4.3 es utilizar solo un ciclo de reloj para almacenar el valor de q_i en un registro temporal y al mismo tiempo si es necesario realizar la sobreescritura que se realiza al terminar el ciclo *for* interno del Algoritmo 19. Esto implicaría decrementar el número de ciclos de reloj que requiere la arquitectura 1 para calcular una multiplicación Montgomery. Esto implica modificar la máquina de estados de la Figura 4.4 por la máquina de estados que se muestra en la Figura 4.5.

Los estados en la Figura 4.5 son :

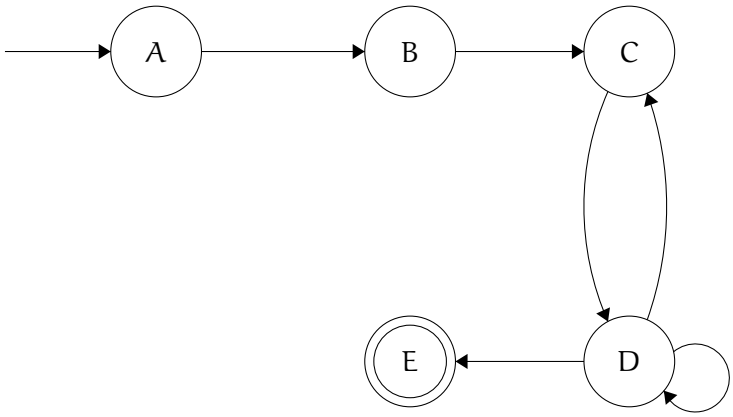


Figura 4.5: Máquina de estados finitos para el control del multiplicador Montgomery (arquitectura 2).

- A, representa el estado en el que la arquitectura se encuentra en modo *reset*, todas las señales son inicializas en este estado.
- B, en este estado se llenan los bloques de memoria con los valores de Y, X y M.
- C, en este estado se almacena el valor de q_i en un registro, para uso futuro, y si es necesario se realiza la división que anteriormente se calculaba en el estado E.
- D, cuando la arquitectura se encuentra en este estado, representa la ejecución del ciclo *for* interno del Algoritmo 18.
- E, este es el estado final, en este momento, el resultado se encuentra en el bloque de memoria del resultado, el cual será entregado de k en k bits en cada ciclo de reloj.

La reducción del ciclo de reloj que se implementó está pensada para que en el mismo instante en que se está realizando la última división del ciclo *for* externo, se calcule también el valor del q_i que se estará utilizando en el ciclo *for* interno en las siguientes iteraciones logrando que la arquitectura se ejecute en $d \times (d+1)$ ciclos de reloj, en vez de $d \times (d+2)$. Esto se logra con una idea muy sencilla, para ir accediendo a los dígitos de los operandos se incrementa la dirección de memoria en 1, lo cual da el siguiente dígito a utilizar. Al momento de realizar la última división del ciclo *for* externo, se deben tener todos los valores necesarios para calcular el nuevo valor de q_i. Para esto, en la última iteración del ciclo *for* interno, en vez de acceder a la siguiente dirección de memoria, se debe de saltar una, esto es, incrementar en 2 la dirección de memoria para leer el valor de $a0$.

Para implementar esta arquitectura, se necesita modificar los direccionamientos de las memorias, se requieren algunos registros en la máquina de estados para saber cuando será la última iteración del ciclo *for* interno, y de esta forma incrementar en 2 la dirección de memoria correspondiente para obtener el nuevo valor de $a0$ para calcular el siguiente valor de q_i.

El manejo de las direcciones hace que se incremente la lógica utilizada; pero también, se ahorra un ciclo de reloj en cada iteración del ciclo *for*. De esta forma, se necesita un ciclo para calcular q_i y si es necesario realizar la división por β, n ciclos para ejecutar el ciclo *for* interno. Esto da un total de n veces el ciclo *for* externo, el cual tiene que ejecutar $n+1$ ciclos de reloj en cada iteración, obteniendo un total de $n(n+1)$ ciclos de reloj. Recordando que d es el número de dígitos de k bits que contiene un operando. Esto da un ahorro de d ciclos de reloj para una multiplicación Montgomery.

4.3. Arquitectura del exponenciador modular para campos finitos $\mathbb{GF}(p)$.

Para la exponenciación modular, se utilizó el algoritmo *Montgomery Powering Ladder*, para realizar las multiplicaciones parciales se utilizó el algoritmo de Montgomery presentado en la sección anterior, Algoritmo 19.

El algoritmo para exponenciación *Montgomery Powering Ladder* se muestra en el Algoritmo 20. Para almacenar los valores de R0 y R1 se pueden utilizar bloques de memoria. En cada iteración del ciclo *for* se realizan 2 multiplicaciones independientes, esto es, los valores de entrada de un multiplicador no dependen del otro. Para esto, se pueden utilizar 2 multiplicadores paralelos para realizar estas operaciones, uno para calcular el nuevo valor de R0 y otro para calcular el nuevo valor de R1.

Otro aspecto importante es que para calcular el nuevo valor de R0 el primer operando del multiplicador es siempre R0 y el segundo se determina dependiendo del valor del i-ésimo bit del exponente. De la misma forma, para calcular el nuevo valor de R1 el primer operando del multiplicador siempre es R1 y el segundo se determina dependiendo del i-ésimo bit del exponente, Algoritmo 20. Tomando esto en cuenta, se propone una primera arquitectura para el algoritmo de exponenciación, Figura 4.6 (se omiten las entradas de reset, M, M' y las direcciones de memoria. El módulo MM significa un Multiplicador Montgomery de la sección anterior).

Observando que para ambos multiplicadores siempre es la misma entrada para su segundo multiplicando, ya sea R1 cuando el i-ésimo exponente es '1' o R0 cuando el

Algoritmo 20: Montgomery Powering Ladder

Entrada: m: mensaje de entrada, $d = (d_{k-1}, \cdots, d_0)$ exponente
Salida: $C = m^d$

```
1: R0 ← 1;
2: R1 ← m;
3: for i = k − 1 downto 0 do
4:    if d_i == 1 then
5:       R0 ← R0 × R1;
6:       R1 ← R1 × R1;
7:    else
8:       R0 ← R0 × R0;
9:       R1 ← R1 × R0;
10:   end if
11: end for
    devolver R0;
```

i-ésimo exponente es '0', como se muestra en el algoritmo20, se puede modificar la arquitectura para que el segundo operando del multiplicador sea el mismo. También debido a que se deben realizar 2 lecturas en R0 o R1 dependiendo del i-ésimo bit del exponente, ya sea para calcular $R0 \times R0$ o $R1 \times R1$, se necesita leer en 2 direcciones diferentes, ya sea de R0 o de R1. Por lo tanto se necesita utilizar bloques de memoria de 2 puertos, para poder hacer lecturas ya sea de R0 o R1 en 2 localidades diferentes, Figura 4.7.

Debido a que se necesita realizar varias lecturas a las memorias durante una multiplicación no se debe ir escribiendo en ellas mismas ya que la siguiente lectura daría un valor diferente del que se necesita. Para esto se necesita utilizar otros bloques de memoria para guardar los resultados parciales. A estas les llamaremos R00 y R11. En un inicio, estas memorias serán de escritura, una vez realizada la primera multiplicación se convertirán en memorias de lectura, para poder realizar la siguiente multiplicación con los nuevos valores calculados. Las memorias R0 y R1 pasarán a ser las memorias de escritura donde ahora se guardará el valor de la nueva multiplicación. Para esto se utiliza una bandera llamada `orden`, que permite permutar las memorias de lectura a escritura y viceversa en cada iteración del ciclo del *for* externo. Las memorias que son de lectura pasarán a ser de escritura y las que eran de escritura pasarán a ser de lectura, como se muestra en la Figura 4.8.

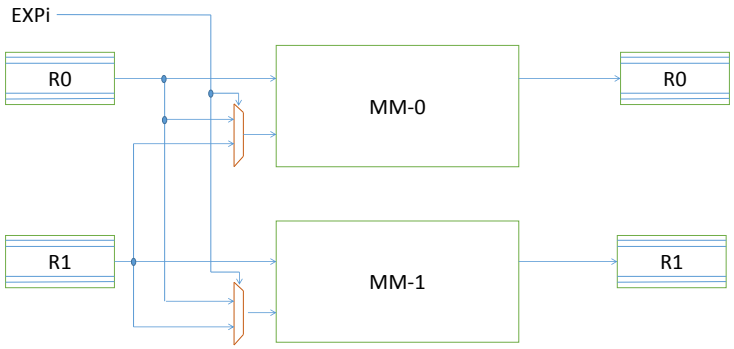


Figura 4.6: Arquitectura del algoritmo Montgomery Powering Ladder reutilizando el multiplicador Montgomery propuesto.

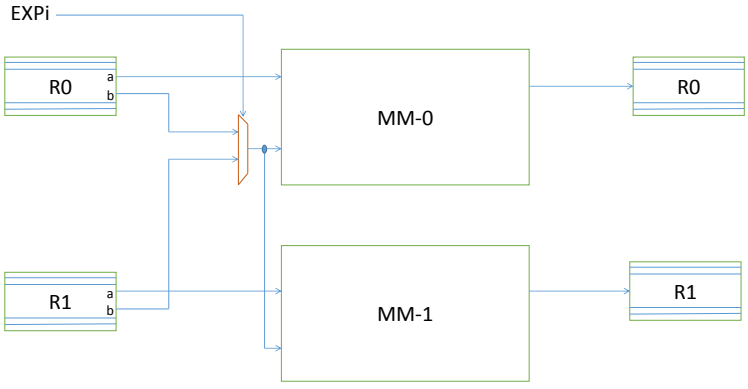


Figura 4.7: Arquitectura del algoritmo Montgomery Powering Ladder con memorias de dos puertos.

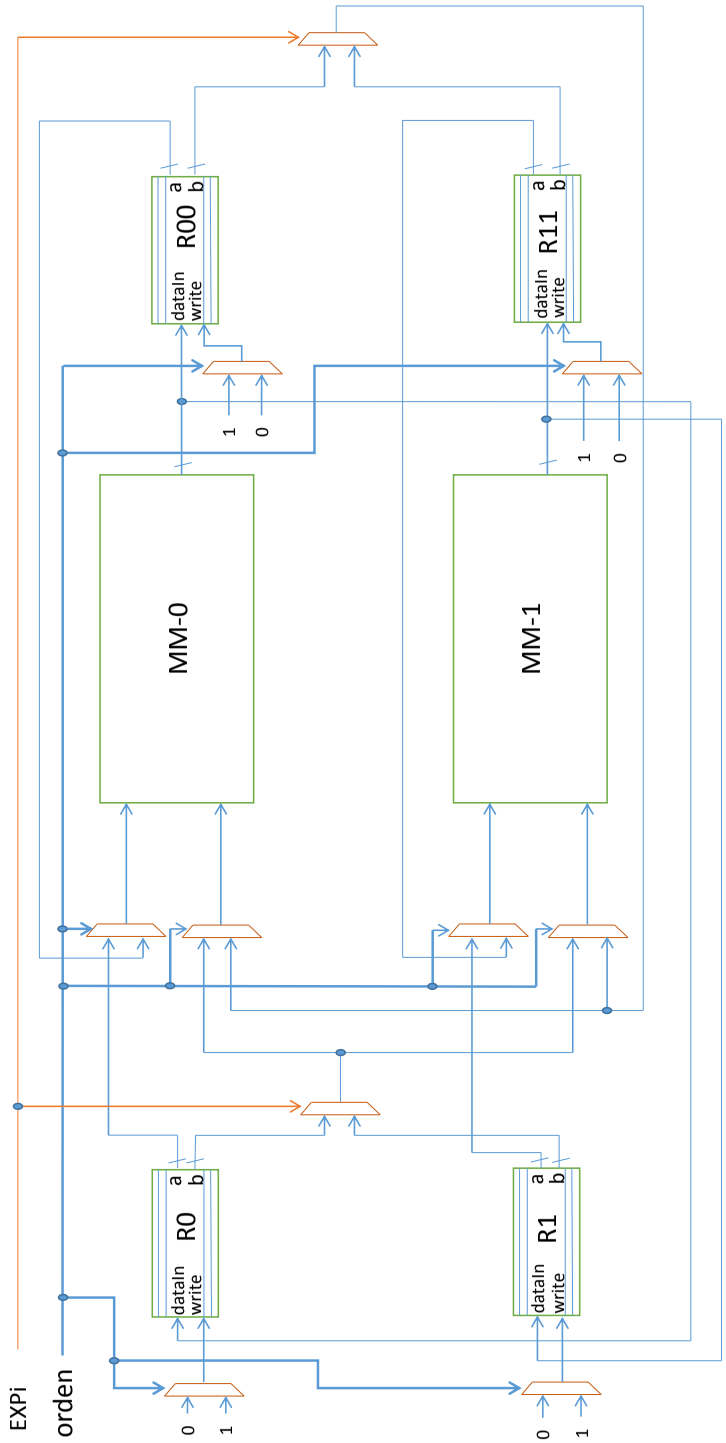


Figura 4.8: Arquitectura del algoritmo Montgomery Powering Ladder final.

Se utilizó una máquina de estados para el control de la arquitectura del exponenciador. Esta máquina de estados se muestra en la Figura 4.9, la cual contiene los siguientes estados:

- A, en este estado se llenan las memorias BRAMs con los valores necesarios para iniciar la exponenciación.
- B, durante este estado se establece la señal *reset* en 0, con lo cual en el siguiente ciclo se iniciará la exponenciación. También se inicializa un contador el cual indica la siguiente palabra de la memoria a leer donde se almacena el exponente.
- C, este estado solamente está a la espera de que el multiplicador termine su trabajo, y una vez que el multiplicador termina, se encarga de modificar la bandera $orden$, con lo cual las memorias de lectura se vuelven de escritura, y las de escritura que son las que contienen el resultado parcial anterior ahora serán de lectura. Si se ha llegado a la última dirección de memoria del exponente y se han procesado todos los bits significa que la exponenciación ha finalizado y se pasará al estado final, si no, se regresará al estado B.
- D, este estado indica que la exponenciación.

