Universidad Autónoma Metropolitana
Unidad Azcapotzalco

División de Ciencias Básicas e Ingeniería

Proyecto Terminal en Ingeniería en Computación

Diseño e Implementación de un Co-Procesador Matemático de Aritmética Entera basado en un FPGA

Proyecto que presenta:
Joel Noyola Bautista

para obtener el titulo de:
Ingeniero en Computación

Director de Proyecto:

Dr. Felipe Monroy Pérez M. en C. Oscar Alvarado Nava

México, D.F. **Diciembre de 2011**

Resumen

En los sistemas criptográficos modernos de llave-pública se requieren cálculos que hacen uso intensivo de los recursos computacionales. Realizar de manera eficiente todos esos cálculos, incluyendo la multiplicación y exponenciación modular son el centro de estudio en las actividades de criptografía. Las aplicaciones criptográficas pueden ser implementadas en ambos software y hardware. Las implementaciones en software son diseñadas y codificadas en lenguajes de programación de alto nivel como C, C++ o algún lenguaje ensamblador, para ser ejecutados en procesadores de propósito general. Las implementaciones en hardware son diseñadas y codificadas en lenguajes de descripción de hardware, tales como VHDL y Verilog HDL; con VHDL, por ejemplo, su objetivo principal es el de implementarse en FPGAs (Field Programmable Gate Array) por sus siglas en inglés.

El coprocesador de aritmética entera trabaja con números enteros muy largos, los cuales no pueden ser representados en procesadores de propósito general, más específicamente, su tarea principal es cálcular la multiplicación modular a través del algoritmo de Montgomery. Para entender la multiplicación modular se necesitan bases en la teoría de números. En el capítulo 2 trata acerca de los conceptos básicos que se necesitan de la teoría de números para entender el algoritmo de Montgomery para multiplicación modular, éste se puede consultar en el capítulo 3.

Se incluye en el apéndice, en la sección de códigos fuente, la implementación del coprocesador en software y hardware. La implementación en software está en Maple y para la versión hardware se implementó en VHDL.

Agradecimientos

- A la División de Ciencias Básicas e Ingeniería de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco.

Dedicatoria

A el hombre más hermoso de los hijos de los hombres.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivaciones

Muchos algoritmos criptográficos confian en el cálculo intensivo de operaciones, tales como la multiplicación modular de enteros muy largos. Tipicamente en un rango de 512 y 2048 bits. Esto implica que los operandos no pueden ser procesados directamente en la CPU con registros de 32 o 64 bits. Por lo tanto, generalmente los enteros largos, son representados por arreglos de palabras de precisión simple (enteros sin signo de 32 o 64 bits). En las claves públicas criptográficas, el residuo de la operación *mod*, generalmente es realizado por un primo o un producto de primos. Una multiplicación modular, combina la multiplicación de enteros muy largos y el residuo del producto modular en una simple operación de la forma $c = a \cdot b \ mod \ n$.
El algoritmo de Montgomery para la multiplicación modular es considerado el algoritmo más rápido para calcular $a \cdot b \ mod \ n$ en computadoras donde a, b y n son enteros muy largos[9].

A través de dispositivos programables, como los FPGA's, es posible implementar circuitos que lleven a cabo el cálculo de operaciones aritméticas de manera eficiente e independiente, ya que pueden aprovechar el paralelismo inherente del hardware. Este trabajo consiste en desarrollar un corocesador matemático de aritmética entera, para números enteros muy largos, precisamente, para acelerar aplicaciones criptográficas.

Particularmente, se hará un operador de multiplicación modular a través del algortimo de Montgomery. Esto nos ayudará a disminuir el tiempo de procesamiento de las aplicaciones criptográficas, ya que el procesamiento será de manera independiente por el coprocesador implementado en el FPGA.

1.2. Objetivos

Diseñar e implementar un coprocesador matemático de aritmética entera, basado en un FPGA, el cual será ocupado como un periférico coprocesador, de baja velocidad, sobre un FPGA XC2VP30 de la familia Virtex II Pro; con el fin de acelerar el proceso de calcular la multiplicación modular a nivel de hardware, sin que se vea afectada la exactitud ofrecida por los lenguajes de alto nivel.

1.2.1. Objetivos Particulares

- **Estudiar el algoritmo de multiplicación de Montgomery.** Es importante entender bien cómo funciona el algoritmo de Montgomery para la multiplicación modular para implementarlo de manera eficiente como un módulo coprocesador.
- **Implementar y analizar el algoritmo de Montgomery para la multiplicación modular en Maple.** Un lenguaje de alto nivel, como Maple, permite realizar cálculos de tipo científico, además Maple te permite trabajar con números que sobrepasan la precisión simple o doble de los procesadores de propósito general.
- **Desarrollar un módulo coprocesador en hardware para el operador de multiplicación modular.** El algoritmo de Montgomery para la multiplicación modular es útil porque trabaja con un número r que es potencia de dos, esto permite que las multiplicaciones y divisiones sean hechas mucho más rápido a través de desplazamientos y corrimientos y operaciones lógicas.
- **Integrar el coprocesador en una sistema mínimo.** Comenzar a implementar el circuito en el FPGA.
- **Determinar validez de resultados.** Determinar si los resultados obtenidos por el coprocesador son los correctos y verificar si el tiempo de ejecución, para obtener el resultado, se acelera.
- **Desarrollar interfaz hardware-software.** Son las adecuaciones tanto en la parte hardware como en la parte software para que haya comunicación y coordinación.
- **Realizar pruebas software vs hardware.** Comparar la el tiempo de ejecución del coprocesador en su versión software con su versión hardware.

1.3. Organización del proyecto

El proyecto está dividido en varias secciones. En la primera de ellas se habla acerca de los conceptos básicos de la teoría de números; se tratan temas desde un punto de vista más matemático, como los números enteros, divisibilidad, los números primos,

el algoritmo de euclides, relaciones de equivalencia, aritmética modular, etc. Todos estos temas, antes menciondos forman una parte fundamental para entender el algoritmo de Montgomery de la multiplicación modular.

En la segunda sección se explica el algoritmo de Montgomery para la multiplicación modular, y se describen algunos pseudocódigos para implentar el algoritmo en software y hardware.

La tercera sección es la descripción en hardware del algoritmo de Montgomery, se detallan los módulos que integran el coprocesador, así como el funcionamiento de cada uno de los módulos. Se Presenta la arquitectura que se diseñó e implementó para el coprocesador y la máquinda de estados que lo representa.

La cuarta sección está dedicada a los resultados y conclusiones durante la elaboración y culminación de este proyecto. Además integra la simulación del coprocesador donde se puede observars el funcionamiento del algoritmo de Montgomery, calculando la multiplicación modular. Finalmente, la quinta sección, es un apéndice, éste apéndice contiene los códigos fuente de las implementaciones del algortimo de Montgomery tanto software como hardware.

Capítulo 2

Conceptos básicos de la Teoría de números

La criptografía ha sido muy importante a través de los años para el envío de información secreta. Hoy en día las computadoras y el internet han hecho de la criptografía parte de nuestras vidas. En los sistemas criptográficos modernos, los mensajes son representados por valores numéricos antes de ser codificados y transmitidos. Los procesos de codificación son operaciones matemáticas que convierten los valores numéricos de entrada en valores numéricos de salida. Construyendo y analizando ésos sistemas criptográficos se requieren herramientas matemáticas. Una de la más importantes es la Teoría de Números. Este capítulo presenta las herramientas básicas, de dicha teoría, necesarias para entender el algoritmo de Montgomery.

2.1. Los números enteros

Las ecuaciones de la forma $a = b + x$ no siempre tienen solución en el conjunto de los números naturales $\mathbb{N}$ (no siempre pueden resolverse).
Ejemplo. La ecuación $x + 1 = 0$. No tiene solución en $\mathbb{N}$.

Con el objeto de evitar esta limitación, se extiende el conjunto $\mathbb{N}$ a otro, en el que todas las ecuaciones de la forma anterior pueden resolverse y que contenga a $\mathbb{N}$. Se extiende así el conjunto $\mathbb{Z}$ de los números enteros, con un subconjunto a los números naturales, es decir; los números naturales, son un subconjunto de los números enteros ($\mathbb{N} \subseteq \mathbb{Z}$). Los números enteros son fundamentales para la aritmética y la teoría de números.

2.2. Divisibilidad

La teoría de números se refiere a las propiedades de los números enteros. Una de las más importantes es la divisibilidad.

Definición. *Dados a y b enteros con $a \neq 0$. Decimos que a* **divide** a *b, si existe un entero k tal que $b = ak$. Esto es denotado por $a \mid b$. Otra forma de decir esto es que b es múltiplo de a.*[1][2]

Ejemplos.
$3 \mid 15 \Rightarrow 15 = 3 \cdot 5$.
$-15 \mid 60 \Rightarrow 60 = (-15) \cdot (-4)$.
$7 \nmid 18$.

El símbolo $\nmid$ significa que 7 no divide a 18, es decir; no existe k en los números enteros ($k \in \mathbb{Z}$) tal que $18 = 7 \cdot k$.

Se tienen las siguientes propiedades de divisibilidad.

1. *Para cada $a \neq 0$, $a \mid 0$ y $a \mid a$. También para cada b, $1 \mid b$.*
2. *Si $a \mid b$ y $b \mid c$, entonces $a \mid c$.*
3. *Si $a \mid b$ y $a \mid c$, entonces $a \mid (sb + tc)$ para todos los enteros s y t.*

La propiedad 1 es evidente de la definición. La propiedad 2 se puede mostrar, utilzando la definición, escribiendo $b = a \cdot k$ y $c = b \cdot l$ por lo que $c = a \cdot (k \cdot l)$ de donde se concluye que $a \mid c$.
Finalmente si escribimos $b = a \cdot k_1$ y $c = a \cdot k_2$, entonces $s \cdot b + t \cdot c = a(sk_1 + tk_2)$, por lo tanto $a \mid sb + tc$, probamos que la propiedad 3 es cierta.

2.3. Números Primos

Un número $p > 1$ que solo es divisible por 1 y por él mismo es llamado un **número primo**[1] [1] [2]. Existen algunas técnicas, en la teoría de números, diseñadas para contar el tamaño de algún subconjunto de los números enteros, en este caso el conjunto de los números primos menores o iguales n. Para mostrar esto podemos dar un ejemplo: la **criba de Eratóstenes**.

Un número entero $n > 1$ que no es un número primo se llama **compuesto** lo que significa que n debe ser expresado como un producto ab de enteros, con $1 < a, b < n$.

La criba de Eratóstenes es un algoritmo que permite hallar todos los números primos menores que un número natural dado n. Se forma una tabla con todos los números naturales comprendidos entre 2 y n y se van tachando los números que no son primos de la siguiente manera: cuando se encuentra un número compuesto que no ha sido tachado, ese número es declarado primo, y se procede a tachar todos sus múltiplos.

[1]Euclides probó que existe un número infinito de primos.

El proceso termina cuando el cuadrado del mayor número confirmado como primo es mayor que n.

Pseudocódigo para la criba de Eratóstenes.

Algoritmo 1 Criba de Eratóstenes

Entrada: Un número natural n.
Salida: Los números primos anteriores a n (incluyendo n si n es primo).

```
1: Escribir todos los números naturales desde 2 hasta n.
2: para i = 2 hasta [√n] hacer
3:    si  i no ha sido marcado entonces
4:       para j = i hasta n ÷ i hacer
5:          Ponga una marca en i × j
6:       fin para
7:    fin si
8: fin para
```

Notacion:

$[n]$. Es la parte entera de n.
$n \div i$. Es el cociente de dividir n entre i.