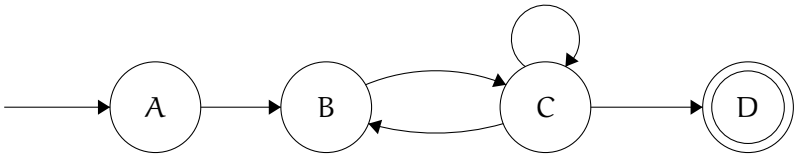


Figura 4.9: Máquina de estados finitos para el control del exponenciador modular Montgomery Powering Ladder.

Debido a que la cantidad de multiplicaciones que se realizan es igual al número de bits que contiene el exponente, el número de ciclos de reloj de esta arquitectura es de $(k \times (k+2)+2) \times bitsExponente$ al utilizar la arquitectura 1 y de $(k \times (k+1)+2) \times bitsExponente$ al utilizar la arquitectura 2 del multiplicador Montgomery, donde $k = bitsOperando/bitsDigito$. Por ejemplo, para un tamaño de operandos de 1024 bits, un tamaño de dígito de 32 y un tamaño de exponente de 1024 bits, el número de ciclos de reloj necesarios para calcular la exponenciación modular es de $(1024/32 \times (1024/32+2)+2) \times 1024 = 1,116,160$. Si la arquitectura corriera a una frecuencia de 72 MHz tardaría aproximadamente 15.42 ms en ejecutar los $1,116,160$ ciclos de reloj.

4.4. Sumario

En esta sección se mostraron los algoritmos para multiplicación y exponenciacion modular en el campo primo $\mathbb{GF}(p)$ que se propusieron y consideraron en esta tesis. Se analizaron diferentes opciones para definir un nuevo algoritmo de multiplicación Montgomery en $\mathbb{GF}(p)$ que permitiera una implementación más compacta que las reportadas en el estado del arte, eliminando la operación de corrimiento de bits hacia la derecha y reemplazándola por la sobre escritura de los bits a descartar. Utilizando este algoritmo se propuso una nueva arquitectura que solo hace uso de bloques de memorias BRAM para almacenar los operandos y el resultado final, sin necesidad de hacer uso de los registros de corrimiento. Teniendo como base la arquitectura para multiplicación Montgomery propuesta en esta investigación, se diseñó una nueva arquitectura para exponenciación Montgomery.

En el siguiente capítulo se presenta la implementación en *hardware* reconfigurable (FPGAs) de las arquitecturas propuestas. También se presentan algunas métricas que nos permiten realizar una comparación entre distintas arquitecturas, se presentan los resultados obtenidos al implementar las arquitecturas propuestas en diversos FPGAs y una comparación con los trabajos del estado del arte.

Capítulo 5

Implementación y Resultados

En este capítulo se describe la estrategia para validar y evaluar las arquitecturas propuestas para multiplicación y exponenciación en $\mathbb{GF}(p)$ desarrolladas en esta tesis. Para ello, se describe el proceso de implementación de las arquitecturas en *hardware* reconfigurable y la estrategia de evaluación a partir de los resultados de síntesis, considerando el área, velocidad y eficiencia como métricas. Se explica el método de evaluación utilizado en esta investigación. Por último se presentan los resultados obtenidos y una comparación con los trabajos más representativos en el estado del arte.

5.1. Plataforma de experimentación

El primer enfoque para implementar un algoritmo, en la mayoría de las veces, es en un procesador o microprocesador de propósito general. Éste, por lo general soporta un conjunto predefinido de instrucciones a partir de las cuales se pueden implementar instrucciones más complejas que no estén predefinidas. Sin embargo, hay que tener en cuenta que para ejecutar una operación que no está en el conjunto de instrucciones del microprocesador se necesita hacer varias llamadas a instrucciones que si están en él. Lo cual implica un consumo mayor de tiempo en las instrucciones no predefinidas. Y, si se tienen restricciones de tiempo en un algoritmo, utilizar un procesador de propósito general puede no ser la mejor idea.

Otra opción para implementar un algoritmo cuando se tiene restricciones de tiempo, lo cual significa que el algoritmo debe de ejecutarse en el menor tiempo posible, es hacer un circuito integrado de aplicación específica (ASIC por sus siglas en inglés, *application-specific integrated circuit*). Implementar un algoritmo en un ASIC significa que ese circuito solo podrá hacer cálculos específicos para una aplicación. Ésto implica que los recursos que se utilicen para diseñar ese circuito serán los mínimos posibles, y

el circuito será muy eficiente en cuanto a tiempo y consumo de energía. En un principio esto parece perfecto, los ASICs son totalmente eficientes, requieren pocos recursos y consumen poca energía. Sin embargo, los ASICs no pueden ser modificados después de su fabricación. Por lo tanto una modificación o actualización al algoritmo requiere que todos los ASICs sean reemplazados, y los clientes tiene que desechar ese circuito y reemplazarlo por uno que contenga las actualizaciones o nuevas características que necesitan. Por ejemplo, un algoritmo de seguridad informática puede implementarse con un cierto nivel de seguridad, pero en pocos meses o años la capacidad de las computadoras incrementará y se necesitará mayor seguridad en la implementación del algoritmo, o un nuevo algoritmo de seguridad informática podría implementarse y en pocos meses se podrían descubrir ataques que hagan a ese algoritmo inseguro y se necesite modificarlo para garantizar su seguridad. Este proceso es muy costoso tanto en tiempo como en dinero.

El *hardware* reconfigurable es una tecnología intermedia que permite implementar una arquitectura *hardware* pero con las flexibilidades que ofrecen los microprocesadores de propósito general. La plataforma de implementación del cómputo reconfigurable son los dispositivos FPGA (*Field Programmable Gate Arrays*), dispositivos que integran cientos o miles de módulos de lógica digital configurable, con interconexiones configurables. De esta forma, se puede implementar una aplicación en un FPGA obteniendo mejores tiempos de ejecución que implementándolos en microprocesadores de propósito general y teniendo mayor flexibilidad que las implementaciones en ASICs.

En este trabajo se utilizarán los FPGAs como plataforma de experimentación debido a que permiten una rápida generación de prototipos así como su verificación. De la misma forma, estos dispositivos ofrecen flexibilidad y rapidez al momento de implementar o modificar diversos algoritmos.

5.1.1. Lenguajes de descripción de *hardware*

Los lenguajes de descripción de *hardware* (HDL, *Hardware Description Language*) son utilizados para describir la arquitectura, diseño y operación de circuitos electrónicos. Estos lenguajes hacen posible una descripción formal de un circuito electrónico, y posibilitan su análisis automático y su simulación.

Los lenguajes HDLs pueden ser utilizados a diferentes niveles de abstracción como se muestra en la Figura 5.1. El nivel de abstracción más alto es un diseño algorítmico también conocido como diseño de comportamiento. En este nivel de abstracción el sistema se describe en términos de lo que hace o como se comporta y no en términos de sus componentes y la interconexión entre ellos. En el nivel estructural, por otra

parte, se describe el sistema como una colección de compuertas lógicas y componentes que están interconectadas para realizar una función deseada.

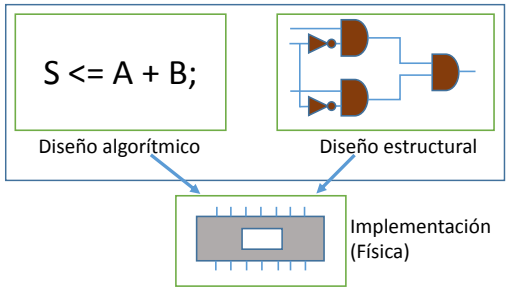


Figura 5.1: Niveles de abstracción de los lenguajes de descripción de *hardware*

Los HDLs por lo general permiten que herramientas de simulación como es *ModelSim*, *Active-HDL* o *ISim* de Xilinx permitan probar el diseño para encontrar errores y mejoras antes de implementarlo físicamente sobre algún dispositivo.

Dentro de los lenguajes de descripción de *hardware* se encuentran:

- VHDL
- ABEL
- Verilog (Cadence)
- AHDL (Altera *Hardware Description Language*)
- Handel-C (Celoxica)
- System-C (Synopsys)

Una gran ventaja de los HDLs está en que, utilizando un programa llamado sintetizador es posible inferir, a partir de la expresión textual del programa, el conjunto de operaciones lógicas y el circuito equivalente necesario para realizar la función del programa. Esto permite saltar desde el ámbito de la simulación software al de la implementación real del *hardware* sobre circuitos lógicos reales tales como los ASIC o los FPGA.

5.1.2. FPGAs - Field Programmable Gate Array

Un FPGA moderno consiste de un arreglo en 2 dimensiones de bloques lógicos programables. Alrededor del FPGA hay bloques lógicos especiales conectados a los pins de entrada/salida. Los bloques lógicos consisten de multiples celdas lógicas las

cuales a su vez consisten de generadores de funciones y elementos de almacenamiento. Xilinx y Altera son algunos de los fabricantes de FPGAs. En este trabajo se realizarán pruebas con diversos FPGAs del fabricante Xilinx, como son los FPGAs de la familia Virtex y Spartan.

Generadores de funciones

Los dispositivos FPGAs utilizan generadores de funciones para implementar lógica booleana en vez de utilizar compuertas lógicas físicamente. Por ejemplo: para implementar la función booleana:

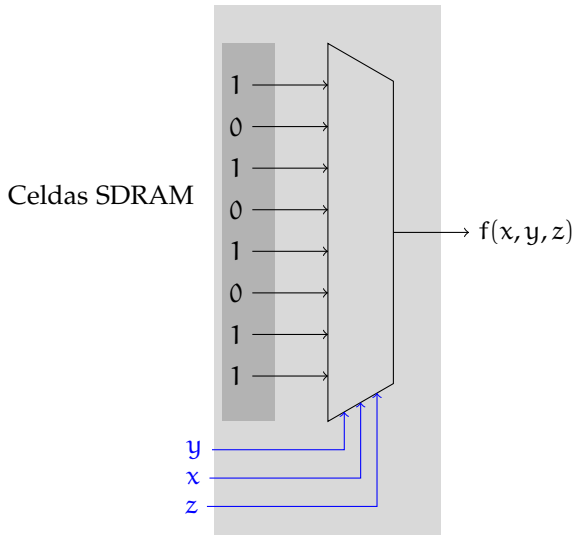
$$f(x, y, x) = xz + z'$$

usando un generador de funciones de 3 entradas, primero se crea una tabla de verdad para esta función, Figura 5.2. Para cada entrada de la función, la tabla de verdad muestra cuál será el resultado. Si cada uno de los resultados posibles de la función fueran almacenados en una memoria estática, la función puede representarse como un multiplexor de 3 entradas y una salida de 8 posibles. Las 3 entradas de la función $f(x, y, z)$ actuarían como la entrada del multiplexor seleccionando una de los valores almacenados en la memoria estática tal como se muestra en la Figura 5.2. Este proceso es comúnmente conocido como tabla de consulta o *look-up table (LUT)*.

Un aspecto interesante es que el tiempo de propagación es fijo en una LUT. No importa que tan complejo o sencillo sea el circuito, si este cabe en una LUT el tiempo de propagación que ésta necesita es siempre el mismo. Para implementar una función booleana con más entradas que las que podrían caber en una simple LUT, se utilizan multiples LUTs interconectadas. La función se puede descomponer en un subconjunto de funciones, las cuales utilicen una simple LUT y al final combinarlas para obtener el resultado final.

Elementos de almacenamiento

Mientras que los generadores de funciones proveen los bloques de construcción básicos para circuitos combinacionales, elementos adicionales dentro del FPGA proveen una gran cantidad de funciones. Los *flip-flops* Tipo-D son incorporados dentro de los FPGA. Los *flip-flops* pueden ser utilizados de diversas formas, la más común es como elementos de almacenamiento. Los *flip-flops* pueden ser configurados como *latch*, operando en el flanco de reloj positivo o negativo.



(a) Look-up table.

x	y	z	$xy + z'$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

(b) Tabla de verdad.

Figura 5.2: Correspondencia de tabla de verdad y Look-up table de la función $f(x, y, z) = xz + z'$.

Celdas lógicas

Una celda lógica en un FPGA se refiere a la combinación de un LUT y un *flip-flop* tipo D. Algunos fabricantes de FPGAs comparan la capacidad de estos dispositivos a través del número de celdas lógicas que estos contienen junto con algunos otros recursos.

Bloques lógicos

Mientras que las celdas lógicas son los bloques de construcción básicos en los diseños de FPGAs, en la actualidad es más común agrupar varias celdas dentro de un mismo bloque, conocido como **bloque lógico**. Esto permite a un grupo de celdas lógicas que están geográficamente cerca tener caminos de comunicación rápidos, reducir el tiempo de propagación y mejorar el diseño de la implementación.

Bloque de función específica

Las plataformas FPGA combinan lógica configurable con bloques integrados en el dispositivo. La razón de poner bloques de función específica dentro de un FPGA es para acercarse un poco más a los diseños ASIC, ya que estos consumen menos potencia y por lo general requieren menos espacio. Algunos de los bloques de función

específica que comúnmente se encuentran en los FPGAs actuales son:

- BRAM, Bloques de memorias RAM.
- DSPB, Bloques de procesamiento de señales.
- Procesadores.
- DCM, Administrador de reloj digital.

5.1.3. FPGAs utilizados como plataformas de cómputo

En esta tesis se usaron los siguientes FPGAs como plataformas de cómputo para implementar y validar las arquitecturas y algoritmos propuestos:

Spartan 3 se decidió utilizar este FPGA debido a que algunos trabajos del estado del arte utilizan este FPGA, de esta forma se puede tener una comparación más justa con esos trabajos. Esta familia de FPGAs fue puesta al mercado en el 2003.

Spartan 6 este FPGA viene integrado en las tarjetas de desarrollo Atlys, una de estas tarjetas se utilizó para desarrollar un codiseño *hardware-software*. En 2009 Xilinx introduce al mercado la familia de FPGAs Spartan 6.

Virtex 5 y Virtex 7 dos FPGAs modernos y muy utilizados actualmente para implementar prototipos y aplicaciones de criptografía. En Junio del 2010 Xilinx introduce la serie 7 la cual incluye la familia de FPGAs Virtex 7, la familia de FPGAs Virtex 5 fue introducida al mercado en 2009.

Las características más importantes que afectan a los resultados son el tipo de LUT, el tipo de slice, el tipo de multiplicadores embebidos y la cantidad y tipo de bloques de memoria disponibles.