Ejemplo. Utilizando la criba de Eratóstenes podemos saber cuáles son los números primos que hay de 2 hasta $n = 120$. El Cuadro 1 muestra cómo, utilizando un color con cada iteración, funciona el algoritmo arriba descrito. Entonces para este ejemplo, los números primos que hay de 2 hasta $n = 120$ son: $2, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101,\ 103, 107, 109, 113$.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Tabla 2.1: Criba de Eratóstenes para $n = 120$.

Proposición. *Todo entero positivo es un producto de primos, esta factorización es única.*[1][2]

Para mostrar que esta propiedad es cierta tomémos a $n > 1 \in \mathbb{Z}$. Si n es un número primo, entonces es evidente que la proposición es verdadera.
Si n no es un número primo, entonces en su descomposición tiene un número primo $n = p_1 n_1$, análogamente $n_1 = p_2 n_2$ donde podemos observar que $n_1 < n$ y $n_2 < n_1$. Si continuamos este proceso, entonces tenemos $n_1 > n_2 > n_3 > \cdots > n_k$, por lo tanto $n = p_1 p_2 p_3 \cdots n_{k+1}$. El proceso termina pues de p_i, $n_i < n$. Por lo tanto el último factor es un número primo y $n = p_1 p_2 \cdots p_r$. Como se podrán dar cuenta, solamente se ha probado la existencia de la proposición, para mostar que es única es un camino largo, por lo que se omitirá, si el lector desea saber más de esto puede consultar en la bibliografía sugerida[1]. A esta proposición se lo conoce como el **Teorema fundamental de la Aritmética**.

Ejemplos.

220	2
110	2
55	5
11	11
1	

Ejemplo 1: $220 = 2 \cdot 2 \cdot 5 \cdot 11 \cdot 1$.

180	2
90	2
45	3
15	3
5	5
1	

Ejemplo 2: $180 = 2 \cdot 2 \cdot 3 \cdot 3 \cdot 5 \cdot 1$.

2.4. Máximo Común Divisor

El **máximo común divisor** de a y b es el entero positivo más grande que divide a ambos a, b y es denotado por $mcd(a, b)$.

Ejemplos. $mcd(6, 4) = 2$, $mcd(5, 7) = 1$, $mcd(24, 60) = 12$

Decimos que a y b son **primos relativos** si $mcd(a, b) = 1$. Hay dos formas de encontrar el mcd:

1. Si se puede factorizar a y b en primos, háganlo. Para cada número primo, observar las potencias que aparecen en las factorizaciones de a y b y tomar la más pequeña de las dos. Poner esas potencias de primos juntas para obtener el mcd. Esto es fácil de entender con un ejemplo.

   Ejemplos. $1728 = 2^6 \cdot 3^2$, $135 = 3^3 \cdot 5$, $mcd(1728, 135) = 3^2 = 9$
   $mcd(2^5 \cdot 3^4 \cdot 7^2, 2^2 \cdot 5^3 \cdot 7) = 2^2 \cdot 3^0 \cdot 5^0 \cdot 7^1 = 2^2 \cdot 7 = 28$.

2. Suponer que a y b son números grandes. Entonces, factorizarlos no debería ser una tarea fácil. El mcd puede ser calculado por un procedimiento conocido como el **Algoritmo de Euclides**.

Ejemplos. Calcular $mcd(482, 1180)$.
Solución: Dividimos 428 en 1180. El cociente es 2 y el residuo es 216. Ahora dividimos el residuo 216 en 482. El cociente es 2 y el residuo es 50. Dividimos el residuo 50 en el residuo anterior 216. El cociente es 4 y el residuo es 16. Continuamos este procedimiento de dividir el residuo más reciente en el residuo anterior. El último residuo mayor que cero es el mcd, el cual es 2 en este caso:

$$1180 = 2 \cdot 482 + 216$$

$$482 = 2 \cdot 216 + 50$$

$$216 = 4 \cdot 50 + 16$$

$$50 = 3 \cdot 16 + 2$$

$$16 = 8 \cdot 2 + 0$$

Notar cómo los números son desplazados:

$$Residuo \longrightarrow divisor \longrightarrow dividendo \longrightarrow ignorar.$$

Aquí está otro ejemplo:

$$12345 = 1 \cdot 1111 + 1234$$

$$11111 = 9 \cdot 1234 + 5$$

$$1234 = 246 \cdot 5 + 4$$

$$5 = 1 \cdot 4 + 1$$
$$4 = 4 \cdot 1 + 0$$

Por lo tanto, el $mcd(12345, 11111) = 1$.

Usando estos ejemplos como guía, podemos podemos dar una descripción más formal del **Algoritmo de Euclides**.

2.5. Algoritmo de Euclides

Supongamos que a es más grande que b. Si no, conmutamos a y b. El primer paso es dividir a por b, por lo tanto a está representado de la forma:

$$a = q_1 \cdot b + r_1.$$

Si $r_1 = 0$, entonces $b \mid a$ y el máximo común divisor es b. Si $r_1 \neq 0$, entonces continuamos representado a b en la forma:

$$b = q_2 \cdot r_1 + r_2.$$

Continuamos este camino, ayuda a encontrar el último residuo mayor que cero, con la siguiente secuencia de pasos:

$$a = q_1 \cdot b + r_1$$
$$b = q_2 \cdot r_1 + r_2$$
$$r_1 = q_3 \cdot r_2 + r_3$$
$$\cdots$$
$$\cdots$$
$$\cdots$$
$$r_{k-2} = q_k \cdot r_{k-1} + r_k$$
$$r_{k-1} = q_{k+1} \cdot r_k.$$

La conclusión es que $mcd(a, b) = r_k$.[1][2]

Pseudocódigo para el algoritmo de Euclides.

En la Figura 1 se muestra el diagrama de flujo que corresponde al algoritmo de Euclides. Hay dos aspectos importantes en este algoritmo:

1. No requiere la factorización de los números.
2. Si se desea implementar en algún lenguaje de programación, el algoritmo es fácil de programar. Y es más rápido si se tratara de implementar de manera recursiva.

Algoritmo 2 Algoritmo de Euclides

Entrada: Números enteros a y b.
Salida: El máximo común divisor de a y b (mcd).

1: $r_0 \longleftarrow a$, $r_1 \longleftarrow b$.
2: $i \longleftarrow 1$.
3: **mientras** $r_i \neq 0$ **hacer**
4: $\quad r_{i+1} \longleftarrow r_{i-1} mod r$.
5: $\quad i \longleftarrow i + 1$.
6: **fin mientras**
7: **devolver** r_{i-1}.

Proposición. *Dados dos enteros a y b, con a o b distintos de cero, y dado $d = mcd(a, b)$. Entonces existen enteros x, y tales que $a \cdot x + b \cdot y = d$. En particular, si a y b son primos relativos, entonces existen enteros x, y con $a \cdot x + b \cdot y = 1$.*

En general, mostraremos que si r_j es el residuo que se obtiene a partir del algoritmo de Euclides, entonces existen números enteros x_j, y_j, tales que $r_j = ax_j + by_j$. Empezamos con $j = 1$. Tomando $x_1 = 1$ y $y_1 = -q_1$, encontramos que $r_1 = ax_1 + by_1$. Similarmente $r_2 = a(-q_2) + b(1 + q_1 q_2)$. Ahora supongamos que tenemos $r_i = ax_i + by_i$ para toda $i < j$. Entonces

$$r_j = r_{j-2} - q_j r_{j-1} = ax_{j-2} + bx_{j-2} - q_j(ax_{j-1} + by_{j-1}).$$

Lo cual implica que

$$r_j = a(x_{j-2} - q_j x_{j-1}) + b(x_{j-2} - q_j y_{j-1}).$$

Continuando, obtenemos el resultado para toda j, en particular para $j = k$. Por lo que $r_k = mcd(a, b)$ como se dijo en proposición la anterior.
Como consecuencia de esta proposición se deduce otra proposición que está relacionada con la prueba del Teorema Fundamental de la Aritmética.

Proposición. *Si p es un primo, y p divide a un producto de enteros ab, entonces $p \mid a$ o $p \mid b$. Más generalmente, si un primo p divide a un producto $a \cdot b \cdots z$, entonces p deberá dividir a uno de los factores $a \cdot b \cdots z$.*

Para poder hacer evidente esta proposición, primero, vamos a trabajar con el caso $p \mid ab$. Si $p \mid a$, hemos terminado. Ahora asuminos que $p \nmid a$. Afirmamos que $p \mid b$. Sabemos que si p es un número primo, entonces $mcd(a, p) = 1$ ó igual a p. El hecho de que $p \nmid a$, el mcd no puede ser p. Por lo tanto, $mcd(a, p) = 1$, por lo que existen x, y números enteros con $ax + py = 1$. Multiplicamos por b y obtenemos $abx + pby = b$. Donde $p \mid ab$ y $p \mid p$, tenemos $p \mid abx + pby$, entonces $p \mid b$, como habíamos afirmado.

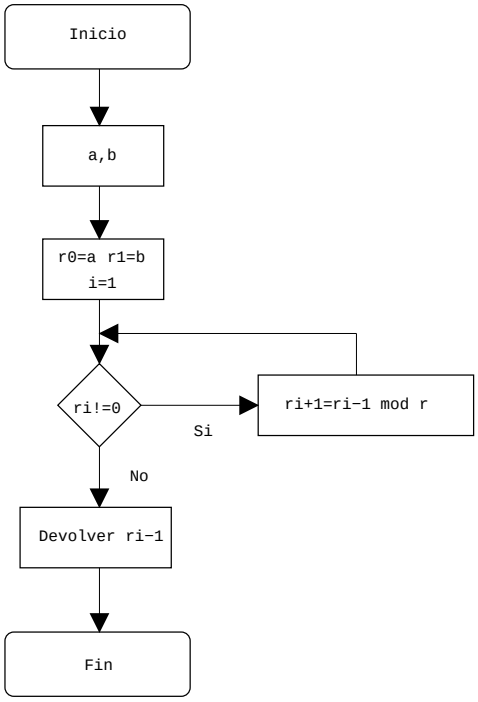

Figura 2.1: Diagrama correspondiente al algoritmo de Euclides.

Si $p \mid ab \cdots z$, entonces $p \mid a$ ó $p \mid b \cdots z$. Si $p \mid a$ terminamos.De otro modo, $p \mid b \cdots z$. Ahora tenemos un producto más pequeño. También $p \mid b$, el cual, en este caso hemos terminado, o p divide al producto de los demás factores. Continuando con este camino, eventualmente, encontraremos que p divide a uno de los factores del producto.

La propiedad de los números primos escrita en la proposición anterior es válida sólo para números primos. Por ejemplo, si sabemos que un producto ab es divisible por 6, no podemos concluir que a o b es un múltiplo de 6. El problema es que $6 = 2 \cdot 3$, el 2 podría ser a, mientras que 3 podría ser b, como se puede ver en el ejemplo: $60 = 4 \cdot 15$. Más generalmente, si $n = ab$ es algún número compuesto, entonces $n \mid ab$, pero $n \nmid a$ y $n \nmid b$. Por lo tanto, los números primos son los únicos números enteros, los cuales cumplen la propiedad de la proposición.

2.6. Algoritmo Extendido de Euclides

Resolviendo $a \cdot x + b \cdot y = d$

Recordemos que en el algoritmo de Euclides, no utilizamos a los cocientes. En esta parte, veremos cómo utilizarlos. Un hecho muy básico, que definimos anteriormente, es que, dados dos enteros a y b, hay dos enteros x y y tales que

$$a \cdot x + b \cdot y = mcd(a, b)$$

¿Cómo encontramos a x y y? Supongamos que empezamos dividiendo a en b, así $b = q_1 \cdot a + r_1$, entonces procedemos como en el algoritmo de Euclides. Sea la suceción de cocientes $q_1, q_2, ...q_n$, Así, en el primer ejemplo de la Sección 1.3, tenemos $q_1 = 2, q_2 = 2, q_3 = 4, q_4 = 3, q_5 = 8$. Estableciendo las siguientes secuencias:

$$x_0 = 0, x_1 = 1, x_j = -q_{j-1} \cdot x_{j-1} + x_{j-2}$$

$$y_0 = 0, y_1 = 0, y_j = -q_{j-1} \cdot y_{j-1} + y_{j-2}$$

Entonces

$$a \cdot x_n + b \cdot y_n = mcd(a, b)$$

En el primer ejemplo, tenemos el siguiente cálculo:

$$x_0 = 0, x_1 = 1$$

$$x_2 = -2x_1 + x_0 = -2$$

$$x_3 = -2x_2 + x_1 = 5$$

$$x_4 = -4x_3 + x_2 = -22$$

$$x_5 = -3x_4 + x_3 = 71$$

Similarmente, calculamos $y_5 = -29$. Por lo tanto tenemos que

$$482 \cdot 71 + 1180 \cdot (-29) = 2 = mcd(482, 1180)$$

Notar que no usamos el último cociente. Si nosotros usaramos éste, deberímos calcular $x_{n+1} = 590$, el cual es 1180 divido por el *mcd*. De igual forma, $y_{n+1} = 241$, el cual es 482/2.
Al método anterior a menudo se le conoce como el **Algoritmo extendido de Euclides**[1][2]. Este algoritmo se usará para resolver **Congruencias**[1]. Para números pequeños, existe otra manera de encontrar x y y, pero en este trabajo se presenta un algoritmo para trabajar con números muy grandes, por este motivo se omitirá dicho método.

Pseudocódigo para el algoritmo de extendido de Euclides.

En la Figura 2 se muestra el diagrama de flujo para el el algortimo extendido de Euclides.

2.7. Congruencias

Salta a la vista, con lo tratado hasta este momento que la divisibilidad es un concepto fundamental. En esta sección continuaremos estudiando un poco más la divisibilidad, pero desde un punto de vista diferente. Una congruencia no es otra cosa que una afirmación acerca de la divisibilidad. Sin embargo es más que una notación conveniente. El comportamiento de las congruencias es muy parecido a la igualdad.

Algoritmo 3 Algoritmo extendido de Euclides

Entrada: Números enteros a y b.
Salida: El máximo común divisor de a y b (mcd).

1: $r_0 \longleftarrow a$, $r_1 \longleftarrow b$, $t_0 \longleftarrow 0$, $s_0 \longleftarrow 1$, $t_1 \longleftarrow 1$, $s_1 \longleftarrow 0$.
2: $i \longleftarrow 1$.
3: **mientras** $r_i \neq 0$ **hacer**
4: Divida r_{i-1} entre r_i para obtener el cociente q_i y el residuo r_{i+1}.
5: $s_{i+1} \longleftarrow s_{i-1} - q_i s_i$.
6: $t_{i+1} \longleftarrow t_{i-1} - q_i t_i$.
7: $i \longleftarrow i + 1$.
8: **fin mientras**
9: **devolver** r_{i-1} es el máximo común divisor de de a y b y se expresa $r_{i-1} = as_{i-1} + bt_{i-1}$.

Por este hecho, la notación para las congruencias fue elegido intencionalmente para parecerse al símbolo de igualdad.
Una de las nociones básicas que ayuda mucho en la teoría de números son las congruencias o la aritmética modular. La siguiente definición se usa en general sólo para números primos.

Definición. *Si un número entero m diferente de 0, divide a la diferencia $a - b$, se dice que a es congruente con b módulo m y se escribe $a \equiv b \ (mod\ m)$. Si $a - b$ no es divisible entre m, se dice que a no es congruente con b módulo m y se escribe $a \not\equiv b \ (mod\ m)$.*

$m \mid a - b$. Es decir $a - b$ es múltiplo de m.

Otra manera de verlo es que si al dividir a entre p y b entre p, entonces: $a = q_1 p + r_1$ y $b = q_2 p + r_2$. Ahora si $r_1 = r_2$, entonces $a - b = (q_1 - q_2)p$.

Ejemplos.

Para $m = 6$

$52 \equiv 4 \ mod\ 6$
$\Rightarrow 6 \mid 52 - 4 = 48.$
$25 \equiv 1 \ mod\ 6$
$\Rightarrow 6 \mid 25 - 1 = 24.$

$7 \equiv 1 \ mod\ 6$
$\Rightarrow 6 \mid 7 - 1 = 6.$
$21 \equiv -3 \ mod\ 6$
$\Rightarrow 6 \mid 21 - (-3) = 24.$

Las congruencias tienen muchas aplicaciones en las ciencias de la computación, son usadas para criptografía simple y compleja, relojes y calendarios, generar números aleatorios y tablas de dispersión, etc. Por esta razón estudiaremos un poco las siguientes propiedades que poseen las congruencias.

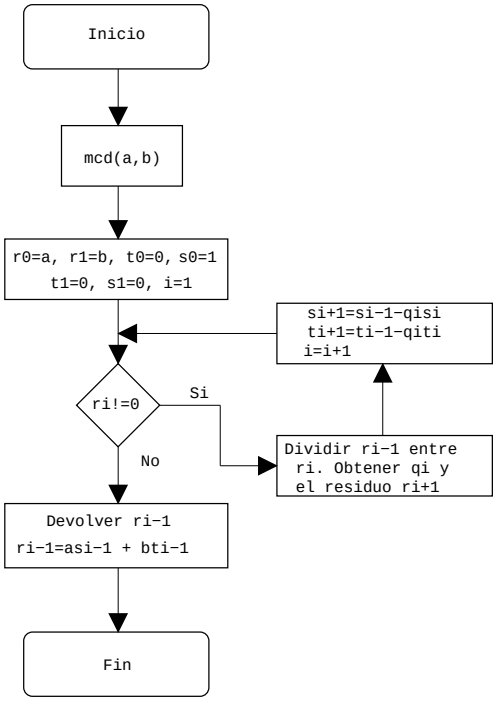

Figura 2.2: Diagrama correspondiente al algoritmo extendido de Euclides.

2.7.1. Propiedades

Sean a, b, c y n, enteros con $n \neq 0$

1. $a \equiv 0$ *(mod n) si y sólo si* $n \mid a$.
2. $a \equiv a$ *(mod n).*
3. $a \equiv 0$ *(mod n) si y sólo si* $b \equiv a$ *(mod n).*
4. *Si* $a \equiv b$ *y* $b \equiv c$ *(mod n), entonces* $a \equiv c$ *(mod n).*

La propiedad número 1, $a \equiv 0 \pmod{n}$, significa que $a = a - 0$ es un múltiplo de n, lo cual es lo mismo que $n \mid a$. La propiedad número 2, $a - a = 0 \cdot n$, así $a \equiv a \pmod{n}$. La propiedad número 3 se puede mostrar, si $a \equiv b \pmod{n}$, escribir $a - b = nk$. Entonces $b - a = n(-k)$, así $b \equiv a \pmod{n}$. Para probar la otra implicación, sólo basta con invertir los roles de a y de b y se seguir el mismo camino. Para la propiedad número 4, escribir $a = b + nk$ y $c = b + ln$. Entonces $a - c = n(k - l)$, así $a \equiv c \pmod{n}$.

2.8. Relaciones de equivalencia

Las relaciones se generalizan del concepto de función. Por ejemplo, la presencia del par ordenado (a, b) en una relación se puede interpretar como que existe una relación

de a a b. Se puede pensar en una relación R, de un conjunto A a un conjunto B, como el subconjunto al aplicar el producto cartesiano $A \times B$.

Ejmplos.
Tomemos un número primo p. Dos números enteros m y n están relacionados $(m, n) \in R$, o bien $m\ R\ n$, si tienen el mismo residuo al dividir por p.

$$\begin{array}{c|l} p & Q \\ \hline m & \\ r & \end{array} \qquad \begin{array}{c|l} p & q \\ \hline n & \\ r & \end{array}$$

Si escribimos a $m = Q \times p + r$ y $n = q \times p + r$, para algunos números enteros Q y q, así $m - n = Q \times p - q \times p = (Q - q)p$, osea $(m, n) \in R$, es decir m y n están relacionados.

Definición. *Una relación R en un conjunto A es de equivalencia*[3][4] *si*

1. Es relfexiva.
2. Es simétrica, es decir $(a, b) \in R \Rightarrow (b, a) \in R$.
3. Es transitiva, es decir $(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$.

Ejmplos.
Dados dos enteros m y n, si $m - n$ es divisible por p, donde p es un número primo, entonces R es de equivalencia.

Es muy sencillo darse cuenta que el ejemplo anterior es una relación de equivalencia. Tenemos que para toda m, $m - m = 0$, por lo tanto es relfexiva. Para la simetría, como $(m, n) \in R$, lo cual lo podemos escribir $m - n = kp$ para algún k en los números enteros, si multiplicamos por -1, entonces $n - m = (-k)p$, por lo tanto la simetría también se cumple. Finalmente, tenemos que $(m, n) \in R$ y $(n, l) \in R$, lo cual implica que, $m - n = kp$ y $n - l = zp$, sumamos ambas expresiones, $m - l = kp + zp$, por lo tanto $m - l = (k + z)p$, así $(m, l) \in R$. Así queda mostrado que la proposición anteriormente escrita es una relación de equivalencia.

Proposición. *Si R es una relación de equivalencia en un conjunto A. Para cada $a \in X$. Sea*

$$[a] = \{a \in X \mid xRa\}$$

.
(En palabras, $[a]$ es el conjunto de todos los elementos de X que están relacionados con a). Entonces

$$S = \{[a] \mid a \in X\}$$

es una partición[2] de X.

[2]Una partición de un conjunto X divide a X en subconjuntos que no se traslapan.

Si el lector desea obsevar una demostración formal para esta proposición, revisar con más detalle[4][3].

Los conjuntos $[a]$ definidos anteriormente se llaman **clases de equivalencia**[4] de X dada por la relación R.

Ahora supongamos que tenemos aun número primo p, fijo. Entonces m está relacionado con n módulo p si al dividirlo por p ambas tienen el mismo número, es decir; tenemos a $m = p \times Q_1 + r$ y $n = p \times Q_2 + r$, así $m - n = p(Q_1 - Q_2)$, por lo tanto m es equivalente a n módulo p si su diferencia es un múltiplo de p.

Ejemplos.

Para el número primo $p = 7$, m está relacionado con el 0 si $(m - 0)$ es un múltiplo de 7, es decir:

$$[0] = \{0, 7, 14, 21, ...\} \qquad [4] = \{4, 11, 18, 25, ...\}$$
$$[1] = \{1, 8, 15, 22, ...\} \qquad [5] = \{5, 12, 19, 26, ...\}$$
$$[2] = \{2, 9, 16, 23, ...\} \qquad [6] = \{6, 13, 20, 27, ...\}$$
$$[3] = \{3, 10, 17, 24, ...\} \qquad [7] = \{7, 14, 21, 28, ...\} = [0]$$

Las clases de equivalencia correspodientes a la relación anteriror son:
$[0], [1], [2], [3], [4], [5], [6], [7]$. En general, trabajaremos con enteros *mod* p, denotados $\mathbb{Z}_p$. Para ilustrar un poco más acerca de esto, utilicemos el ejemplo anterior; a los enteros $\mathbb{Z}_7 = \{0, 1, 2, 3, 4, 5, 6\}$, donde $\mathbb{Z}_7$ es el conjunto de clases de equivalencia para el número primo $p = 7$.