En el caso del FPGA Spartan 3 se tienen LUTs de 4 entradas y multiplicadores embebidos de 18×18 bits. Se cuenta con un poco más de 1,872 Kbits total memoria que puede ser configurada en bloques y arriba de 520 Kbits de memoria RAM distribuida. Los bloques de memoria RAM permiten configurarse de uno o dos puertos.

El FPGA Spartan 6 cuenta con LUTs de 6 entradas, las cuales pueden ser utilizadas como 2 LUTs de 5 entradas o un LUT de 6 entradas. Esto permite gran flexibilidad. A diferencia del Spartan 3 en vez de multiplicadores, este FPGA cuenta con bloques DSP48A1 los cuales tienen un alto rendimiento en operaciones aritméticas y en procesamiento de señales. Estos a su vez pueden ser utilizados como multiplicadores de 18×18. Permite *pipeline* y capacidades en cascada. Los bloques de memoria de estos

FPGAs pueden almacenar hasta 18 Kbits como un solo bloque u opcionalmente un bloque BRAM puede configurarse como 2 bloques de memoria independientes de hasta 9 Kbits. Estos BRAMs pueden configurarse de uno o dos puertos independientes.

Tanto el Virtex 5 como el Virtex 7 tienen LUTs de 6 entradas los cuales pueden configurarse como 2 LUTs de 5 entradas. Los bloques de memoria de estos FPGAs pueden ser de hasta 36 Kbits y pueden configurarse para ser de uno o dos puertos independientes. Estos FPGAs contienen bloques DSP48E1 los cuales pueden ser utilizados para operaciones aritméticas y de procesamiento de señales. Estos DSPs pueden ser utilizados como multiplicadores de hasta 25×18, en complemento a dos, y opcionalmente sumador, restador o acumulador y con *pipeline*.

5.2. Descripción y validación funcional de las arquitecturas *hardware*

El ciclo de diseño para programar FPGAs inicia con la descripción funcional del diseño, utilizando cualquiera de los lenguajes de descripción de *hardware*.

En este trabajo se decidió utilizar el lenguaje de descripción de *hardware* VHDL debido a que es un lenguaje estándar y existen diversas herramientas que proporcionan soporte para este lenguaje. Dentro de las herramientas que tienen soporte para VHDL hay unas en especial dedicadas a simular un sistema descrito en lenguaje VHDL. Una de estas herramientas es *ModelSim* de Mentor Graphics el cual permite simular y depurar diseños para FPGAs descritos en VHDL y Verilog.

Se realizó la descripción de las arquitecturas en VHDL del multiplicador Montgomery y del exponenciador Montgomery Powering Ladder, se realizó un diseño de comportamiento. Estas arquitecturas fueron simuladas en el software *ModelSim*, el cual permite hacer una simulación del *hardware* así como la depuración de las mismas.

Las implementaciones fueron realizadas pensando en que se pudieran sintetizar para distintos tamaños de dígitos y operandos. Esto se logró utilizando los parámetros *generics* de VHDL. La instrucción *generics* define y declara propiedades o constantes del módulo. Las constantes declaradas en esta sección son como los parámetros en las funciones de cualquier otro lenguaje de programación, por lo que es posible introducir valores, en caso contrario tomará los valores por defecto. De esta forma se puede escribir solo una vez la descripción de *hardware* para los diseños y simular o sintetizarlos para distintos tamaños de dígitos y operandos, los cuales se especifican en el momento que se requiera. El multiplicador Montgomery se implementó para los tamaños de dígitos de $2, 4, 8, 16, 32$ bits, y para un tamaño de operando de $256, 512, 1024$ bits. El

exponenciador Montgomery Powering Ladder se implementó para un tamaño de dígito $2, 4, 8, 16, 32$ bits, y un tamaño de operandos de $256, 512, 1024, 2048, 4096$ bits ya que actualmente se están utilizando estos tamaños de operandos en las implementaciones para algunos criptosistemas como es RSA.

Para poder validar los resultados, se implementaron ambos algoritmos en el lenguaje de programación Java. Los algoritmos fueron probados para distintos tamaños de dígitos y operandos. Se crearon vectores de prueba, ver Apéndice A, para poder validar los resultados obtenidos en las simulaciones de *ModelSim*.

5.3. Implementación de las arquitecturas descritas en lenguaje HDL

Las arquitecturas *hardware* fueron implementadas para distintos FPGAs del fabricante Xilinx, éstos son Spartan 3, Spartan 6, Virtex 5 y Virtex 7. Se siguió el flujo de diseño de la Figura 5.3, en la cual se muestran los pasos para que una arquitectura descrita en VHDL pueda ser implementada en un FPGA.

- *Síntesis*, Xilinx Synthesis Technology (XST) es una aplicación de Xilinx que sintetiza diseños en lenguaje de descripción de *hardware* para crear un archivo de lista de conexiones o *netlist* llamado archivo NGC. Este archivo contiene datos del diseño lógico y restricciones. XST toma como entrada los archivos de descripción de *hardware* HDL, un archivo de proyecto con los nombres de los archivos HDL y un archivo *.xst* con las opciones de síntesis.

- *NGDBuild* lee un archivo de lista de conexiones (*netlist*) y genera un archivo NGD (Xilinx Native Generic Database) que contiene una descripción lógica del diseño en términos de elementos lógicos, como son compuertas ANDs, ORs, LUTs, Flip-Flops, y bloques de memoria RAM.

- *Map*, este programa mapea un diseño lógico a un FPGA de Xilinx. La entrada de esta etapa es un archivo NGD. La salida es un archivo NCD (Native Circuit Description) el cual contiene una representación física del diseño mapeado a los componentes del dispositivo FPGA.

- *Place and route*, también conocido como PaR, toma como entrada un archivo NCD, coloca y enruta el diseño. Su salida es un archivo NCD que será utilizado por el generador de *bitstream*.

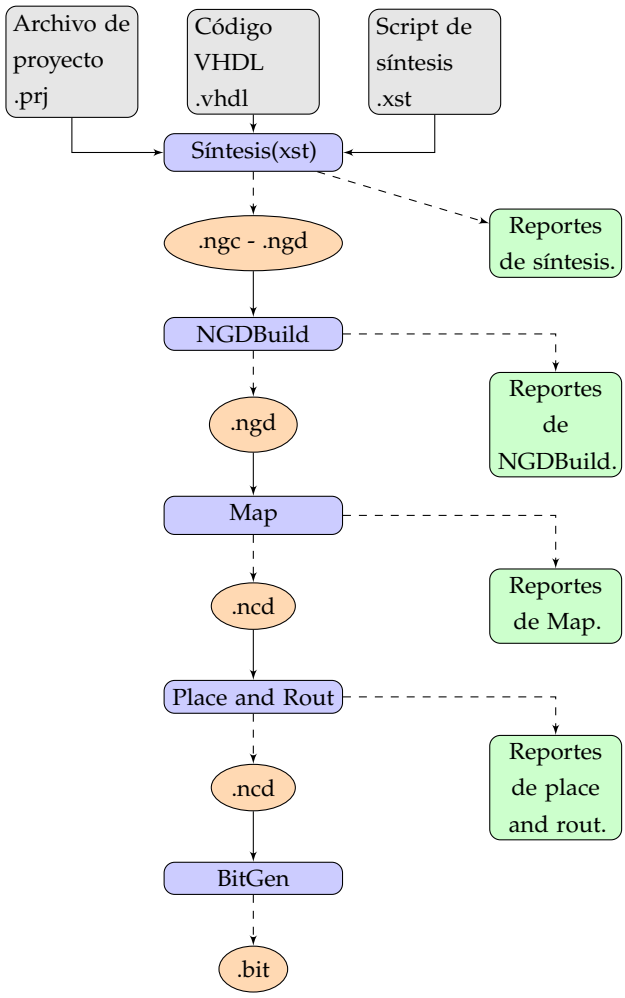


Figura 5.3: Flujo de diseño para FPGAs de Xilinx

- *BitGen*, esta herramienta toma como entrada un archivo NCD generado por la herramienta de *place and route* y genera un archivo BIT el cual es un archivo binario con extension .bit. Este archivo puede ser descargado a las celdas de memoria del FPGA para configurarlo con el diseño deseado.

La síntesis, *ngdbuild*, *map* y *place and route* se realizó utilizando el software ISE 14.6 de Xilinx.

5.4. Métricas de evaluación

Seleccionar cómo evaluar diseños en FPGAs es una tarea difícil y a veces un poco ambigua. Existen algunas métricas que son utilizadas en el estado del arte. Las dos

métricas simples, usualmente consideradas, son *complejidad en tiempo*, a veces llamada rendimiento o *throughput* y la *complejidad en área*.

Para diseños combinacionales (como son sumadores, multiplicadores paralelos completos, etc.) la complejidad en tiempo es determinada por la frecuencia de operación máxima, que a su vez es determinada por el retraso máximo de la ruta combinacional. En el caso de diseños secuenciales (como son sistemas de cifrado en bloque, multiplicadores secuenciales, etc.) la complejidad en tiempo también debe considerar el número total de ciclos de reloj requeridos antes de que el resultado esté listo. A continuación se presentan las métricas utilizadas en este trabajo.

5.4.1. *Throughput*

También conocido como rendimiento, es una métrica muy importante para medir el tiempo de desempeño de una arquitectura *hardware* [20]. El *throughput* de un diseño se obtiene multiplicando la frecuencia permitida por el número de bits procesados por ciclo de reloj. Para los algoritmos criptográficos el *throughput* se define de la siguiente manera:

$$\texttt{Throughput} = \frac{\text{Frecuencia Permitida} \times \text{Número de Bits}}{\text{Número de ciclos}} (\texttt{bits/s})$$

En un diseño en FPGAs un *throughput* alto es deseable y significa que el diseño tiene un buen desempeño.

5.4.2. Área

En términos sencillos significa cuántos recursos físicos *hardware* son ocupados por un diseño. Desafortunadamente no hay una métrica universal para medir el costo de *hardware* de un diseño basado en un FPGA. Después de implementar un diseño en un dispositivo FPGA, las herramientas de síntesis y *place and route* por lo general proveen reportes que indican cuantos recursos se están utilizando dentro del FPGA. Algunos de los recursos utilizados que proporcionan los reportes son los siguientes:

- Número de *slices*
- Número de *slices flip-flops*
- Número de 4-entradas o 6-entradas *look up table* LUTs
- Número de bloques de entrada/salida
- Número de relojes o *clocks*

- Máximo retardo en un diseño combinacional
- Máxima frecuencia de reloj permitida
- Bloque de memoria RAM, BRAMs
- Número de multiplicadores utilizados
- etc.

El área utilizada de una arquitectura *hardware* puede ser reportada en términos de *LUTs* así como en términos de *slices*. Una comparación ideal sería comparar todos los recursos disponibles en un mismo FPGA. Un diseño que hace uso de los bloques dedicados integrados en un FPGA utiliza menos recursos de lógica programable comparado con un diseño que implementa todas las operaciones en lógica programable y no hace uso de los bloques dedicados del FPGA.

Se ha observado que incluso el mismo código implementado en FPGAs diferentes de la misma familia obtiene resultados diferentes en cuanto a área y *throughput*. Esto se acentúa un poco más cuando se implementa el mismo código en FPGAs de distinto fabricante. En algunos casos, cuando se está interesado en clasificar un diseño en FPGA, se pueden ignorar algunos factores.

Se podría decir, como primera aproximación, que el diseño más rápido es aquel que logra la velocidad más alta sin importar en que dispositivo esté implementado. Sin embargo, cuando se está interesado en un diseño compacto (un diseño optimizado para área en *hardware*), este criterio no puede ser aplicado. La comparación de un diseño compacto solo puede ser justificada si es realizada en dispositivos similares.

Ambos factores área y *throughput* proveen una medida para comparar diferentes diseños. Para decidir qué tan eficiente es un diseño, en este trabajo se utilizará la siguiente métrica *Throughput*/área.

5.4.3. *Throughput*/Área

Es la razón de las dos métricas mostradas anteriormente y permite evaluar qué tan eficiente es un diseño en términos de área y *throughput*.

La razón es alta en el caso en que el *throughput* es alto y el área sea poca. Una eficiencia alta es deseable en un diseño. Esta medida también es conocida como eficiencia y se calcula de la siguiente manera para dispositivos FPGAs:

$$\text{Eficiencia} = \frac{\text{Throughput}}{\text{slices}} (\text{bps/slice}).$$

Esta métrica es comúnmente utilizada para medir arquitecturas con propósitos criptográficos especialmente cuando se requiere tomar en cuenta el área utilizada, así como la velocidad obtenida [43], [72]. Esta métrica también ha sido utilizada en algunos *banchmarks* como es ATHENa [21].

5.5. Validación de las arquitecturas mediante codiseño *hardware-software*

Para validar las arquitecturas desarrolladas para multiplicación y exponenciación se decidió implementar un codiseño *hardware-software* del algoritmo criptográfico de llave pública RSA, para cifrar y descifrar información, el cual se basa en hacer exponenciación modular en campos finitos $\mathbb{GF}(p)$.

En la Figura 5.4 se muestran las fases generales que se siguieron para implementar el codiseño *hardware-software*.