Esto puede ser considerado como el conjunto $\{0, 1, 2, ..., n - 1\}$, con suma, resta y multiplicación *mod* n. Una vez que fijamos un entero $n > 1$, las propiedades, que citamos, nos permiten hablar acerca de la aritmética *mod* n en el conjunto $\mathbb{Z}_n$ de enteros entre 0 y $n - 1$. Se define:

Sean a, b, c d y n, enteros con $n \neq 0$ y supongamos que $a \equiv b$ (mod n) y $c \equiv d$ (mod n).

$$a + c \equiv b + d,\ a - c \equiv b - d,\ a \cdot c \equiv b \cdot d\ (mod\ n).$$

La propiedad anterior dice que podemos desarrollar operaciones aitméticas de suma, resta y multiplicación de congruencias. Si escribimos $a = b + nk$ y $c = d + nl$, para los números enteros k y l. Entonces $a + c = b + d + n(k + l)$, así $a + c \equiv b + d$ (mod n). Para probar que $a - c \equiv b - d$ es similar. Para la multiplicación, tenemos que $ac = bd + n(dk + bl + nkl)$, así $ac \equiv bd$.

Ejemplos.
La aritmética *mod* 7 tiene propiedades muy interesante ya que es un campo[3][3]. Consideremos a las tablas + y × para la aritmética *mod* 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 1: operador +

×	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: operador ×

Aquí está un ejemplo de cómo podemos hacer álgebra *mod* n. Consideremos el siguiente problema: Resolver $x + 7 \equiv 3 \ (mod\ n)$.

Solución: $x \equiv 3 - 7 \equiv -4 \equiv 13 \ (mod\ n)$.

Hagamos un ejemplo más con respecto a la aritmética *mod* n, consideremos a la Tabla 1. Se puede construir un autómata M que haga la aritmética del operador +.

Sea el alfabeto $\Sigma = \{\hookleftarrow a, b, c\}$, donde $\hookleftarrow$ significa $\langle RESET \rangle$, $a = 0$, $b = 1$ y $c = 2$. El lenguaje regular[4][5] asociado para el autómata es $L(M) = \{w \mid w\ es\ sum\ mod\ 7\}$, donde sum es la suma de los símbolos que pertenecen al alfabeto.

2.9. Exponenciación Modular

Supongamos que queremos calcular 2^{1234} $(mod\ 789)$. Si primero calculamos 2^{1234}, entonces reducimos *mod* 789, estaremos trabajando con números muy largos, apesar de que la respuesta tiene solo tres dígitos. Por lo tanto deberíamos desarrollar la operación y después calcular el residuo. Calculando las potencias consecutivas de 2 requerirá que se dearrolle la multiplicación modular 1233 veces. Como se puede analizar, este método es muy lento para la práctica, especialmente cuando son números muy grandes. Una manera más eficiente de resolver esto, es la siguiente (todas serán congruencias *mod* 789).

[3]Es una estructura albebraica con propiedades de suma, multiplicación, resta y división que cumple con los axiomas de campo. Ejemplo: $\mathbb{R}$, $\mathbb{Q}$ y $\mathbb{Z}_p$, donde p es un número primo.

[4]Un lenguaje es regular si existe un autómata que lo reconoce.

Empezamos con $2^2 \equiv 4 \ (mod\ 789)$ y repetidamente elevamos al cuadrado ambos lados para obtener las siguientes congruencias:

$$2^4 \equiv 4^2 \equiv 16$$

$$2^8 \equiv 16^2 \equiv 256$$

$$2^{16} \equiv 256^2 \equiv 49$$

$$2^{32} \equiv 34$$

$$2^{64} \equiv 367$$

$$2^{128} \equiv 559$$

$$2^{256} \equiv 37$$

$$2^{512} \equiv 580$$

$$2^{1024} \equiv 286$$

Ya que $1234 = 1024+128+34+16+2$ (esto significa que 1234 es igual a 10011010010 en binario), tenemos que

$$2^{1024} \equiv 286 \cdot 559 \cdot 367 \cdot 49 \cdot 4 \equiv 481 \ (mod\ 789).$$

Notar que nunca necesitamos trabajar con números más grandes que 788^2. Si queremos calcular $a^b \ (mod\ n)$, el mismo método trabaja en general, podemos hacer esto con a lo más $2\log_2(b)$ multiplicaciones $mod\ n$ y nunca tendremos que trabajar con números más grandes que n^2. Lo cual significa que esta operación la podemos hacer rapidamente sin necedidad de mucha memoria
Este método es muy útil si a, b y n son números de 100 dígitos. Si nosotros hacemos el cálculo, en una computadora, de a^b, entonces reducimos el $mod\ n$. Lo cual seguramente, provocará un desbordamiento en memoria de la computadora.

2.10. El pequeño Teorema de Fermat

Dos de los resultados más básicos en la teoría de números son el Teorema de Fermat. Admirados originalmente por sus valores teoricos, recientemente se ha provado que tiene importancia en aplicaciones criptográficas.

El pequeño Teorema de Fermat. *Si p es un primo y $p \nmid a$, entonces*

$$a^{p-1} \equiv 1 \ (mod\ p).$$

Ejemplos. $2^{10} = 1024 \equiv 1 \ (mod\ 11)$. De esto podemos evaluar $2^{53} \ (mod\ 11)$: Escribimos $2^{53} = (2^{10})^5 2^3 \equiv 1^5 2^3 \equiv 8 \ (mod\ 11)$. Notar que cuando trabajamos mod 11, esencialmente estamos trabajando con exponentes mod 10, no mod 11, de $53 \equiv 3 \ (mod\ 10)$, deducimos que $2^{53} \equiv 2^3 \ (mod\ 11)$.

Capítulo 3

Algoritmo de Montgomery

3.1. Introducción

La motivación para el estudio de algoritmos eficientes y de alta velocidad para la multiplicación modular viene de las aplicaciones en la criptografía: las llaves públicas [6]. Ciertamente una de los avances más útiles e interesantes ha sido la introducción al llamado algoritmo de multiplicación modular de Montgomery. El algoritmo de Montgomery es usado para acelerar la multiplicación modular y la exponenciación modular. El algoritmo de Montgomery calcula

$$MonPro(a, b) = a \cdot b \cdot \ r^{-1} \ mod \ n \tag{3.1}$$

dado a, b n y r tales que $mcd(n, r) = 1$. Por lo tanto el algoritmo trabaja con algún r, el cual es, primo relativo con n, esto es más útil cuando r es una potencia de 2. En este caso el algoritmo de Montgomery desarrolla divisiones por una potencia de 2, lo cual es una operación instrinsicamente más rápida para computadoras de propósito general; esto conduce a una implementación más sencilla que la multiplicación modular ordinaria, la cual es típicamente más rápida.

El algoritmo de Montgomery es considerado el más rápido de los algortimos para calcular, xy mod n, en computadoras cuando los valores de x, y y n son muy grandes[9]. En esta sección se describe el algoritmo de Montgomery para la multiplicación modular.

3.2. La multiplicación de Montgomery

Supongamos que se quiere calcular xy mod n en una computadora. Primeramente elegimos un número entero postivo r más grande que n y primo realtivo a n. El valor de r es usualmente 2^k para algún entero positivo k. Esto es porque la multiplicación, la división y el módulo por r se pueden hacer facilmente con operaciones lógicas y

corrimientos en una computadora.

Sea n el modulo, un entero de $k - bits$, es decir; $2^{k-1} \leq n < 2^k$, y sea r 2^k. El algoritmo de multiplicación de Montgomery requiere que r y n sean primos relativos, es decir; $mcd(r, n) = mcd(2^k, n) = 1$. Así este requisito se satiface si n es impar. A fin para describir el algoritmo de multiplicación de Montgomery, primero definimos el residuo-n de un entero $a < n$ como $\bar{a} = a \cdot r \ (mod\ n)$. Con esto es fácil mostrar que el conjunto

$$\{a \cdot r \ mod\ n \mid \ 0 \leq a \leq n - 1\}$$

es un sistema de residuos completo [1] [4], es decir; contiene a todos los números entre 0 y $n - 1$. Así, hay una correspondencia uno a uno entre los número en el rango 0 y $n - 1$ y los números del conjunto anterior. El algoritmo de reducción de Montgomery aprovecha esta propiedad utilizando una una rutina de multiplicación más rápida la cual calcula el residuo-n de el producto de dos enteros, los cuales sus residuos-n son dados. Dados dos residuos-n $\bar{a}$ y $\bar{b}$ el producto de Montgomery está definido como el residuo-n

$$\bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1} \ (mod\ n,) \tag{3.2}$$

donde r^{-1} es el inverso de r mod n, es decir; este es el número con la propiedad $r \cdot r^{-1} = 1 \ (mod\ n)$. El número c, resultado de (3.2) es en realidad el residuo-n del producto $c = a \cdot b \ (mod\ n)$, donde

$$\begin{aligned} \bar{c} \qquad &= \bar{a} \cdot \bar{b} \cdot r^{-1} \ (mod\ n) \\ &= a \cdot r \cdot b \cdot r \cdot r^{-1} \ (mod\ n) \\ &= c \cdot r (mod\ n) \end{aligned}$$

A fin para describir el algoritmo de reducción Montgomery, adicionalmente, necesitamos calcular, n', el cual es un entero con la propiedad $r \cdot r^{-1} - n \cdot n' = 1$. Los enteros r^{-1} y n', ambos pueden ser calculados con el algoritmo extendido de euclides. Ya que el $mcd(r, n) = 1$, entonces existen dos números r^{-1} y n' con $0 < r^{-1} < n$ y $0 < n' < r$. El cálculo de de Monpro$(\bar{a}, \bar{b})$ se logra a través de:
Pseudocódigo para el algoritmo de reducción.

Algoritmo 4 -$MonPro(\bar{a}, \bar{b})$-

Entrada: Números enteros $\bar{a}$ y $\bar{b}$.
Salida: El cálculo de $\bar{a} \cdot \bar{b} \cdot r^{-1}$ mod n.

```
1: t := ā · b̄;
2: u := (t + (t · n′ mod r) · n)/r;
3: si u > n entonces
4:     u := u − n;
5: fin si
6: devolver u;
```

El algoritmo está basado en el hecho de que el cálculo para $\bar{a} \cdot \bar{b} \cdot r^{-1} \bmod n$ puede ser hecho muy eficientemente por algoritmo ***reducción***[9] [10] [8] [6].

Esta rutina requiere de la aritmética módulo r, la cual es muy sencillo calcular en una computadora si $r = 2^k$. Así el producto de Montgomery es un algoritmo potencialmente más rápido que el cálculo ordinario de $a \cdot b \ mod \ n$ el cual envuelve divisiones por n. Sin embargo, la conversión de un residuo ordinario a un residuo-n, el cálculo de n' y volver a convertir al residuo ordinario consumen gran cantidad de tiempo y no es buena idea usar esto si queremos usar el algoritmo de Montgomery para calcular la multiplicación modular de números muy pequeños. Esto alcanza un buen desempeño para enteros largos.