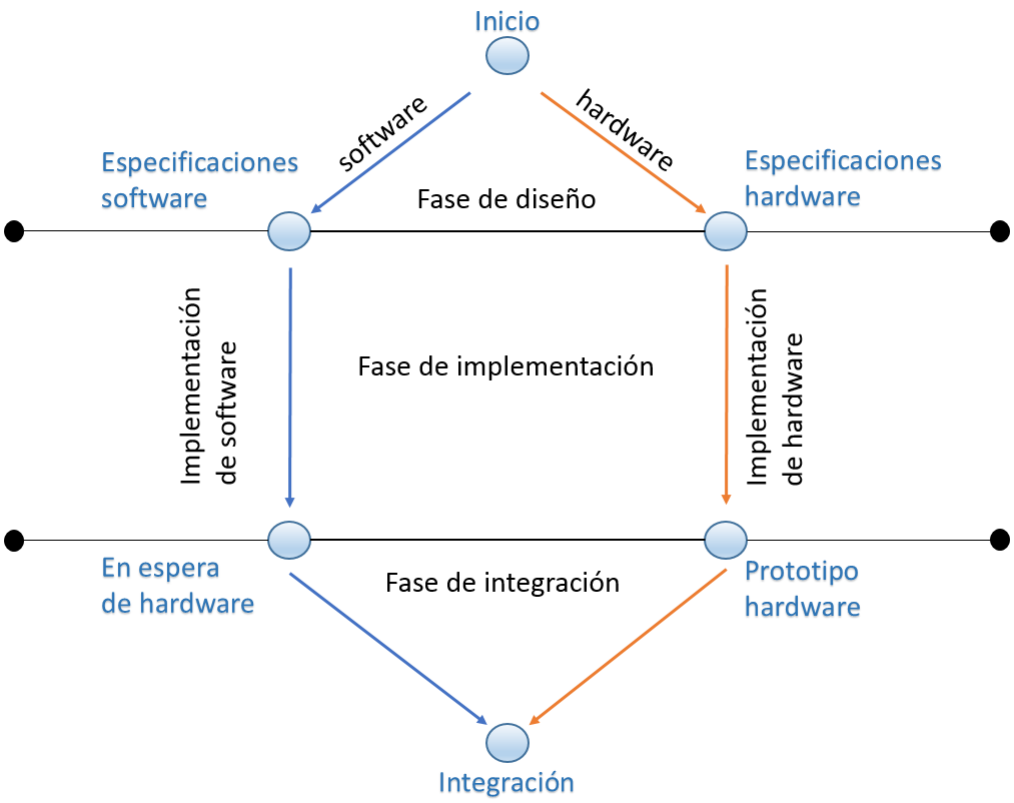


Figura 5.4: Fases de codiseño *hardware-software*.

En primer lugar se establecieron las especificaciones del diseño, lo que se desea

implementar en *hardware* y lo que se desea implementar en software. Lo que se implementó en *hardware* fueron las operaciones de multiplicación y exponenciación modular en campos finitos $\mathbb{GF}(p)$. A excepción de la etapa de integración de la Figura 5.4 todas las demás etapas han sido cubiertas en capítulos anteriores.

Las arquitecturas se validaron en una *Atlys Spartan 6 FPGA Development Board* la cual es una plataforma de desarrollo digital basada en el FPGA Spartan 6.

La tarjeta posee periféricos de gama alta, incluyendo Ethernet, video HDMI, memoria DDR2 de 128 Mb, audio y puertos USB, haciéndola ideal para un completo sistema digital en torno al procesador Xilinx de MicroBlaze, ver Figura 5.5. La tarjeta de prototipado Atlys es totalmente compatible con todas las herramientas CAD de Xilinx, incluyendo ChipScope, EDK y el WebPack libre.

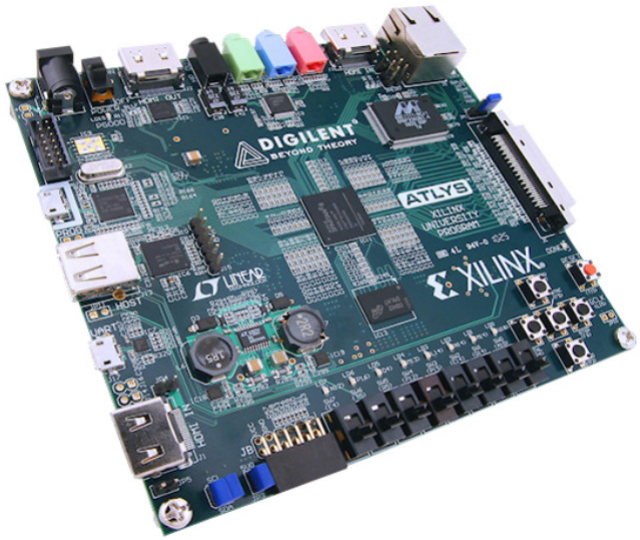


Figura 5.5: Tarjeta de desarrollo Digilent Atlys Spartan 6.

Debido a que prácticamente teniendo las operaciones de multiplicación y exponenciación modular en el campo primo $\mathbb{GF}(p)$ se tiene el algoritmo RSA, solo se necesita que la parte del software haga uso de estas operaciones.

Para implementar el codiseño *hardware-software* se utilizó la herramienta de Xilinx XPS (Xilinx Plataform Studio). En primer lugar se configuró la tarjeta a utilizar. Se configuró un procesador MicroBlaze. Para no tener problemas en cuanto a la frecuencia de operación del procesador MicroBlaze con la arquitectura se escogió una frecuencia de operación de 50MHz. Se configuró una memoria local de 16Kb, al igual que el tamaño de las memorias cache para datos y para instrucciones. De los periféricos disponibles, se agregó una memoria DDR2 y un periférico RS232 UARt. Este último permite establecer una comunicación desde la tarjeta de desarrollo con la computadora

a través de una *hyperterminal*.

Una vez configurado el procesador, se configuró la arquitectura *hardware* para el exponenciador modular en campos finitos $\mathbb{GF}(p)$. Esto se realizó utilizando el asistente de XPS, Create Peripheral. Se utilizó la opción de crear nuevo periférico. Se configuró un bus AXI4-Lite, ya que es un bus sencillo de configurar y cumple con los requerimientos necesarios para implementar la arquitectura *hardware*. En la Figura 5.6 se muestra un diagrama de bloques del bus AXI4-Lite.

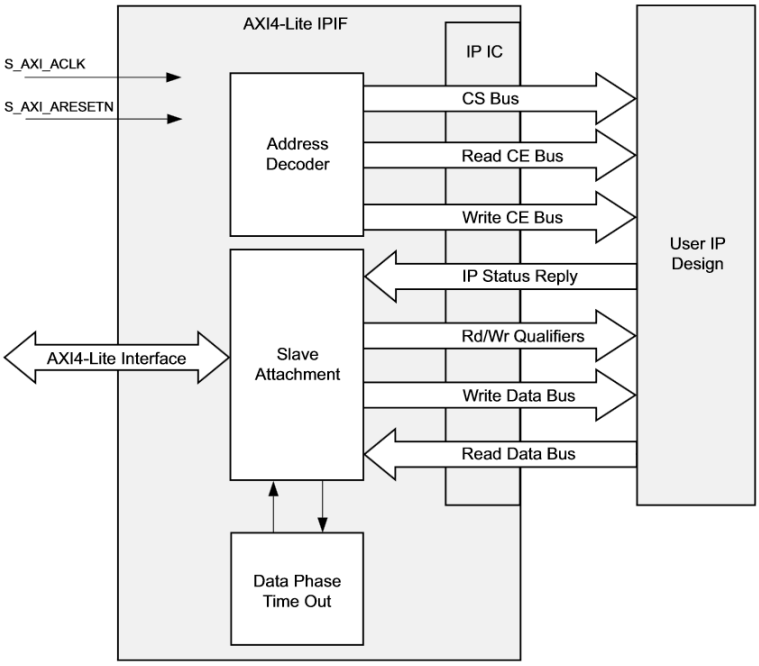


Figura 5.6: Diagrama de bloques del bus AXI4-lite.

Se decidió que el periférico tenga una señal de reset y acceso a registros desde nivel de software. Se configuraron 8 registros para ser accedidos desde software. Estos registros son manejados internamente por el bus AXI4-Lite como se muestra en la Figura 5.7 y se pueden modificar y consultar sus valores con funciones en lenguaje C, el asistente también crea estas plantillas.

El asistente para un periférico personalizado crea diversos archivos para configurar el periférico, uno de estos archivos es una plantilla en lenguaje VHDL en la cual ya están configurados los registros para ser accedidos y modificados. Otro archivo muy útil, es un archivo .h en lenguaje C, el cual se puede incluir en el código a utilizar. Este archivo .h tiene definidas varias funciones, entre las más importantes son el *reset* de la arquitectura, la escritura y lectura de los registros.

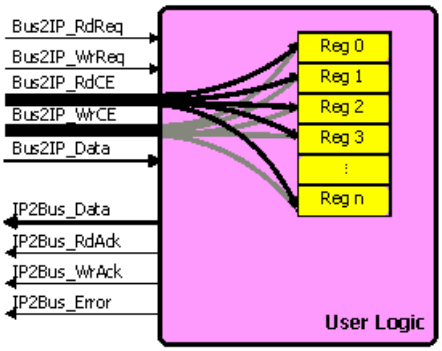


Figura 5.7: Configuración de registros para el bus AXI4-Lite.

Una vez configurado el periférico personalizado, se validaron los resultados ejecutando la operación de exponenciación modular con diversos vectores de prueba.

La arquitectura que se probó fue un exponenciador modular de 1024 bits, con un tamaño de dígito de 16 bits. Debido a que el bus AXI4-Lite solo utiliza registros de máximo 32 bits, en el código en C se generó un mensaje de 1024 que se ingresa a la arquitectura de 32 en 32 bits. Una vez que se tienen los 1024 bits del mensaje dentro de la arquitectura se procede a calcular la operación. Cuando la arquitectura termina de calcular la exponenciación modular almacena el resultado en un registro temporal de 1024 bits para que este pueda ser consultado desde el código en software, esto es el código en lenguaje C. Está validación también se realizó con los vectores de prueba del Apéndice A.

5.6. Análisis de resultados

En esta sección se analizan los resultados obtenidos al realizar el flujo de diseño de la Figura 1.2 para los FPGAs Spartan 3, Spartan 6, Virtex 5 y Virtex 7, así como para los distintos tamaños de operando y dígito y para las diferentes arquitecturas. Los resultados que se extrajeron son el número de *slices* que ocupa la arquitectura, la frecuencia máxima de operación y el número de LUTs utilizados, a partir de estos valores se calculó el *throughput* y la eficiencia.

5.6.1. Multiplicador en $\mathbb{GF}(p)$

Arquitectura 1.

En la Figura 5.8 se muestran los resultados de síntesis de la arquitectura para el multiplicador Montgomery para un tamaño de operando de 1024 bits en un FPGA

Spartan 3. Mostrando resultados para el número de *slices*, frecuencia de operación, Mbits por segundo procesados y eficiencia, la cual indica la cantidad de bits que procesa un *slice* por unidad de tiempo.

En la Figura 5.8a se puede observar la cantidad de *slices* que requiere la arquitectura para procesar una multiplicación Montgomery. Se puede observar que el tamaño de operando no afecta en nada el número de *slices*, sin embargo, el tamaño del dígito es el que afecta proporcionalmente el área usada. Esto es debido a que procesar 256 bits o 1024 solo requiere más ciclos de reloj en esta arquitectura, algunos pocos *slices* más para manejar algunas banderas, y unas memorias BRAMs más grandes. Pero las memorias BRAMs no afectan en el número de *slices* final. Por otro lado, procesar dígitos de 2 bits contra 32 bits, hace un incremento más notable, ya que los valores parciales y las señales donde se guardan valores temporales serán más grandes lo cual hace un incremento significativo en cuanto al número de *slices*.

La Figura 5.8b muestra los resultados obtenidos respecto a la frecuencia de operación. De igual forma, se observa que el tamaño del operando no afecta casi nada a la frecuencia de operación. Por otro lado, el tamaño de dígito es el que marca de manera significativa la frecuencia de operación para la arquitectura. La razón por la cual el tamaño del operando no afecta en nada a la frecuencia de operación es debido a que si se tiene un tamaño de operando mayor solo se realizará un mayor número de iteraciones, por ejemplo se realizará un mayor número de multiplicaciones, pero el tiempo de estas multiplicaciones no incrementará. Sin embargo, si se utiliza un tamaño de dígito mayor, el tiempo que tarda hacer una multiplicación sera mayor, no es lo mismo calcular una multiplicaciones de dos números de 8 bits que de dos números de 32 o 64 bits. También se muestran otras medidas muy importantes, estas son el rendimiento expresado en Mbps y la eficiencia, expresada en Mbps/*slice*. En la Figura 5.8c se puede observar que a medida que el tamaño de dígito incrementa se tiene un mejor rendimiento. Sin embargo, también hay que tener en cuenta que utilizar un tamaño de dígito más grande también implica un incremento en el número de *slices* utilizados y una baja en la frecuencia de operación. El mejor resultado se obtiene para k=32 con un rendimiento de 38.68 Mbps, para un operando de 1024 bits. El rendimiento es una métrica que solo se encarga de medir los tiempos de ejecución. Una medida que permite tomar en cuenta el área utilizada es la eficiencia Mbps/*slice*. En este caso la mejor eficiencia se logra para un tamaño de dígito k=16 con una eficiencia de 0.139 Mbps/slice, para un operando de 1024 bits.

En la Figura 5.9 se muestran los resultados obtenidos para un tamaño de operando de 1024 bits para distintos FPGAs. Los FPGAs utilizados son el Spartan 3, Spartan 6,

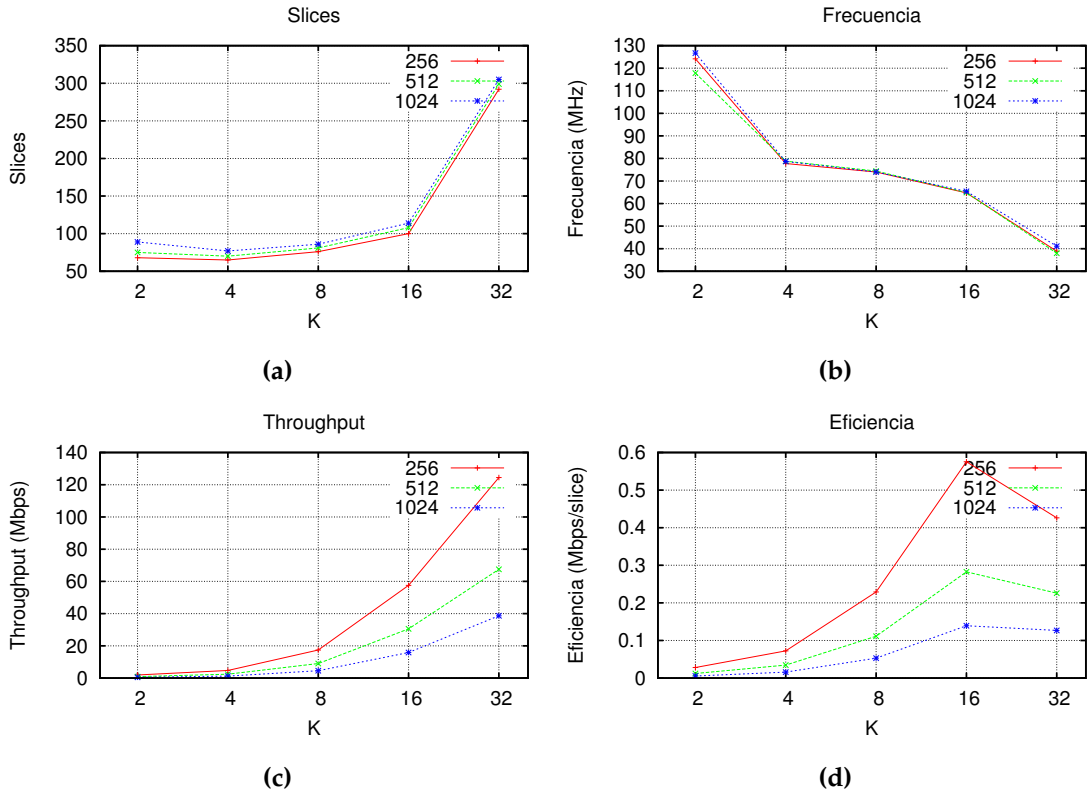


Figura 5.8: Resultados de la síntesis del multiplicador Montgomery compacto para el FPGA Spartan3E

Virtex 5 y Virtex 7. En la Figura 5.9a se puede observar que el número de *slices* que utiliza un FPGA spartan 3 el cual es un FPGA con LUTs de 4 entradas, es mayor que el número de *slices* que utiliza un FPGA con LUTs de 6 entradas como son los FPGAs Spartan 6, Virtex 5 y Virtex 7. Por ejemplo, para una multiplicación de 1024 bits con un tamaño de operando de 32 btis el Spartan 3 tiene un área de 305, el Spartan 6 de 57, el Virtex 5 de 111 y el Virtex 7 de 60 *slices*. En el Spartan 6, Virtex 5 y Virtex 7 se logran arquitecturas 5, 2.7 y 5 veces respectivamente más pequeñas que en el Spartan 3. En esta caso, la razón por la cual el número de *slices* es mejor en un FPGA que en otro es por las mejoras a la tecnología de FPGAs en vez de a las mejoras del diseño *hardware*. De la misma forma, en la Figura 5.9b se puede observar que la frecuencia de operación es mucho mejor en los FPGAs más modernos como es el Virtex 7. Esta mejora también se debe al FPGA y no a la arquitectura *hardware*. Las otras dos medidas dependen prácticamente del número de *slices* y de la frecuencia de operación, y debido a que esas dos métricas son mejores en los FPGAs nuevos, el *throughput* y la eficiencia también mejora mucho en estos dispositivos modernos. Por último, se puede observar

que el *throughput* mejora conforme se incremente el tamaño del dígito a utilizar, en este caso, el mejor *throughput* se alcanza con un tamaño de dígito de 32 bits. Por su parte, la mejor eficiencia varía un poco en cuanto al FPGA utilizado. La mejor eficiencia se alcanza con algunos FPGAs con un tamaño de dígito de 16 bits, y con otros con un tamaño de dígito de 32 bits.

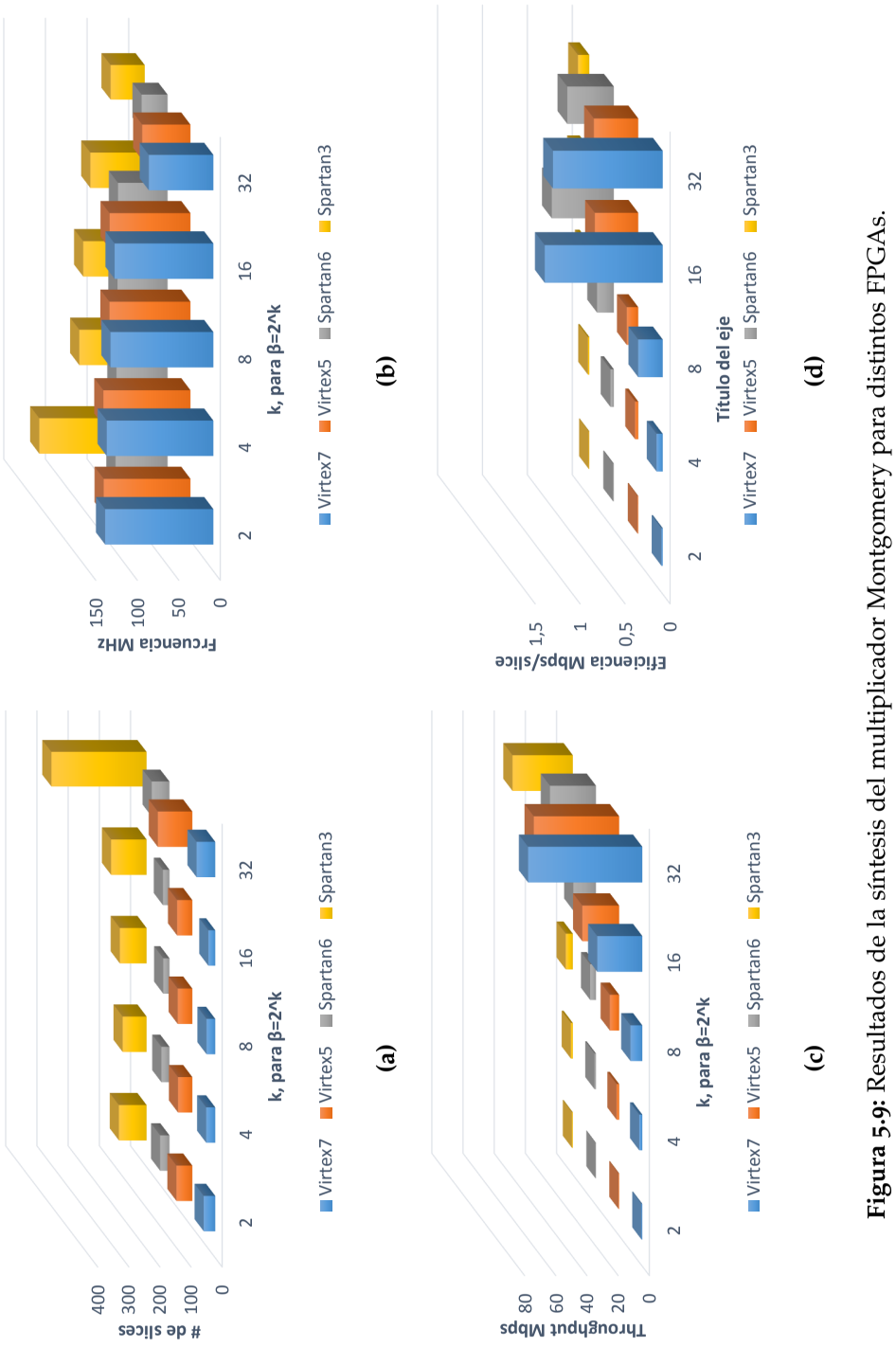


Figura 5.9: Resultados de la síntesis del multiplicador Montgomery para distintos FPGAs.

Arquitectura 2.

El número de *slices* que utiliza esta arquitectura es muy similar al número de *slices* que utiliza la arquitectura 1, solo son unos pocos *slices* más los que se necesitan en la arquitectura 2. En cuanto a la frecuencia de operación tampoco hay una diferencia significativa, los resultados son muy parecidos. El *throughput* es ligeramente superior en esta arquitectura, esto es debido a los ciclos de reloj que se ahorran. En cuanto a la eficiencia los resultados son muy similares, lo poco que se gana en el *throughput* al ahorrarse ciclos de reloj se pierde con el incremento de *slices* para controlar algunas banderas adicionales. Ambas arquitecturas son eficientemente similares. La arquitectura 1 es entre 5 y 10 *slices* más pequeña que la arquitectura 2, pero la arquitectura 2 tiene un *throughput* ligeramente superior al de la arquitectura 1.

5.6.2. Exponenciador en $\mathbb{GF}(p)$

En la Figura 5.10 se muestran los resultados de síntesis de la arquitectura para la arquitectura Montgomery Powering Ladder para un tamaño de operando de 1024 bits, así como para diversos tamaños de dígito, en el FPGA Spartan 3. Se muestran resultados para el número de *slices*, frecuencia de operación, Kbits por segundo procesados y también la eficiencia.

En la Figura 5.10a se puede observar la cantidad de *slices* que requiere la arquitectura para procesar una exponenciación modular. Se puede observar que el tamaño de operando no afecta en nada el número de *slices*, al igual que en la arquitectura del multiplicador. Sin embargo, el tamaño del dígito es el que hace un incremento significativo en cuanto al número de *slices*. La arquitectura más compacta se logra con un tamaño de dígito pequeño, $k = 2$.

La Figura 5.10b muestra los resultados obtenidos respecto a la frecuencia de operación. De igual forma, se observa que el tamaño del operando no afecta la frecuencia de operación al igual que en la arquitectura del multiplicador. Por otro lado, el tamaño de dígito es el que determina de manera significativa la frecuencia de operación.

En cuanto al rendimiento de la arquitectura se observa que el mejor resultado es el de tamaño de dígito mayor, entre mayor sea el dígito a procesar se tiene un mejor rendimiento, pero también se ocupa un mayor número de *slices*, ver Figura 5.10c.

En cuanto a la eficiencia, se observa que los mejores resultados son para k=32 y k=16.

En la Figura 5.11 se muestran los resultados de síntesis de la arquitectura del exponenciador modular Montgomery Powering Ladder para distintos FPGAs: Spartan

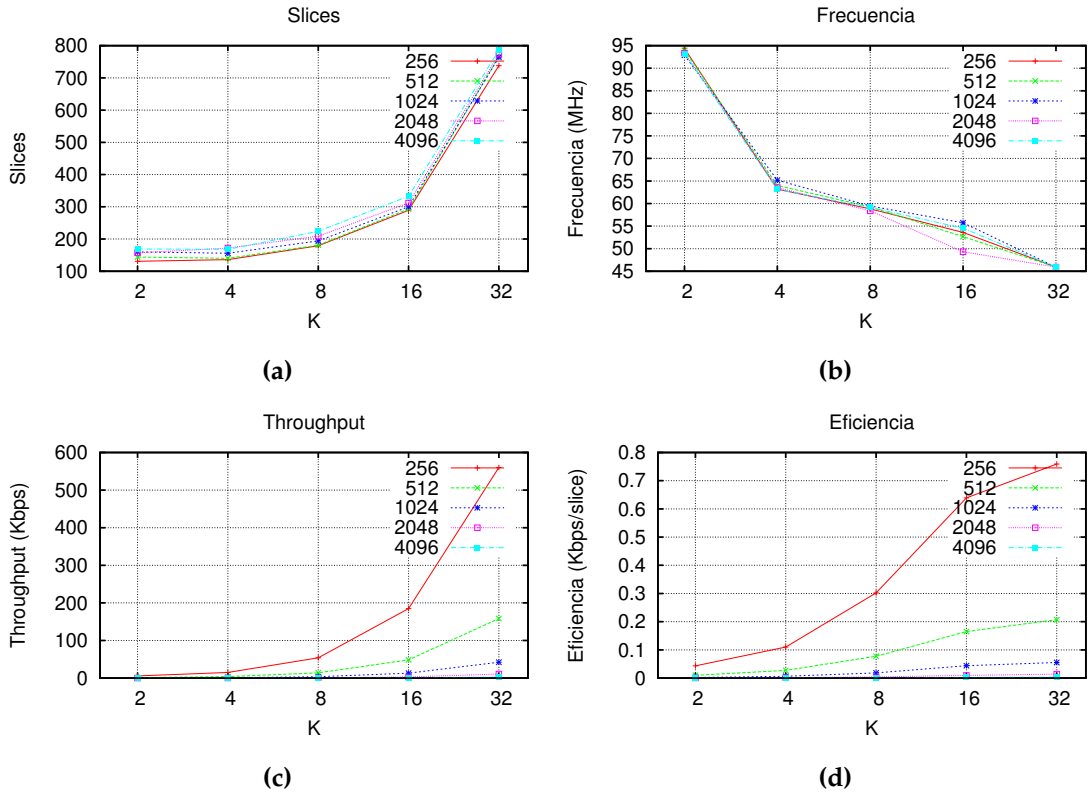


Figura 5.10: Resultados de la síntesis del exponenciador Montgomery Powering Ladder para el FPGA Spartan3E.

3, Spartan 6, Virtex 5 y Virtex 7 para un tamaño de operando de 1024 bits. En estas gráficas se puede observar que los resultados son mucho más favorables en un FPGA más reciente y con mejores capacidades que en uno relativamente antiguo o de baja capacidad. Por ejemplo, en cuanto al número de *slices* que requiere la arquitectura, los resultados son más favorables en un FPGA Virtex 7 o Spartan 6. Así también, en cuanto a la frecuencia máxima de operación que alcanza la arquitectura, es mucho mejor en el Virtex 7 que en el Spartan 3 o Spartan 6. Esta mejora de la frecuencia en el Virtex 7 con respecto a los otros FPGAs se debe principalmente a que el rendimiento que puede alcanzar un DSP en un virtex 7 es mayor que el que se puedo alcanzar en un Spartan 3 o Spartan 6. En el Spartan 3 y en el Spartan 6, la frecuencia de operación son muy parecidas, sin embargo, en lo que se refiere al área utilizada en número de *slices*, el Spartan 6 es mucho mejor que el Spartan 3. De igual forma que el multiplicador, se puede observar que el *throughput* mejora conforme se utilice un tamaño de dígito más grande. Y por su parte, la mejor eficiencia se alcanza con tamaños de dígito de 16 y 32 bits.