3.3. Teorema 1

Si $c = a \cdot b \bmod n$, entonces $c = MonPro(a, b)$;

Prueba:

$$\begin{aligned} c &= c \cdot r \ mod \ n \\ &= a \cdot b \cdot r \ mod \ n \\ &= a \cdot r \cdot b \cdot r \cdot r^{-1} \ mod \ n \\ &= a \cdot b \cdot r^{-1} \ mod \ n \\ &= MonPro(a, b) \end{aligned}$$

3.4. Teorema 2

$c = MonPro(c, 1)$;

Prueba:

$$\begin{aligned} c &= c \cdot r \cdot r^{-1} \ mod \ n \\ &= c \cdot r^{-1} \ mod \ n \\ &= c \cdot 1 \cdot r^{-1} \ mod \ n \\ &= MonPro(c, 1) \end{aligned}$$

Una vez mostrado que el algoritmo de Montgomery funciona, utilizaremos el procedimiento $MonPro$ para calcular $c := a \cdot b \ mod \ n$ como sigue:

Pseudocódigo para el algoritmo de Montgomery. La siguiente figura es un

Algoritmo 5 -$montgomery(a, b, n)$-

Entrada: Números enteros a, b y n.
Salida: Multiplicación Modular $c = a \cdot b \ mod \ n$.

1: *Calcular n' usando el Algoritmo Extendido de Euclides*;
2: $a := a \cdot r \ mod \ n$;
3: $b := b \cdot r \ mod \ n$;
4: $c := MonPro(a, b)$;
5: $c := Monpro(c, 1)$;
6: **devolver** c;

diagrama de estados para el algoritmo anterior, este diagrama de estados será implementados por varior módulos que se detallan en la sección Descripción en Hardware del algoritmo de Montgomery.

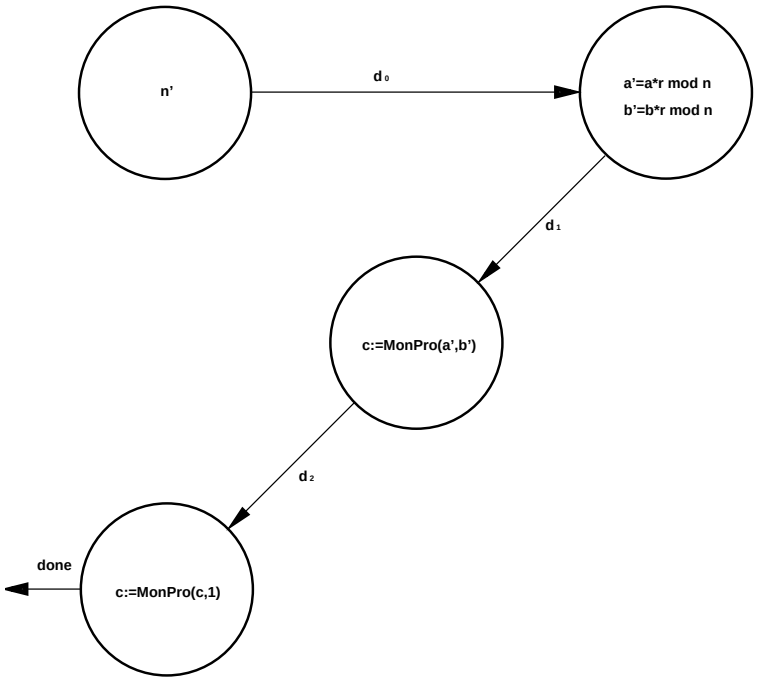

Figura 3.1: Máquina de estados para la multiplicación modular usando el algoritmo de Montgomery

Capítulo 4

Descripción en Hardware del algoritmo de Montgomery

4.1. Introducción

El algoritmo de Montgomery visto en el capítulo dos es muy simple, sin embargo la descripción en hardware es muy complicada. La descripción en hardware del algoritmo de Montgomery está compuesta por varios módulos: Cálculo de r, Divisor por restauración, Algoritmo Extendido de Euclides para calcular n', Dominio de Montgomery y Algoritmo de Montgomery, en caso de que los operandos sean de precisión mayor a 32 bits, se hace uso del módulo Formateo para los resultados y del módulo Memoria para manejar los operandos. Cada uno de esos módulos son detallados en el presente capítulo, así como el control, la coordinación y sincronización de todas las señales que integran a cada uno de los módulos.

4.2. Descripción general

El coprocesador cálcula la operación $X \cdot Y \ mod \ M$, donde X, Y son enteros muy largos, de n bits, a través del algoritmo de Multiplicación de Montgomery (MM). El resultado será $MM(X,Y) = XYr^{-1} \ mod \ M$, donde $r = 2^n$ y M es un entero en el rago de $2^{n-1} < M < 2^n$ tal que $mcd(r, M) = 1$.

Se recibe como entrada dos valores enteros X, Y y se analiza cada uno ellos para corroborar si los números se encuentran dentro del rago de precisión dependiendo de la versión del coprocesador, es decir; ya sea de 128 bits, 512 bits o 1024 bits, con el propósito de saber si es posible realizar la operacion aritmética. En el caso de que el rango de precisión sea correcto, los operandos son normalizados para poder manipularlos, seguir a realizar las operaciones y obtener los resultados. También se recibe un valor entero M. Para las aplicaciones criptográficas; M es usualmente un primo o un producto de primos. Antes de enviar el resultado este puede ser o no formateado, y utilizar este mismo resultado para realizar otras operaciones. La Figura 4.1 muestra el diagrama global del módulo coprocesador, donde se puede observar: datos de entrada, la precisión de los operandos y el resultado.

El cálculo de r es muy sencillo basta con obtener el $r = log_2 \ n + 1$, es por eso

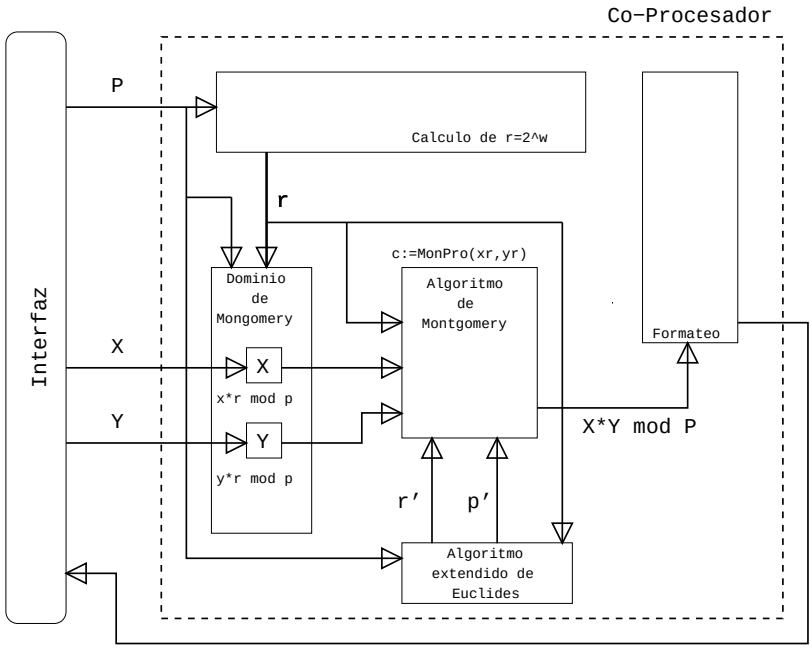

Figura 4.1: Circuito coprocesador matemático

que no se hace mucha importancia y una descripción muy detallada de este módulo que integra al coprocesador.

4.3. División por restauranción

El algoritmo de división por restauración es un algoritmo muy eficiente para calcular el cociente y el residuo de divisiones enteras [7], así como la multiplicación se puede realizar por medio de una serie de sumas y corrimientos, la división se puede realizar por medio de substracciones y corrimientos. Utilizando estos principios se diseñó el divisor de números enteros sin signo con la ayuda de:

- Registros: almacenamiento y corrimiento
- Unidades aritméticas: substractor y comparador
- Unidad de control

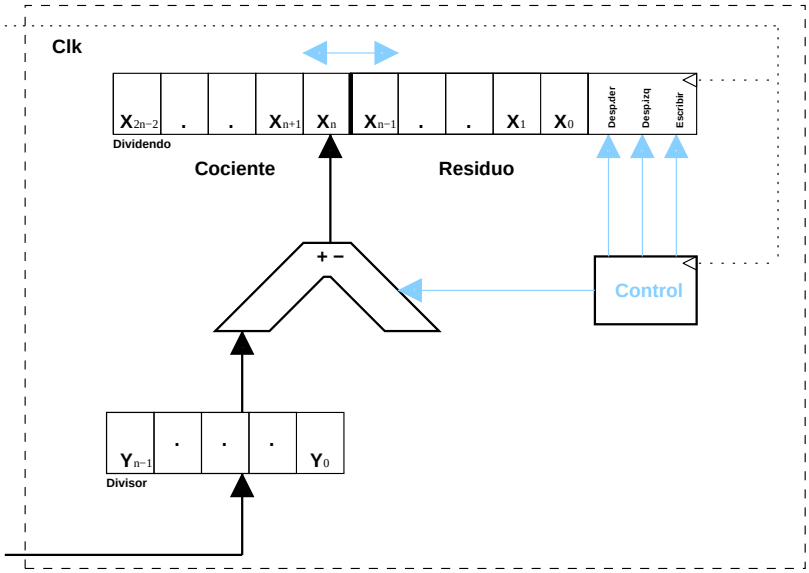

Figura 4.2: Circuito del módulo divisor de n bits

La Figura 4.2 muestra el circuito que corresponde a la arquitectura del módulo divisor serial (división por restauración) para número de n bits.
El diagrama de estados del divisor puede observarse en la Figura 4.3 y más adelante, en el apartado de Módulo coprocesador se podrá observar el diseño hardware que se implementó para el divisor. El divisor es muy imporante para los módulos de Dominio de Montgomery y Algoritmo Extendido de Euclides.

En S_0: si $St = 1$, se cargan los operandos, $Ld = 1$ e inician las operaciones, de lo contrario el circuito está inactivo. En S_1: si $C = 1$, indica que el cociente requerirá más bits, entonces $V = 1$ y se suspenden las operaciones de lo contrario se realiza un corrimiento a la izquierda $Sh = 1$. En S_2, S_3 y S_4: si $C = 1$, se realiza la substracción $Su = 1$. Se realiza el corrimiento a la izquierda $Sh = 1$ pasando al siguiente estado. En S_5: si $C = 1$, se realiza la substracción $Su = 1$ y se pasa al estado final.