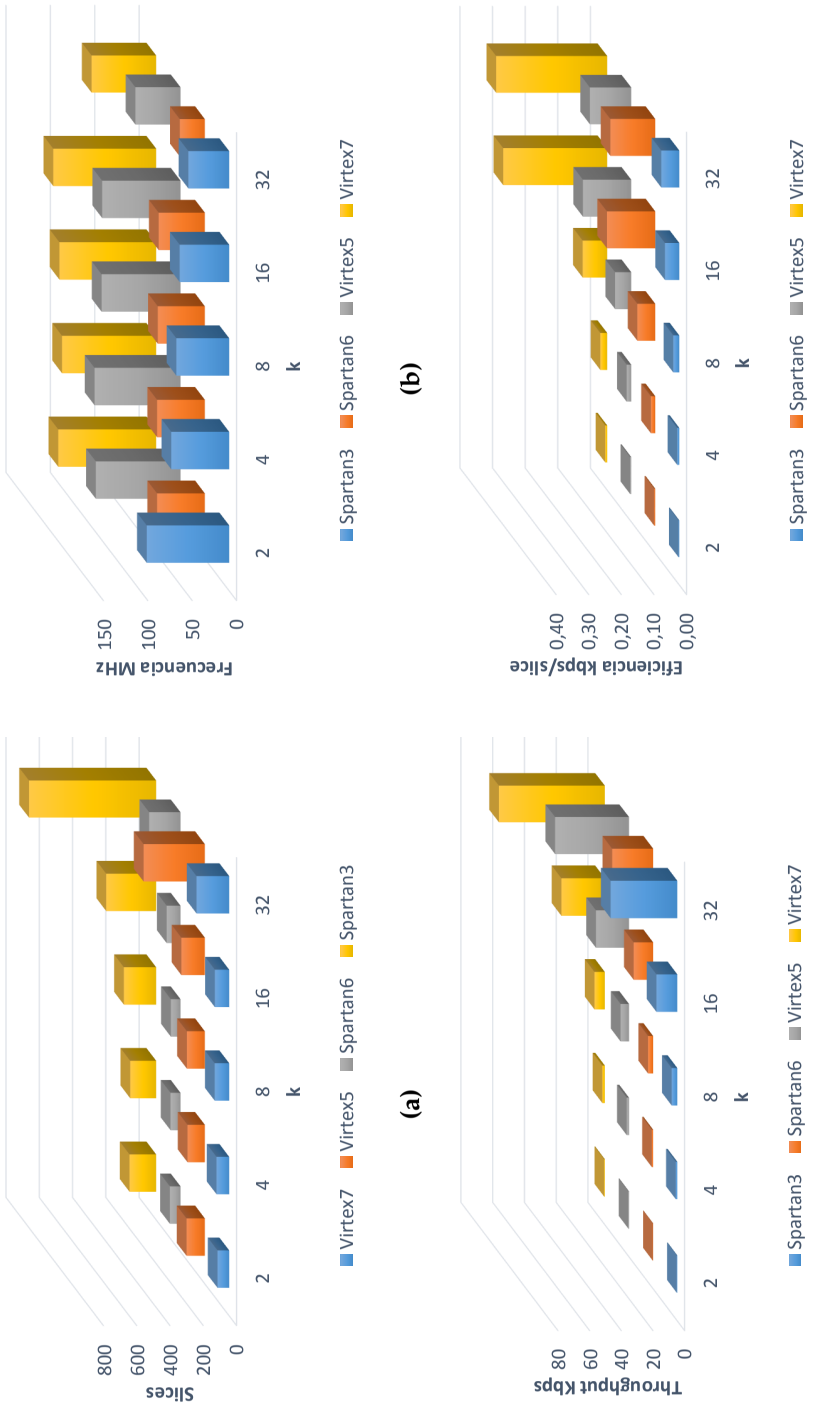


Figura 5.11: Resultados de la síntesis del exponenciador Montgomery Powering Ladder para distintos FPGAs.

Teniendo esto en cuenta, se llegó a la conclusión de que para esta arquitectura los resultados dependen en gran medida de la eficiencia del multiplicador. Existe una relación lineal de la arquitectura del algoritmo Montgomery Powering Ladder con respecto al multiplicador que se utiliza para realizar los cálculos parciales.

En la Tabla 5.1, se muestra una comparación con los trabajos del estado del arte para la exponenciación modular para campos finitos $\mathbb{GF}(p)$. En esta tabla se observa que el trabajo que hasta ahora había reportado el menor número de *slices* para una exponenciación modular es [48] un un FPGA de Xilinx y [39] para un FPGA de Altera. En [48] se realiza una implementación del algoritmo CIOS Montgomery para realizar las multiplicaciones parciales, y el método Montgomery Powering Ladder, que es el mismo que se ocupó en esta investigación. El número de *slices* que requiere su arquitectura es de 3899, y de 16 BRAMs, logrando un tiempo de ejecución de de 7.95ms para un Spartan 3E. Por otra parte en [62] se muestra uno de los trabajos que tienen un tiempo de ejecución de 1.38 ms, sin embargo hace hace uso de 7303 *slices* del FPGA, un Virtex 5 de Xilinx. Esta arquitectura requiere de 4060 *slices*, logrando un tiempo de ejecución de 2.03ms para un Virtex 5. Parecería que no hay mucha diferencia en cuanto a los *slices* de esta arquitectura con la que se mencionó anteriormente, ya que la primera utiliza un número de *slices* de 3899 y la otra de 4060. Sin embargo el uso de un FPGA a otro afecta a los resultados finales, el número de *slices* que se requieren en un Virtex 5 por lo general es mucho menor que el que se requiera para un Spartan 3e, esto se debe a que para el Virtex 5 se consiguen mucho mejores resultados tanto en área como en frecuencia, y como consecuencia se mejora el *throughput* así como la eficiencia. Esto es debido a las mejoras del FPGA Virtex 5 en cuanto al Spartan3, por ejemplo, las LUTS en un FPGA Virtex5 son de 6 entradas mientras que en un Spartan 3 son de 4 entradas. También los CLB (Bloque Lógico Configurable) de un Virtex 5 contienen el doble de LUTS y Flip-Flops que un Spartan 3.

La razón de comparar la arquitectura propuesta con un trabajo que se enfoca en lograr el mejor tiempo de ejecución y no al área que consume, es para analizarlo desde el punto de vista de la eficiencia que tiene. Esto es, para saber cuantos bits por segundo procesa cada *slice* del FPGA.

Los resultados obtenidos en esta investigación en cuanto al número de *slices* son mucho mejores que cualquiera de los trabajos mostrados. Por otra parte, es un diseño más lento que los otros trabajos. Sin embargo, se ha demostrado que se puede realizar una implementación mucho más compacta que los trabajos reportados, que aunque más lenta, es más eficiente.

En comparación con [48] su diseño utiliza 3899 *slices*, mientras que los recursos

de área consumidos por la arquitectura de exponenciación realizada en este trabajo es de 300 *slices* para k=16 y 764 para k=32. Así también la eficiencia que presenta la arquitectura propuesta en [48] es de 0.033 kbps/slice, mientras que la arquitectura de exponenciación propuesta en este trabajo tiene una eficiencia de 0.043 kbps/*slice* para k=16 y 0.55 para k=32. Así, la arquitectura de exponenciación desarrollada en esta tesis mejora la eficiencia de la arquitectura reportada en [48] en hasta un 66 %. Por otro lado, la arquitectura aquí propuesta se tarda casi el triple en procesar las misma operación pero es 5 veces más pequeña para k=32. Para k=16 la arquitectura de exponenciación propuesta en esta tesis es 10 veces más lenta que la arquitectura reportada en [48] pero es 13 veces más compacta.

En comparación con uno de los exponenciadores más rápidos reportados en el estado del arte [62], se puede observar que también se tiene una mejor eficiencia. Se realizó una comparación justa para un FPGA Virtex 5. La mejor eficiencia en [62] es para $d = 2$ con 0.124 kbps/*slice* mientras que para este trabajo la mejor eficiencia en el Virtex 5 se logró para k=16 con 148 kbps/*slice*. En cuanto al tiempo de ejecución se puede observar que la arquitectura aquí propuesta es 24 veces más lenta, pero también cabe resaltar que es 28 veces más pequeña que la reportada en [62].

De la misma forma, en el trabajo [70] se muestra un exponenciador modular utilizando el algoritmo *Montgomery Powering Ladder* y el algoritmo de Montgomery para las multiplicaciones parciales. Se reportan los resultados para un Virtex 5, logrando un área de 3218 *slices*, así como un *throughput* de 322.01 Kbps y una eficiencia de 0.100 Kbps/*slice*. En comparación con la arquitectura aquí propuesta para Virtex 5 con $k = 16$ con un área de 141 *slices*, un desempeño de 20.87 Kbps y una eficiencia de 0.148 Kbps/slice, se logró una arquitectura 22 veces más pequeña, siendo apenas 15 veces más lenta y logrando una mejor eficiencia en términos de Kbps/*slice*.

Tabla 5.1: Comparación de trabajos de exponenciación modular en FPGAs.

Trabajo	Metodo	Technologia	Área (*slices*)	Periodo (ns)	#ciclos	avg Cyc (x 1000)	avg T (ms)	Thrg (Kbps)	(Thrg/*slices*)
[48]	MPL[1]	Spartan3	3899	8.4	$(s(14+s)/p) \times size$	946	7.95	128.84	0.033
[39]	RLE[2]	Virtex 2 pro 100	12791	11.97	-	319	3.82	268.06	0.021
[1]	LRE	Altera Stratix EP1s40	341 LEs	5.05	$(n+3)(n+4)(l+p)$	5550	28.03	36.52	0.107
[62](d=2)	MSB[3]	Virtex 5	7303	2.6	$(k/2+w2) \times (ke+2)$	529	1.38	744.6	0.102
[62](d=4)	LSB[4]	Virtex 5	6217	4.5	$(k/4+w4) \times (2 \times ke+3)$	397	1.79	572.5	0.092
[62](d=2)	LSB	Virtex 5	4060	2.6	$(k/2+w2) \times (2 \times ke+3)$	793	2.03	503.6	0.124
[8](r2)	LSB	XC4K	4865	19.2	$(2 \times k+20) \times (2 \times ke)$	2122	40.74	25.1	
[37](4 to 2)	MSB	Virtex 2	26436	10.3	$(k+2) \times (ke+3)$	1054	10.85	94.3	
[70]	MPL	Virtex 5	3218	2.89	$N(N+N/w+g)$	1097	3.18	322.01	0.100
[55](w=16)	RLE	Virtex7	385	2.13	$2n((d+1)(d+1)+(l+1))$	-	9.28	110.34	0.286
[55](w=64)	RLE	Virtex 7	3046	4.97	$2n((d+1)(d+1)+(l+1))$	-	1.52	673.18	0.221
prop.(k=16)	MPL	Spartan3E	300	17.943	$(size/k \times (size/k+2)+2) \times size$	4327	77.65	13.18	0.043
prop.(k=32)	MPL	Spartan3E	764	21.773	$(size/k \times (size/k+2)+2) \times size$	1116	24.30	42.13	0.055
prop.(B)(k=16)[5]	MPL	Spartan3E	306	18.277	$(size/k \times (size/k+1)+2) \times size$	4261	77.89	13.14	0.042
prop.(B)(k=32)	MPL	Spartan3E	767	21.773	$(size/k \times (size/k+1)+2) \times size$	1083	23.58	43.41	0.056
prop.(k=16)	MPL	Virtex 5	141	11.333	$(size/k \times (size/k+2)+2) \times size$	4327	49.04	20.87	0.148
prop.(k=32)	MPL	Virtex 5	368	19.67	$(size/k \times (size/k+2)+2) \times size$	1116	21.95	46.64	0.126
prop.(k=16)	MPL	Spartan 6	83	19.24	$(size/k \times (size/k+2)+2) \times size$	4327	83.25	12.29	0.148
prop.(k=32)	MPL	Spartan 6	188	35.449	$(size/k \times (size/k+2)+2) \times size$	1116	39.56	25.88	0.137
prop.(k=16)	MPL	Virtex 7	76	8.613	$(size/k \times (size/k+2)+2) \times size$	4327	37.27	27.47	0.316
prop.(k=32)	MPL	Virtex 7	188	13.819	$(size/k \times (size/k+2)+2) \times size$	1116	15.42	66.38	0.353

[1]Montgomery Powering Ladder
[2]Right to Left Exponentiation
[3]Most Significant Bit
[4]Left to Right Exponentiation
[5](B) Arquitectura propuesta con BRAMs y que utiliza un tiempo menos en cada ciclo for externo.

Tabla 5.2: Resultados de implementación de exponenciador en el campo $GF(p)$ usando un FPGA Virtex 7.

Trabajo	Num. bits	*Radix*	Área (*slices*)	Frec. (MHz)	Tiempo (ms)	Periodo (ns)	*Throughput* (Kbps)	Eficiencia (Kbps/*slices*)
[55]	512	16	343	458	1.23	2.18	416.26	1.214
[55]	512	32	801	283	0.54	3.53	948.15	1.184
[55]	1024	16	385	468	9.28	2.14	110.34	0.287
[55]	1024	32	1060	485	2.33	2.06	439.48	0.415
[55]	1024	64	3046	201	1.52	4.98	673.68	0.221
[55]	2048	16	553	370	92.25	2.70	22.20	0.040
[55]	2048	32	1092	413	21.03	2.42	97.38	0.089
[55]	2048	64	3558	399	5.68	2.51	360.56	0.101
prop.	512	16	77	116	4.83	8.66	105.99	1.376
prop.	512	32	188	72	2.07	13.96	246.96	1.314
prop.	512	64	533	62	0.68	16.22	751.67	1.410
prop.	1024	16	76	116	37.27	8.61	27.47	0.361
prop.	1024	32	192	72	15.42	13.82	66.40	0.346
prop.	1024	64	529	61	4.88	16.44	209.80	0.397
prop.	2048	16	74	114	297.95	8.74	6.87	0.093
prop.	2048	32	194	73	118.82	13.73	17.24	0.089
prop.	2048	64	513	60	37.28	16.70	54.93	0.107

En [55] se reporta una arquitectura para exponenciación modular utilizando el algoritmo de exponenciación binaria. Para la multiplicación modular se utiliza el algoritmo de multiplicación modular Montgomery junto con el algoritmo de Karatsuba para la multiplicación, beneficiándose de cada una de sus ventajas. En la Tabla 5.2 se muestra una comparación con [55] de los resultados que aquí se obtuvieron para el Virtex 7 para los distintos tamaños de dígito y de operando.

La mejor eficiencia que se logró en [55] para un operando de 512 bits fue de 1.214 Kbps/slice, para 1024 bits de 0.415 Kbps/*slice* y para 2048 bits fue de 0.101 Kbps/*slice*. Para la arquitectura desarrollada en esta tesis, las mejores eficiencias fueron de 1.410, 0.397 y 0.107 Kbps/slice, para los distintos tamaños de operando de 512, 1024 y 2048 bits, respectivamente.

En este caso en particular se observa que la eficiencia es muy parecida en ambas arquitecturas, para los tamaños de operando de 512 y 1024 bits en [55] se tiene ligeramente una mejor eficiencia, mientras que para 2048 bits, la arquitectura aquí propuesta es la que cuenta con una eficiencia ligeramente mejor. En el caso de 1024 bits, en [55] la mejor eficiencia es de 0.415 Kbps/slice, con un área de 1060 *slices*, logrando los cálculos en 2.33 ms. En este trabajo la mejor eficiencia para 1024 bits es de 0.397

Kbps/slice, con un área de 529 *slices* y una latencia de 4.88 ms. Se puede observar que la arquitectura aquí propuesta usa la mitad de *slices*, pero también que necesita aproximadamente el doble de tiempo para su ejecución.

En el caso de operandos 2048 bits, la mejor eficiencia para [55] es de 0.101 Kbps, con un área de 3558 *slices* en un tiempo de 5.68 ms. Por otro lado, la mejor eficiencia para la arquitectura aquí propuesta para el mismo tamaño de operando es de 0.107 Kbps/slice, utilizando un área de 513 *slices* en un tiempo de 37.28 ms. En este caso, la arquitectura propuesta en esta tesis es casi 6.9 veces más pequeña y se tarda 6.6 veces más de tiempo en ejecutar la misma operación.

En la Tabla 5.3 se presentan dos trabajos que implementan algoritmos criptográficos en software. En [45] se implementa una exponenciación modular, la implementación en software se hace utilizando C con MPI (*massively parallel implementation*) y se ejecuta en un *cluster* de cuatro procesadores Xeon de cuatro núcleos cada uno a una frecuencia de reloj de 2.4 GHz cada uno. En este caso, se puede observar que cuatro procesadores Xeon de cuatro núcleos es demasiado *hardware*, a pesar de esto la arquitectura aquí propuesta presenta una mejora de ser 3 veces más rápida en cuanto al tiempo de procesamiento para un spartan 3, y 4 veces para un Virtex 7.

En [2] se reporta una comparación de software y *hardware* de RSA. Esta implementación de software está pensada para WSN (por sus siglas en inglés *wireless sensor network*) y le toma 22.03 segundos cifrar un mensaje con una llave de 1024 bits. Lo que equivale a realizar una exponenciación de 1024 bits.

Tabla 5.3: Comparación de tiempos de ejecución para implementaciones de RSA en software contra la arquitectura *hardware* propuesta en FPGAs.

Trabajo	Imp	Tiempo
[45]	Xeon Processor	68.40 ms
[45]	Xeon Processor	68.90 ms
[2]	Software Para WSN	22.03 seg
prop.	Virtex 7	15.42 ms
prop.	Virtex 5	21.95 ms
prop.	Virtex 5	21.95 ms
prop.	Spartan 3	24.30 ms

5.6.3. Codiseño *hardware-software*

En la Tabla 5.4 se muestran los recursos utilizados por el codiseño *hardware-software*, cabe mencionar que se configuró un procesador Microblaze a una frecuencia de 50

MHz. Y se utilizó un exponenciador modular Montgomery con un tamaño de operando de 1024 bits, y un tamaño de dígito de 16 bits.

Tabla 5.4: Recursos utilizados por el codiseño *hardware-software*.

Recurso	Consumo
Número de Slices	3768
Número de DSP48A1s	11
Frecuencia de operación	50 MHz

5.7. Sumario

En esta sección se presentó la forma de evaluación de la arquitectura propuesta, así como los pasos que se siguieron para llevar la descripción de las arquitecturas a una implementación en FPGAs. También se presentó el codiseño *hardware-software* que implementa el criptosistema de llave pública RSA. Por último se presentaron los resultados obtenidos en estas arquitecturas así como una comparación con los trabajos más representativos del estado del arte para aritmética modular en campos finitos $GF(p)$. En la siguiente sección se presenta un resumen de las contribuciones logradas en esta investigación y el trabajo futuro.

Capítulo 6

Conclusiones

En es sección se presenta una revisión del trabajo realizado, las contribuciones logradas y las líneas de acción para el trabajo futuro.

6.1. Resumen y contribuciones

En este trabajo se presentó una arquitectura compacta para aritmética en campos finitos $\mathbb{GF}(p)$. Se estudiaron diversos algoritmos para multiplicación en campos finitos $\mathbb{GF}(p)$, de los cuales se seleccionó el algoritmo de Montgomery. El algoritmo de Montgomery requiere convertir los números a multiplicar al dominio Montgomery, realizar la multiplicación en este dominio y posteriormente realizar una conversión del resultado al dominio normal, esto para evitar divisiones. Al realizar estas conversiones parece inapropiado utilizar este algoritmo cuando se requiere la máxima velocidad posible, ya que requiere demasiado tiempo para realizar las conversiones. Sin embargo, cuando se requiere un gran número de multiplicaciones como es el caso de una exponenciación modular, las transformaciones solo se realizan al principio de la exponenciación, se calculan todas las multiplicaciones parciales en el dominio Montgomery y solo al final de la exponenciación se realiza la conversión al dominio normal. Esto lo hace un excelente candidato cuando lo que se requiere es un gran número de multiplicaciones con el mismo módulo.

También se presentaron algunos de los algoritmos para exponenciación modular de los cuales el algoritmo de Montgomery Powering Ladder presenta algunas ventajas como es la contramedida de ataques de medición de consumo de potencia como es *SPA*. El algoritmo Montgomery Powering Ladder puede utilizarse en criptosistemas que requieren exponenciación modular en el campo primo $\mathbb{GF}(p)$ como es RSA, DSA, o el intercambio de llaves Diffie-Hellman.

Por otro lado, los dispositivos reconfigurables juegan un rol muy importante al

diseñar arquitecturas *hardware* para diversos algoritmos, entre los cuales se encuentran los algoritmos criptográficos. Estos presentan algunas ventajas contra los tradicionales ASICs. Por ejemplo, se puede implementar el prototipo de algún algoritmo para posteriormente, una vez que esté bien probado y estable, implementarlo en un ASIC.

Las arquitecturas presentadas en este trabajo son para la multiplicación y exponenciación modular en campos finitos $\mathbb{GF}(p)$ para criptografía ligera de llave pública. Los resultados obtenidos muestran que un uso eficiente de los recursos de los FPGAs, como son las BRAMs y los multiplicadores integrados, así como un enfoque iterativo por dígitos [43], permitieron implementar uno de los diseños más compactos para la exponenciación modular.

El diseño de las arquitecturas propuestas es configurable para que se puedan utilizar distintos tamaños de operando, así como también distintos tamaños de dígitos. Esto permite poder utilizar las arquitecturas en dispositivos en los cuales el nivel de seguridad puede variar.

El propósito de esta investigación fue alcanzado al lograr una implementación con 300 y 764 *slices* para un tamaño de dígito de 16 y 32 bits respectivamente en un FPGA Spartan 3E, y 141, 368 para los mismos tamaños de dígito para un Virtex 5. En la comparación con [55], se concluye que ambas arquitecturas logran una eficiencia muy parecida, el área que se utiliza en [55] se compensa con la velocidad que logra. En este caso, el tiempo que se gasta extra se compensa con el área que se ahorra. De esta forma se observa que se pueden implementar diseños compactos que cumplan con el tiempo que se requiere y ofrezca una eficiencia de Kbps/*slice* igual o mejor que las arquitecturas más rápidas. Aunque la arquitectura propuesta está por debajo de las otras en cuanto a tiempo, cabe destacar que sigue siendo mucho más rápida que una implementación en software, por ejemplo es 4 veces más rápida que la implementación en software de [45] y 1450 veces más rápida que la implementación en software para WSN de [2].

Al tener un diseño compacto se paga el precio de un tiempo de ejecución mayor que los reportados en la literatura; aunque, los resultados finales en cuanto a eficiencia, tiempo/área, en la arquitectura propuesta en esta tesis se tiene una mejora significativa, a excepción de [55] en el cual ambas eficiencias son muy similares. Teniendo esto en cuenta, las arquitecturas aquí propuestas pueden servir para implementar medidas de seguridad en dispositivos móviles o embebidos en los cuales se tienen recursos muy limitados.

En resumen, las contribuciones logradas en esta tesis son:

- Una mejora al algoritmo Montgomery iterativo, Algoritmo 18, presentado en [43].

Este nuevo algoritmo reemplaza la operación de corrimiento de bits por el uso de lecturas y escrituras de dígitos, esto nos permite almacenar el resultado final de una multiplicación Montgomery en un bloque de memoria RAM.

- Arquitecturas *hardware* compactas para multiplicación y exponenciación en el campo finito $\mathbb{GF}(p)$.
- Un codiseño *hardware-software* que permite evaluar las arquitecturas propuestas de multiplicación y exponenciación, misma que puede servir de base para la creación de esquemas de cifrado de llave pública (cifrado, firma digital, intercambio de llaves) también bajo el codiseño *hardware-software*.

6.2. Trabajo futuro

Como trabajo futuro se planea implementar algoritmos para aritmética en campos finitos para criptografía de llave pública tratando de mejorar los tiempos de procesamiento en FPGAs. Explorar nuevos algoritmos o proponer implementaciones diferentes de los ya existentes que ocupen menos recursos de área, o menor tiempo de procesamiento, pero que mantengan o mejoren la eficiencia de los algoritmos y sus implementaciones actuales. De igual forma que la aritmética en campos finitos $\mathbb{GF}(p)$ se utiliza para criptosistemas como es RSA, el campo binario extendido $\mathbb{GF}(2^m)$ se utiliza en criptosistemas como es ECC, por lo cual resultaría atractivo explorar implementaciones compactas para aritmética en este campo.

Apéndices

Apéndice A

Vectores de prueba

Para validar los resultados de las arquitecturas propuestas para multiplicación y exponenciación en $\mathbb{GF}(p)$, se crearon algunos vectores de prueba, para operaciones de 1024 bits.

Los vectores de prueba utilizados en la multiplicación modular para números de 1024 bits se muestran en la Tabla A.1. Se muestran varios valores utilizados para M' dependiendo del valor de k utilizado. Para una mejor lectura, los números están en un formato hexadecimal y están separados de 16 en 16 dígitos hexadecimales.

Los vectores de prueba utilizados para validar los resultados de la arquitectura propuesta para exponenciación modular de 1024 bits, así como para el codiseño *hardware-software* se muestran en la Tabla A.2.

Tabla A.1: Vectores de prueba para multiplicación modular Montgomery de 1024 bits.

Variable	Valor		
A	29df4a5222a76d0c	2a046902131ea045	d32593bf6ab39c00
	90a8872bd4f623f5	f40bd541bfe1d5b2	a66fa0a40d2632f2
	99937e8e1942ba26	73b3e0c459bc250b	9316ccbf2f571c74
	07bf7dbfe5922740	399974950d839f6a	b072ce177f8ce294
	6bd9387c37a9f402	ede9d00a81e5efe4	b860ddc20034fa61
	1660c205c209bb0f		
B	18e75f311ce8a845	9d7c39b2298c0f7a	3bccfa6142ac740c
	be9b6d95d4732066	dfe9f62997e09540	34503d7ea784fa33
	2d5dc8fd555e3de4	fe4bfe44f7daebc0	3eded23c5b911a29
	e89bcabc093a3ae0	a1aa3ecddc08e55d	558c845bff1f0150
	3fc1f56a3a18ea38	352d38f857a51cd8	b321322ffd533f63
	9b02a6014aa6f5ef		
M	6e79434fd91645c8	86aa3169c9087918	65815db7fb2104ac
	291c321351c14fba	f746f25c920bfca4	0c808b0c4cbfbf8f
	6c2fd11a2abfda32	b0d8758135775a55	2e70620f92648e47
	07d5c4bcbe77c01a	8f9bbe410dc16833	8cb4cb1d457aeee3
	5e144098393aa78a	8776e06bb7152560	eab8ce7e4a985b41
	c8695a64f3805605		
M′ k=2	3		
M′ k=4	3		
M′ k=8	33		
M′ k=16	f933		
M′ k=32	89f4f933		
M′ k=64	71591bbd89f4f933		
Resultado	3cc165cef0169ffb	7d4b49d85c0efe63	4918ab3f3690efbd
	d6d73a9c8307d679	ee6ed82f3b13a2ba	66f89294fef247a6
	c586e9d520bec148	f1f1e99d2cb68687	82d96ecacb5bc964
	a06b084dd1be57df	e651448ce544a715	bf9a8031f1d116db
	4665e2bbc04ada9c	d66d7fbab71e369b	d569a73ff5499865
	3a04a64d78e83a2c		

Tabla A.2: Vectores de prueba para exponenciación modular Montgomery de 1024 bits.

Variable	Valor
Exp	bc2945615882c6ae bodb97a2f2ef8198 925ed236af93029f 43a47eaf012173f9 1a1768049f439d3d eab616796cd590d6 5419ec44c7a31f99 52d0e79436f10295 27f872464de1d986 15b604c6300027bd 835389ba36ece33a 83ed1765681fde28 ca760e3cf7bfb06c a49abe86de894b0c 6bf9af084b2135b2 7dcb03717dbca25b
X-Montgomery	21f7a071a25589c7 5c5427f78fe91b48 09b3a861075ab60a 8b1de3b37769ce4b 32d39ee5fd5aace3 f2f56057d40709f0 dc62da913ce2196c 4eab7823abab500d 9cec368045972d00 77a6521dabc03e38 599321d70ba0d984 7915930f930815df 526fa7e2a5081c58 3be2fbf41a064cda 2f7d2af6fac6f92f 1c916c6961e88042
M	297fb290f1c0d6ba 23233ac0ba9966c8 743c5e512d7048d5 4d20ac6a62095376 c1cb8793a41bcdaa 67e7a8d388ce74a5 543e0ffd0b1c30c8 e41b59845fd82bb7 dfa825e659024051 c7a73c8596d4f433 53a94179d06f5ec8 594d2d2e00f15815 ba315e513deae6ac 15208ca17693e52b 43acdab17d307840 e7181910fc2b0d87
Uno-Montgomery	701d09a557af7a32 d2c9f7ba067974d4 695ca18ef5e4b003 13bf581b3c80b377 53ad28a27592e019 0920b0acb2944200 68ba011bd56db4aa 75be6e5c0eef9b0c 20f1c99e9f27e155 21494de770246cc0 a0877251d63c74de 830f0ebfa57ef7da 2d7ca188c7e97f78 13cb4373888a0fc6 9f2dfd710dd2e7a9 56f699a16fdaed6
M′ k=2	1
M′ k=4	9
M′ k=8	c9
M′ k=16	d7c9
M′ k=32	3cfad7c9
M′ k=64	743288bd3cfad7c9
Resutlado	156d2e5433480d08 9e3f2f68213ebcc5 8a37f599bcf12c23 ef1adecc43e590b3 dd49433e73aaf9b2 7a0d20bfd46c9ec0 fc3926795e1c9446 e825d45c19f235be 0f6631f026667c1d 927661661d0f102e b646f08e5fd4bab9 51a87689597b20de 46dafe613d46df55 0572ed5d91e05b52 f2ff0f57c03233a0 bb1573761be87c9c

Bibliografía

[1] Alho, T., Hamalainen, P., Hannikainen, M., and Hamalainen, T. D. Design of a compact modular exponentiation accelerator for modern FPGA devices. *Computers & Electrical Engineering 33*, 5 (2007), 383–391.