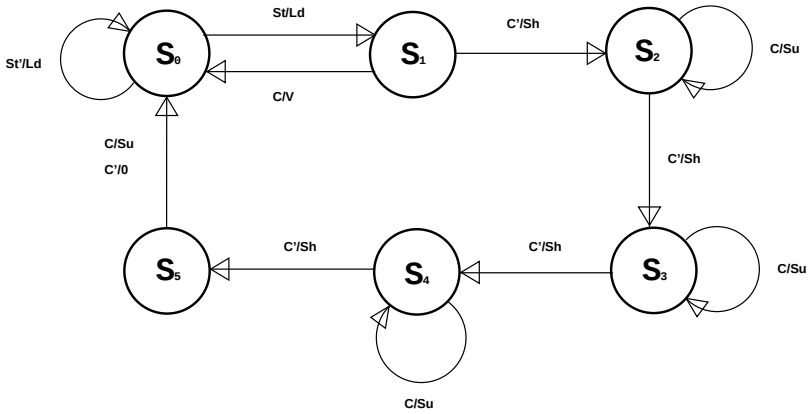

Figura 4.3: Máquina de estados para la división por restauración

4.4. Multiplicación serial

El algoritmo utilizado en esta parte, sigue los principios básicos del algoritmo para multiplicar números en base 10 [7], como se dijo anteriormente, la multiplicación se puede llevar acabo; de manera sencilla, utilizando sumas y corrimientos. Es necesario aclarar que el módulo multiplicador es es un módulo que sirve para multiplicar números sin signo ya que el propósito de este trabajo es multiplicar números muy grandes, pero positivos es por eso que se hace caso omiso a los número signados. Tal vez el lector se esté preguntando el por qué de la construcción de este módulo ya que en los lenguajes de descripción de hardware ya viene una operación de multiplicación. La respuesta a esto es porque en los lenguajes de descripción de hardware la operación viene implementada sólo para operandos de 32 bits y más de eso ya ya no se podrá utilizar. Por esta razón es muy importante prescindir de un multiplicador de n por m bits para cuando queramos multiplicar enteros muy grandes; por ejemplo: 1024 bits por 1024 bits. El módulo multiplicador está formador por:

- Registros: almacenamiento y corrimiento
- Unidad aritmética: sumador
- Unidad de control

A continuación, en la Figura 4.4 se muestra el diagrama de estados que corresponde al módulo multiplicador.

Básicamente en esta máquina de estados finitos son dos estados (S_1 y S_2) los más importantes, porque con éstos dos estados se calcula la multiplicación. En S_0 se verifica si está activado el circuito St, si lo lo está, entonces se cargan los operandos Ld y se procede a calcular la operación con los operandos cargados, de lo contrario no se calcula nada. En S_1 se verifica una bandera M, si está activada pasa al estado S_2, además de calcular una suma Ad; de lo contrario pasa al estado S_3 con un corrimiento a la derecha Sh. En el estado S_2 simplemente se hace un corrimiento más a la

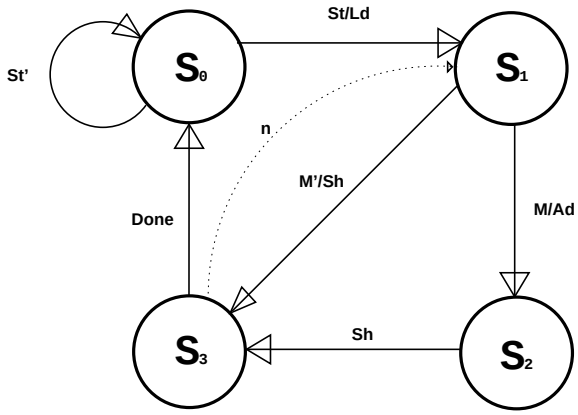

Figura 4.4: Máquina de estados para la multiplicación

derecha y pasa al estado S_3. En el estado S_3 verifica si se han cumplido el número de operaciones, dependiendo de la precisión de los registros para efectuar la multiplicación, si se cumplen pasa al estado S_0 terminando así de calcular la multiplicación encendiendo una bandera *Done*, de lo contrario regresa al estado S_1 y continua hasta terminar el cálculo.
La Figura 4.5 muestra la arquitectura para el módulo multiplicador serial de n por n bits. A diferencia del divisor, ya que parecen los mismos circuitos, es que el multiplicador escribe, suma y desplaza hacia la derecha y el divisor, además de todas las funciones anteriores, realiza un desplazamiento hacia la izquierda, resta y compara.

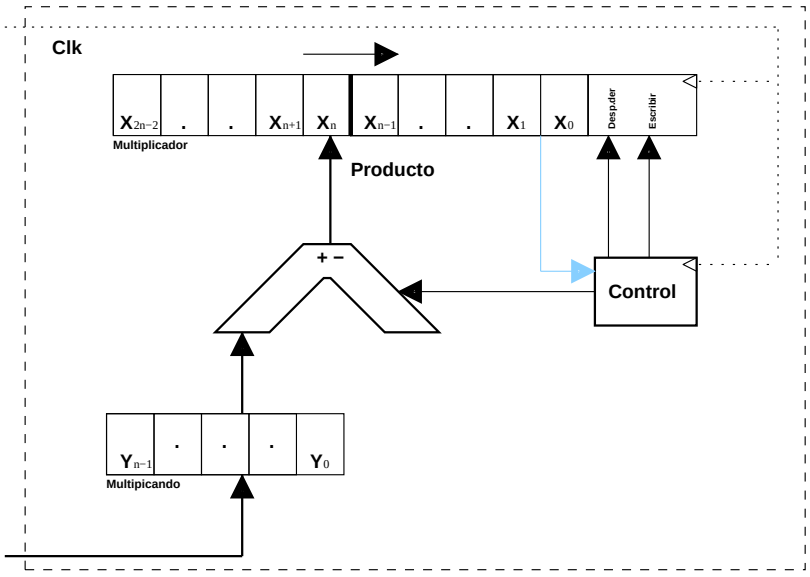

Figura 4.5: Circuito del módulo multiplicador de n por m bits

4.5. Algoritmo Extendido de Euclides

Se debe de recordar que en la sección de conceptos básicos de la teoría de números se habló acerca del algoritmo extendido de Euclides y su gran utilidad, con el cual podemos calcular los inversos modulares. También en el capítulo dos se mostró en la Figura 2.2 un diagrama de flujo que corresponde al pseudocódigo del algoritmo extendido de Euclides. Es importante mencionar que el algoritmo extendido de euclides, para su implementación en hardware, es necesario contar con un divisor; en este caso el módulo Divisor por restauración el cual vimos anteriormente. Es necesario de este divisor porque el lector puede notar en el pseudocódigo que, se necesita calcular cocientes y residuos durante un ciclo, siendo más preciso mientras la condición de un ciclo se cumpla.

El diagrama de estados del algoritmo extendido de Euclides puede observarse en la Figura 4.6 y más adelante, en el apartado de Módulo coprocesador se podrá observar el diseño hardware que se implementó para el algoritmo extendido de Euclides.

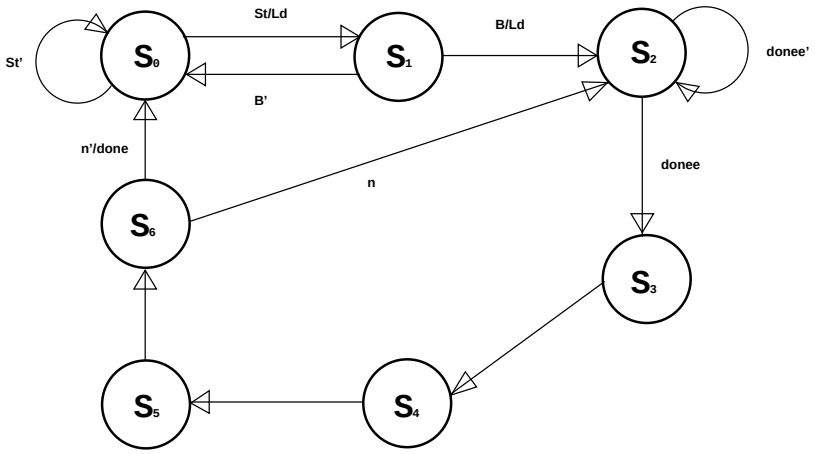

Figura 4.6: Máquina de estados para el cálculo de n' por medio del algoritmo extendido de Euclides

En S_0: si $St = 1$, se cargan los operandos, $Ld = 1$ e inician las operaciones, de lo contrario el circuito está inactivo. En S_1: si $B = 1$, indica que el cociente requerirá más bits, entonces se suspenden las operaciones, de lo contrario se hacen algunos cálculos y se pasa a el siguiente estado. En S_2: se habilita el divisor y se obtiene el cociente y residuo necesarios para pasar a los siguientes cálculos, una vez obtenidos se pasa al siguiente estado. En S_3: se realizan un par de multiplicaciones. En S_4 y S_5: se calculan algunas sumas y asignaciones. En S_6: si $n > 0$, se continua con la ejecución de la máquina de estados pasando al estado 2, de lo contrario $done = 1$ y pasa al estado final.

4.6. Dominio de Montgomery

En el capítulo tres se habló del algoritmo de Montgomery y para que el algoritmo funcione se necesita pasar a los operandos a y b a su respectivo dominio de Montgomery, esto no es otra cosa más que el residuo-n de cada operando. Para la implementación de este módulo en hardware también se hace dependencia del módulo Divisor por restauración y del módulo Cálculo de r. En este módulo se utilizan de corrimientos a izquierda porque se necesita multiplicar los operandos por r y éste es una potencia de dos. El diagrama de estados del módulo Dominio de Montgomery puede observarse en la Figura 4.7 y más adelante, en el apartado de Módulo coprocesador se podrá observar el diseño hardware que se implementó para el presente módulo.

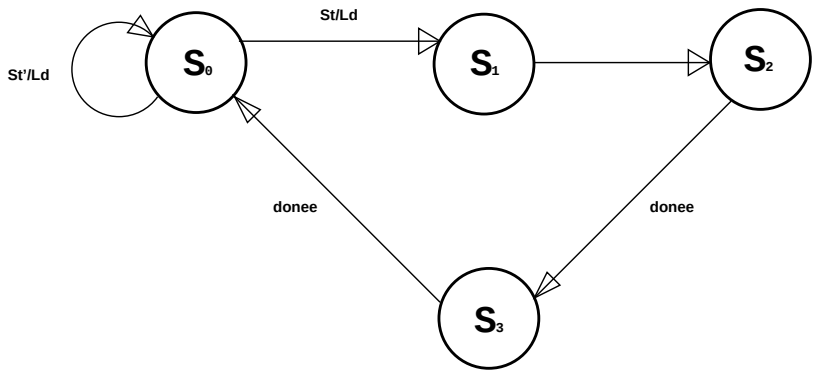

Figura 4.7: Máquina de estados para el cálculo de de los residuos-n de a y b

En S_0: si $St = 1$, se cargan los operandos, $Ld = 1$ e inician las operaciones, calculando a r, de lo contrario el circuito está inactivo. En S_1: se calculan las multiplicaciones por r, esto con corrimientos a la izquierda. En S_2, S_4 : se activa el divisor y se obtiene el residuo, $donee = 1$ y pasa al estado final.

4.7. Reducción de Montgomery

En la sección del algoritmo de Montgomery se detalló este módulo por medio de un pseudocódigo para calcular $c = MonPro(a', b')$. Para que este módulo funcione necesita, primero; haber calculado a r, segundo; haber calculado el inverso modular n' a través del algoritmo extendido de euclides, tencero; tener los residuos-n de los operandos a y b. Recordar que la sección del algoritmo se probó un par de teoremas. Esto para demostrar que para calcular $a \cdot b \cdot \ mod\ n$ se necesta: $c = MonPro(a', b')$ y $c = MonPro(c, 1)$. Este módulo se encarga de las dos operaciones.

El diagrama de estados del módulo Reducción de Montgomery puede observarse en la Figura 4.8 y más adelante, en el apartado de Módulo coprocesador se podrá observar el diseño hardware que se implementó para el presente módulo.