[2] Alkalbani, A. S., Mantoro, T., and Tap, A. O. M. Comparison between RSA hardware and software implementation for WSNs security schemes. In *Information and Communication Technology for the Muslim World ICT4M, 2010 International Conference on* (2014), IEEE, pp. E84–E89.

[3] Babbage, S., and Dodd, M. The stream cipher MICKEY 2.0. *ECRYPT Stream Cipher* (2006).

[4] Biham, Anderson, and Knudsen. Serpent: A new block cipher proposal. In *IWFSE: International Workshop on Fast Software Encryption, LNCS* (1998), pp. 222–238.

[5] Blakley, G. R. A computer algorithm for calculating the product AB modulo M. *IEEE Transactions on Computers C-32* (1983), 497–500.

[6] Blomer, J., and May, A. Low secret exponent RSA revisited. In *Cryptography and Lattices*. Springer, 2001, pp. 4–19.

[7] Blum, T., and Paar, C. Montgomery modular exponentiation on reconfigurable hardware. In *14th IEEE Symposium on Computer Arithmetic, 1999. Proceedings.* (1999), IEEE, pp. 70–77.

[8] Blum, T., and Paar, C. Montgomery modular exponentiation on reconfigurable hardware. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)* (Los Alamitos, CA, Apr. 1999), Koren and Kornerup, Eds., IEEE Computer Society Press, pp. 70–77.

[9] Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J., Seurin, Y., and Vikkelsoe, C. PRESENT: an ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems-CHES 2007*. Springer, 2007, pp. 450–466.

[10] Boneh, D., and Durfee, G. Cryptanalysis of RSA with private key d less than $n^{0,292}$. *Information Theory, IEEE Transactions on 46*, 4 (2000), 1339–1349.

[11] Boneh, D., and Venkatesan, R. Breaking RSA may not be equivalent to factoring. In *Advances in Cryptology EUROCRYPT'98*. Springer, 1998, pp. 59–71.

[12] Cuevas-Farfan, E., Morales-Sandoval, M., Morales-Reyes, A., Feregrino-Uribe, C., Algredo-Badillo, I., Kitsos, P., and Cumplido, R. Karatsuba-ofman multiplier with integrated modular reduction for GF(2^m). *Advances in Electrical and Computer Engineering* (2013), 3–10.

[13] Daemen, J., and Rijmen, V. AES proposal: Rijndael. NIST AES Proposal, June 1998.

[14] De Canniere, C., and Bart, P. Trivium: A stream cipher construction inspired by block cipher design principles. In *Information Security*. Springer, 2006, pp. 171–186.

[15] De Canniere, C., and Dunkelman, Orr, M. KATAN and KTANTAN family of small and efficient hardware-oriented block ciphers. In *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 2009, pp. 272–288.

[16] Diffie, W., and Hellman, M. New directions in cryptography. *IEEE Transactions on Information Theory 22*, 6 (Nov. 1976), 644–654.

[17] Downey, Leong, and Sethi. Computing sequences with addition chains. *SICOMP: SIAM Journal on Computing 10* (1981).

[18] Eisenbarth, T., and Kumar, S. A survey of lightweight-cryptography implementations. *Design & Test of Computers, IEEE 24*, 6 (2007), 522–533.

[19] ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEETIT: IEEE Transactions on Information Theory 31* (1985), 10–18.

[20] Francisco Rodriguez Henriquez N.A. Saqib, Arturo Díaz Pérez, C. K. K. *Cryptographic Algorithms on Reconfigurable Hardware, (Signals and Communication Technology)*, november 14, 2006 ed. Springer, 2007.

[21] Gaj, K., Kaps, J.-P., Amirineni, V., Rogawski, M., Homsirikamol, E., and Brewster, B. Y. ATHENa - Automated Tool for Hardware EvaluatioN: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs. In *FPL* (2010), IEEE, pp. 414–421.

[22] Galbraith, P. S. D. *Mathematics of Public Key Cryptography*. Cambridge University Press, April 2012.

[23] Hermans, J. *Lightweight Public Key Cryptography*. PhD thesis, Department of Electrical Engineering (ESAT), Faculty of Engineering, Arenberg Doctoral School of Science, Engineering & Technology, August 2012.

[24] Hoffstein, J., Pipher, J. C., and Silverman, J. H. *An Introduction to Mathematical Cryptography*. Undergraduate texts in mathematics. Springer-Verlag, pub-SV:adr, 2008.

[25] Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B.-S., Lee, C., Chang, D., Lee, J., and Jeong, K. HIGHT: a new block cipher suitable for low-resource device. In *Cryptographic Hardware and Embedded Systems-CHES 2006*. Springer, 2006, pp. 46–59.

[26] Joye, M., and Yen, S.-M. The Montgomery powering ladder. In *CHES* (2002), B. S. K. Jr., Ç. K. Koç, and C. Paar, Eds., vol. 2523 of *Lecture Notes in Computer Science*, Springer, pp. 291–302.

[27] Karatsuba, A., and Offman, Y. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady 7* (1963), 595–596.

[28] Katagi, M., and Moriai, S. Lightweight cryptography for the internet of things. *Sony Corporation* (2008).

[29] Knuth. *The Art of Computer Programming, Vol. 2*. Addison-Wesley, Reading, 1981.

[30] Koblitz, N. Elliptic curve cryptosystems. *Mathematics of Computation 48*, 177 (1987), 203–209.

[31] Koç, Ç. K. High-speed RSA implementation. Technical report TR201, RSA Data Security, Inc., pub-RSA:adr, Nov. 1994. Version 2.0.

[32] Koc, C. K., Acar, G., and Kaliski, Jr., B. S. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro* (June 1996). 26–33.

[33] Liu, Z., Xia, L., Jing, J., and 0005, P. L. A tiny RSA coprocessor based on optimized systolic Montgomery architecture. In *SECRYPT* (2011), J. Lopez and P. Samarati, Eds., SciTePress, pp. 105–113.

[34] Manochehri, K., and Pourmozafari, S. Modified radix-2 Montgomery modular multiplication to make it faster and simpler. In *International Conference on Information Technology: Coding and Computing, 2005. ITCC 2005.*, vol. 1, IEEE, pp. 598–602.

[35] Matsuda, S., and Moriai, S. Lightweight cryptography for the cloud: Exploit the power of bitslice implementation. In *CHES* (2012), E. Prouff and P. Schaumont, Eds., vol. 7428 of *Lecture Notes in Computer Science*, Springer, pp. 408–425.

[36] Matsuda, S., and Moriai, S. Lightweight cryptography for the cloud: exploit the power of bitslice implementation. In *Cryptographic Hardware and Embedded Systems CHES 2012*. Springer, 2012, pp. 408–425.

[37] McIvor, C., McLoone, M., and McCanny, J. Modified Montgomery modular multiplication and RSA exponentiation techniques. *IEE Proceedings - Computers and Digital Techniques 151*, 6 (2004), 402.

[38] McIvor, C., McLoone, M., and McCanny, J. V. FPGA Montgomery multiplier architectures - A comparison. In *FCCM* (2004), IEEE Computer Society, pp. 279–282.

[39] Michalski, E. A., and Buell, D. A. A scalable architecture for RSA cryptography on large FPGAs. In *FPL'06. International Conference on Field Programmable Logic and Applications, 2006.* (2006), pp. 1–8.

[40] Miller. Use of elliptic curves in cryptography. In *CRYPTO: Proceedings of Crypto* (1985), pp. 417–426.

[41] Montgomery. Speeding the pollard and elliptic curve methods of factorization. *MATHCOMP: Mathematics of Computation 48* (1987).

[42] Montgomery, P. Modular multiplication without trial division. *Mathematics of Computation 44*, 170 (1985), 519–521.

[43] Morales-Sandoval, M., and Diaz-Perez, A. A compact FPGA-based montgomery multiplier over prime fields. In *Proceedings of the 23rd ACM international conference on Great lakes symposium on VLSI* (2013), pp. 245–250.

[44] Morales-Sandoval, M., Uribe, C. F., Kitsos, P., and Cumplido, R. Area/performance trade-off analysis of an FPGA digit-serial GF(2^m) montgomery multiplier based on LFSR. *Computers & Electrical Engineering 39*, 2 (2013), 542–549.

[45] Mourelle, L., Raposo, S., Nedjah, N., and Santana, M. Massively parallel modular exponentiation method and its implementation in software and hardware for high-performance cryptographic systems. *IET Computers & Digital Techniques 6*, 5 (2012), 290–301.

[46] National Institute of Standards and Technology (NIST). The Digital Signature Standard (DSS). Federal Information Processing Standards Publication (FIPS PUB) 186, Aug. 1991. Draft.

[47] Nedjah, N., and de Macedo Mourelle, L. Fast reconfigurable systolic hardware for modular multiplication and exponentiation. *Journal of Systems Architecture 49*, 7 (2003), 387–396.

[48] Oksuzoglu, E., and Savas, E. Parametric, secure and compact implementation of RSA on FPGA. In *Reconfigurable Computing and FPGAs, 2008. ReConFig'08. International Conference on* (2008), pp. 391–396.

[49] Oksuzoglu, E., and Savas, E. Parametric, secure and compact implementation of RSA on FPGA. IEEE, pp. 391–396.

[50] Paar, C., and Weimerskirch, A. Embedded security in a pervasive world. *Inf. Sec. Techn. Report 12*, 3 (2007), 155–161.

[51] Panasenko, S., and Smagin, S. Lightweight cryptography: Underlying principles and approaches. *International Journal of Computer Theory and Engineering 3*, 4 (August 2011).

[52] Poschmann, A. Y. *Lightweight cryptography: cryptographic engineering for a pervasive world*. PhD thesis, Ruhr University Bochum, 2009. http://d-nb.info/996578153.

[53] Publication, F. I. P. S. U.S. Department of Commerce/ National Institute of Standards and Technology,"Digital Signature Standard (DSS)"FIPS PUB 186-2(Federal Information Processing Standards Publication), Juanary 2000.

[54] Rivest, R. L., Shamir, A., and Adelman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.

[55] San, I., and At, N. Improving the computational efficiency of modular operations for embedded systems. *Journal of Systems Architecture* (Oct. 2013).

[56] Schneier, B. Description of a new variable-length key, 64-bit block cipher (Blowfish). *Lecture Notes in Computer Science 809* (1994), 191–204.

[57] Sojka, A., Piotrowski, K., and Langendoerfer, P. Short ECC: a lightweight security approach for wireless sensor networks. In *Proceedings of the 2010 International Conference on Security and Cryptography SECRYPT* (2010), pp. 1–5.

[58] Stajano, F. *Security for Ubiquitous Computing*. Wiley Series on Communications Networking & Distributed Systems (Book 1). Wiley, June 2002.

[59] Stallings, W. *Cryptography and Network Security: Principles and Practice (5th Edition)*, 5 ed. Prentice Hall, 1 2010.

[60] Standards, N. B. Data Encryption Standard. Tech. rep., NBS, 1977.

[61] Sun, M.-C., Su, C.-P., Huang, C.-T., and Wu, C.-W. Design of a scalable RSA and ECC crypto-processor. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference, ASP-DAC '03, Kitakyushu, Japan, January 21-24, 2003* (2003), H. Yasuura, Ed., ACM, pp. 495–498.

[62] Sutter, G. D., Deschamps, J.-P., and Imana, J. L. Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation. *IEEE Transactions on Industrial Electronics 58*, 7 (July 2011), 3101–3109.

[63] Swenson, C. *Modern cryptanalysis - techniques for advanced code breaking*. Wiley, 2008.

[64] Tenca, A. F., and Koç, Ç. K. A scalable architecture for montgomery multiplication. In *CHES* (1999), Ç. K. Koç and C. Paar, Eds., vol. 1717 of *Lecture Notes in Computer Science*, Springer, pp. 94–108.

[65] van Deursen, T., and Radomirovic, S. Attacks on RFID protocols. *IACR Cryptology ePrint Archive 2008* (2008), 310.

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

[67] Walter, C. D. Montgomery's multiplication technique: How to make it smaller and faster. In *CHES* (1999), Ç. K. Koç and C. Paar, Eds., vol. 1717 of *Lecture Notes in Computer Science*, Springer, pp. 80–93.

[68] Weiser, M. Hot topics-ubiquitous computing. *Computer 26*, 10 (Oct. 1993), 71–2.

[69] Wiener, M. J. Cryptanalysis of short RSA secret exponents. *Information Theory, IEEE Transactions on 36*, 3 (1990), 553–558.

[70] Wu, T., Li, S., and Liu, L. Fast, compact and symmetric modular exponentiation architecture by common-multiplicand Montgomery modular multiplications. *Integration, the VLSI Journal 46*, 4 (Sept. 2013), 323–332.

[71] Yalla, P., and Kaps, J.-P. Lightweight cryptography for FPGAs. In *ReConFig'09: 2009 International Conference on Reconfigurable Computing and FPGAs, Cancun, Quintana Roo, Mexico, 9-11 December 2009, Proceedings* (2009), V. K. Prasanna, L. Torres, and R. Cumplido, Eds., IEEE Computer Society, pp. 225–230.

[72] Zhang, Y.-Y., Li, Z., Yang, L., and Zhang, S.-W. An efficient CSA architecture for montgomery modular multiplication. *Microprocessors and Microsystems 31*, 7 (2007), 456–459.