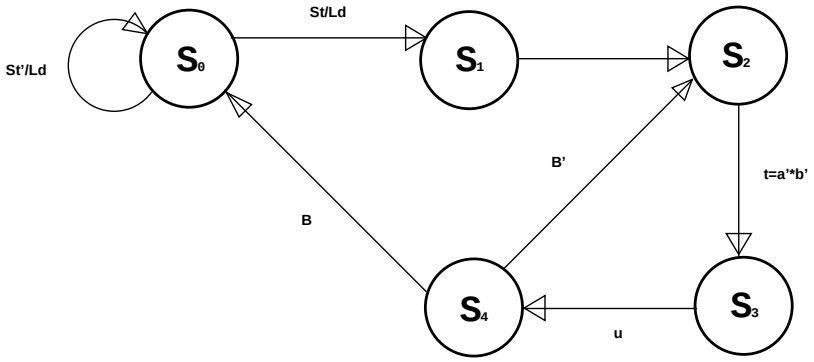

Figura 4.8: Máquina de estados para el cálculo de $a \cdot b \cdot r^{-1}$ *mod* n con ayuda del algoritmo de reducción de Montgomery

En S_0: si $St = 1$, se cargan los operandos, $Ld = 1$ e inician las operaciones, En S_1 S_2 S_3: se calcula t y u, es decir $c = MonPro(a', b')$. En S_4: si $B = 0$ se regresa al estado S_2 y se calcula $c = MonPro(c, 1)$, de lo contrario se pasa el estado final, $B = 1$.

4.8. Módulo coprocesador

Este módulo incluye a todos los demás módulos antes mencionados. La Figura 4.9 muestra la arquitectura que se diseñó para el coprocesador matemático.

Recibe como entrada tres valores: X, Y y P, éstos son números enteros de n de bits y como salida una señal de culminación de la operación y un vector con el resultado: $X * Y$ *mod* n. Si n es mayor a 32 bits se utiliza una versión diferente del coprocesador para que puede trabajar con los números mayores de 32 bits.

Como se dijo en la sección del algoritmo de Montgomery, para que el coprocesador funcione de manera correcta P debe ser un número primo, X y Y deben de ser números enteros positivos menores que P. Además recibe una señal rdy de parte del divisor, esta señal le indica que se ha terminado de hacer la división y que el módulo puede prescindir de los resultador del divisor, también tiene una señal e la cual habilita al divisor para cargar sus operandos y comenzar haciendo los cálculos. El coprocesador cuenta con un circuito combinacional para calcular $r = 2^w$, el cual recibe un vector P como entrada, de tal manera que $mcd(r, p) = 1$. Una vez calculado a r se activa el módulo Dominio de Montgomery para calcular los residuos-n de los operandos de entrada X y Y. Este módulo recibe como entrada X, Y números enteros, un número primo P, el valor de r. Como salida se obtienen $x * r$ *mod* n, $y * r$ *mod* n y una señal d_1 de habilitación del Módulo Algoritmo Extendido de Euclides para calcular el inverso modular. El módulo Dominio de Montgomery depende del módulo División por restauración y un registro de n bits para poder hacer corrimientos a la izquierda.

El módulo Algoritmo Extendido de Euclides recibe como entrada a P, r y una señal st, esta señal es de habilitación del módulo. Esta señal es alimentada por la señal de culminación del módulo Dominio de Montgomery, indicándole que puede activarse y comenzar a calcular las operaciones. Además también cuenta con las señales rdy y e para el control del divisor. Como salida se obtiene p' el cual es el inverso modular de P y una señal de culminación del módulo.

El módulo Algoritmo de Montgomery recibe como entrada, los valores calculados anteriormente por los otros módulos, es decir; recibe al vector P, $r = 2^w$, $x * r \ mod \ n$, $y * r \ mod \ n$, el inverso modular p' una señal de habilitación st, la cual se activa, con la señal de culmincación d_2, del módulo que calcula el inverso modular p'. Además el módulo está comunicado con un registro que hace corrimientos a la derecha, este registro es para porder hacer divisiones de manera más rápida, las cuales son necesarias en el algoritmo de Montgomery. Como salida se obtiene el vector $x * y \ mod \ n$, este vector contiene la operación hecha por el coprocesador utilizando la multiplicación modular del algoritmo de Montgomery y una señal *Done* de culminación de la operación por parte del coprocesador de aritmética entera.

El divisor está formado por un registro de X_{2n-1} bits; en el cual se carga el dividendo cuando quiere hacer una división, un registro de Y_{n-1} bits; aquí se carga el divisor. El divisor cuenta con una unidad aritmética lógica; esta unidad se encarga de hacer operación de substracción y comparación, dependiendo del resultado de la unidad aritmética, el control toma la decisión de hacer un corrimiento a la izquierda, colocando un 1 si la substracción se pudo hacer y un 0 en caso contrario. El control tiene una señal como entrada st, la cual está comunicada con los módulos de Dominio de Montgomery y Algoritmo Extendido de Euclides, ésto habilitarán el divisor cuando presciendan del cociente y el residuo de la división. El control decide si hacer un corrimiento a la izquierda modificando el bit menos significativo del registro X_0 o simplemente hacer el corrimiento sin ninguna modificación, además el control también es quien se encarga de cargar los operandos, dividendo y divisor, al módulo Divisor. Ya cuando se ha terminado de hacer la división el control manda la señal d_0 de culminación y el módulo que hizo la petición de dividir dos núemeros, pordrá obtener el cociente y el residuo del registro, siendo $X_{2n-2} - X_n$ el resultado del cociente y $X_{n-1} - X_0$ para el residuo.

Es importante hablar de los registro para hacer corrimientos $shift\_right$; este registro está comunicado con el módulo Dominio de Montgomery, se encarga de hacer las divisiones a través de corrimientos a la derecha, de manera más rápida y eficiente. El módulo Algoritmo Extendido de Euclides tiene comunicación con otro registro para hacer corrmientos $shift\_right$; se encarga de hacer las multiplicación, a través de corrimientos a la izquierda. Para ambos registros se indica el número de corrimientos que se harán, una vez terminado de hacer el corrimiento, entonces el registro vuelve a cargar el vector resultado a su respectivo módulo que hizo la petición.

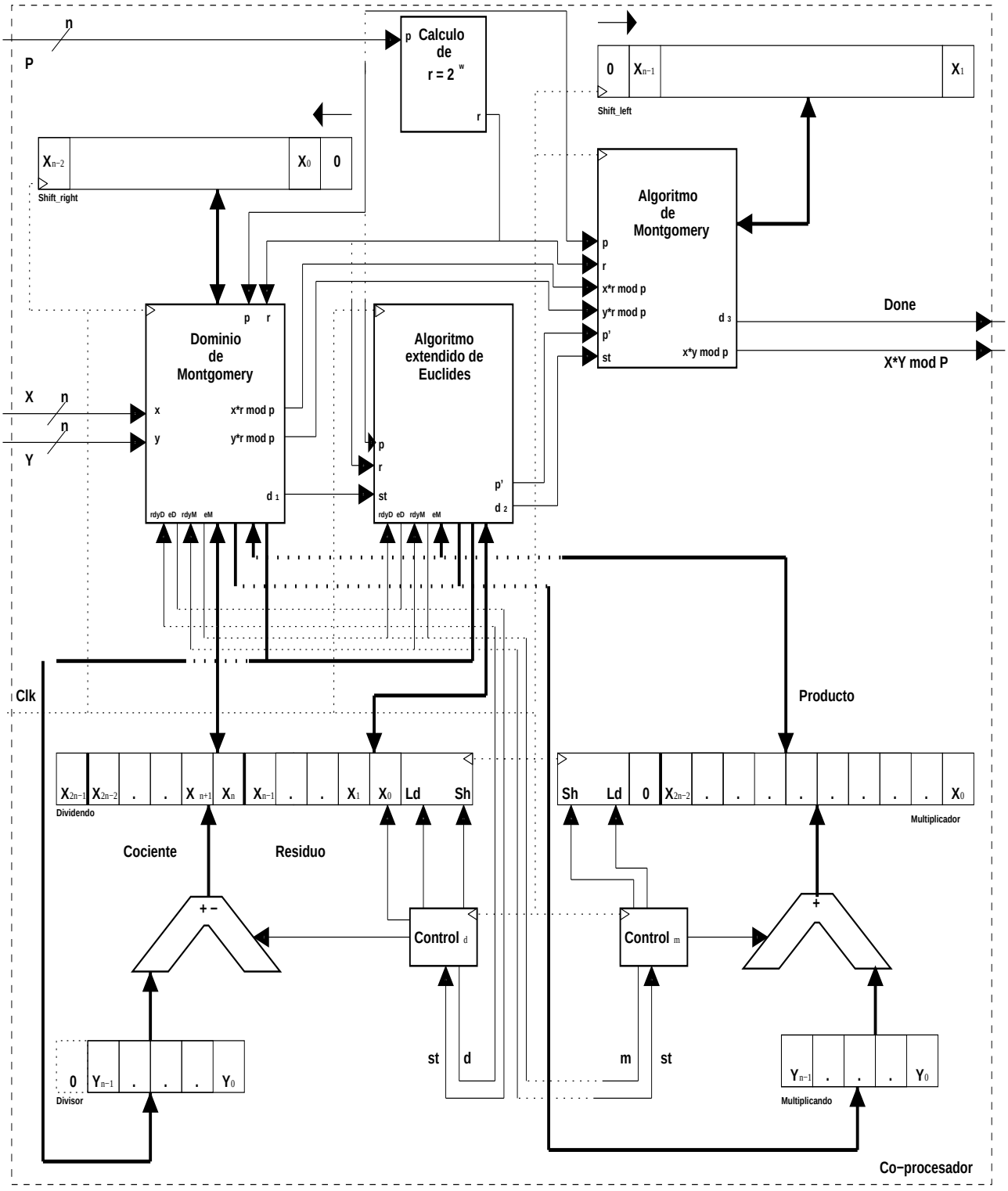

Figura 4.9: Ruta de datos detallada del coprocesador

Capítulo 5

Resultados y conclusiones

5.1. Resultados

Se hicieron varias pruebas con el coprocesador para verificar el correcto funcionamiento. El la sección de Apéndice, se encuentra el apartado de códigos fuente, en esta parte se puede consultar el código de la implementación del algoritmo de Montgomery para la multiplicación modular en Maple, para la versión en software y en VHDL para la versión en hardware.
Para la primer prueba del coprocesador en VHDL se escaló el coprocesador para el funcionamiento con números a, b y p (primo) de 512 bits utilzando una representación en base 16, es decir; números hexadecimales y para la versión software hecha en Maple se utilizaron los mismos números, pero en base decimal; a continuación se muestran los valores:

a_{16} = 123456789987654321123456789876543211234567890543210AB1025BF05012340AB1020BF05C123400000000C12345AB1025BF05

b_{16} = 987654321123654789987654321123456789987654321098765432110BBBB00CC12345678B4512AA0B4512AAA000000CC12345678B4512AA

p_{16} = 5D54A5F6EE7D8AF711FDE83AE16B4DD0EE26A39FDA62B2EFAB5C1ED34F912E6BB8E54C432FCDA6D2E2A3D605A0C662155B4919A4FCA3EAA04D32CA0000000001

$(a \cdot b \ mod \ p)_{16}$ = 286A02772562A88A375A913C785D3D1407C9A8240E505B93C343419C24523 1C0E8F231477E50CF1C95ACC9E88488C252F977AECFDE218CB3781D194995A36FDA

a_{10} = 95344411944784309226009744381993397224943700242426179960352206315901297071060046317439199710167235674935753414714773331570050171414676737051944590622469

b_{10} = 79850945005138942384599094738028569952190792440876674442355252980139799532702018467097321658735875309112762412369837837029491648930140424071479936263830182401741479936263830182

p_{10} = 48881231587580532981831660854602775481747622693643978194946084966243530015028717627956211799755692481994628906250000000000000000000000000000000000000000000000001

$(a \cdot b \ mod \ p)_{10}$ = 21166581721100677086502675258032460754855597933458367078963721661963558551780887747405864852932852679566932729213106474714504058242137032050781382720511624

El resultado de la multiplicación modular $(a \cdot b \ mod \ p)$, hecha por por el coprocesador (hardware) es también un número de 512 bits, escrito en hexadecimal; así como el resultado de la multiplicación hecha por Maple en decimal. Puede verificarse que el resultado es exactamente el mismo, sólo que uno está expresado en hexadecimal y otro está expresado en decimal. La Figura 5.1 muestra la simulación del coprocesador al calcular la multiplicación modular de Montgomery con números de 512 bits para la versión en hardware y la Figura 5.3 muestra la ejecución en Maple para los mismos números, así como el tiempo de cómputo que se consumió al realizar el cálculo.

Para la segunda prueba se utilizaron números a, b y p (primo) de 1024 bits, es decir; el coprocesador se escaló a 1024 bits para poder calcular la multiplicación modular utilizando el algoritmo de Montgomery en su versión hardware. Los números utilizados se muestran a continuación y están representados en hexadecimal para la versión hardware y para la versión software están representados en decimal:

a_{16} = 123456789987654321123456789987654321123456789987654321123456789054321AB01025BF05012340AB1020BF05C123400000000000C12345AB1025BF05123456789987654321123456789987654321123456789987654321123456789054321AB01025BF05012340AB1020BF05C123400000000000C12345AB1025BF05

b_{16} = 98765432112365478998765432123456789987654321123456789987654321123456789054321BBBB00CC12345678B4512AA0B4512AAA000000CC12345678B4512AA98765432112365478998765432123456789987654321123456789987654321123456789054321BBBB00CC12345678B4512AA0B4512AAA000000CC12345678B4512AA

$p_{16} =$ 325FB70E82CF04A18D5A1CA
951CC05305A644977685D0D97F
6D6388D9A1CF08E85B72089EC0
AFD84C3CD3ED8DC0CDB01FCC56
EAE2F08111C64A4423387A6538
DE000E6F8B90C0575941C44F83
741EBBBEEA3425210BEDA7C1CA
8E71AC2DBD123A2F1869F46038
E2A8295E5BB3675AA000000000
000000000000000000000001

$(a \cdot b \ mod \ p)_{16} =$ 2AD13181F9647E299
5B0A5EDF8A6DE5DE1505B00743D
92CFA16F1D0B2D463FD04B6D8F7
575C9D4D58E95AB21E8AA99C78A
4108482C5DAD20602E6E52A1B9F
325360A116C9DD0C38808CAF154
CCDDB890CEEE82E47936F8CFC75
EB67D2A8208C648E2FF1F685815
F1AB0801BFA34BA8FBD62B37F13
B3F79957E051B80BF1498C9

$a_{10} =$ 127835956254899273759502601 7088522637279681299925970086
125107774537552203318267737 6
317581315075186167773506134 8
681425038467609278726866613 2
603975130393161822197730847 6
493530034008965212730346237 2
657311115024673651408605884 8
228261278598051494190986283 1
705789615882589710786342499 2
022079936304198735339177549 3

$b_{10} =$ 107062613365331212661099085 5661722886208524023577644181 7
950610430587898424245060689 9
904941564351919108631756511 0
958573827909455877640840371
066414250189895839028091371 1
025188883821130466599696554 9
746644259475568164577725650 2
408120369079218806928970234 4
230958688434050812710405467 3
466921788662839509780406647 46

$p_{10} =$ 353737464016668451855824972 30030436643291137492731008553
057089413810552025753301229 4
512947342224925759920017160 0
342902209329192528622365365 7
202703470829874277114868164 0
625000000000000000000000000
000000000000000000000000000
000000000000000000000000000
000000000000000000000000000
000000000000000000000000001

$(a \cdot b \ mod \ p)_{10} =$ 3006723343841566494
024565521120581927325595574 7
219730991621900878140489027 5
836062991323659843438226801 0
604881310409558360951221547 8
985884158099954553225638830 8
346201565537654287316363097 3
533201187354798245644643366 4
209558078875971595315905001 1
093455377744179399370623149 6
153850570143874684624783396 6
061328585

El resultado de la multiplicación modular $(a \cdot b \ mod \ p)$, hecha por por el coprocesador (hardware) es también un número de 1024 bits, escrito en hexadecimal; así como el resultado de la multiplicación hecha por Maple en decimal. Puede verificarse que el resultado es exactamente el mismo, sólo que uno está expresado en hexadecimal y otro está expresado en decimal. La Figura 5.2 muestra la simulación del coprocesador al calcular la multiplicación modular de Montgomery con números de 1024 bits para la versión en hardware y la Figura 5.4 muestra la ejecución en Maple para los mismos números, así como el tiempo de cómputo que se consumió al realizar el cálculo.

5.1.1. Hardware vs Software

Se realizaron pruebas únicamente para números de 512 y 1024 bits, la Tabla 5.1 que se muestra a continuación, tiene una realación en tiempos de ejecución medido en segundos para dos versiones distintas, la versión *Software* hecha en Maple y de [8] vs la versión *Hardware*, con las pruebas que se hicieron en este trabajo.

Bits	Lenguaje C	Montgomery VHDL	Montgomery MAPLE
512	1.376	0.003229510	0.06
1024	5.814	0.012621650	0.07

Tabla 5.1: Tabla comparativa hardware-software

5.1.2. Simulación

En esta parte se muestran las simulaciones del coprocesador en VHDL para los dos números que se mostraron anteriormente, además en la simulación también se puede observar el tiempo de ejecución de cada uno de los escalamientos del coprocesador (512 ó 1024 bits).

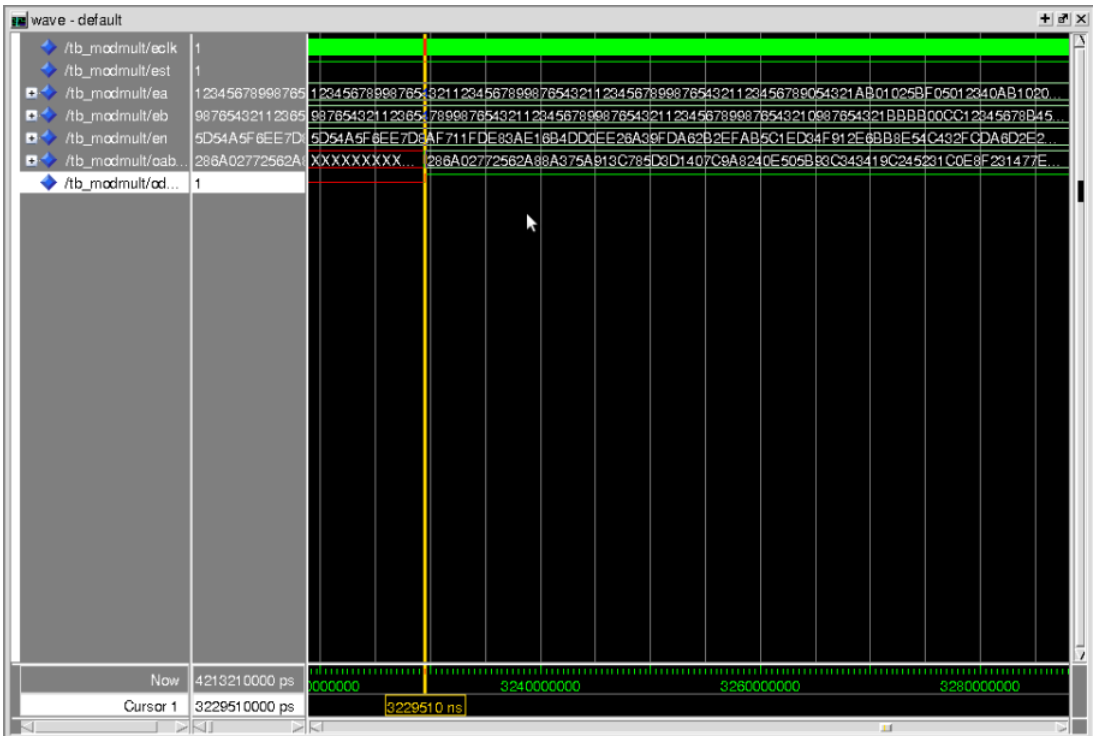

Figura 5.1: Simulación del coprocesador, para el cálculo de la multiplicación modular de Montgomery de números de 512 bits

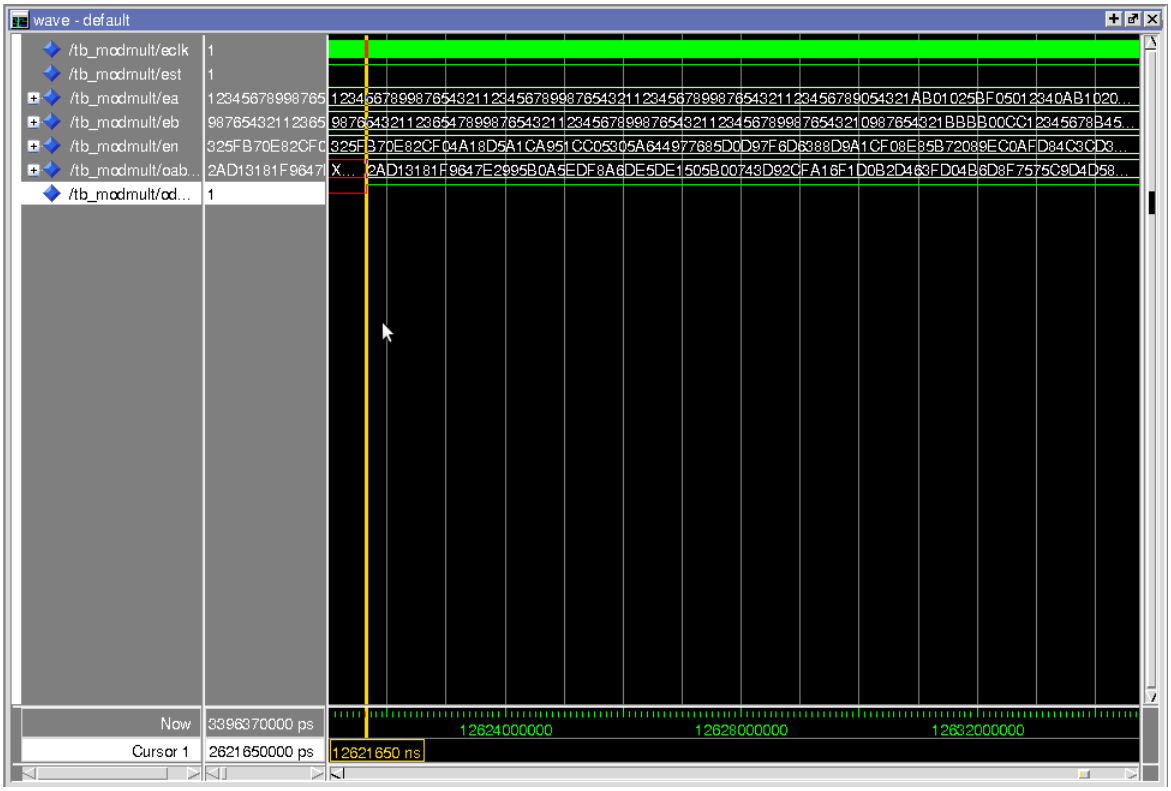

Figura 5.2: Simulación del coprocesador, para el cálculo de la multiplicación modular de Montgomery de números de 1024 bits

5.1.3. Pruebas en Maple

En esta parte se muestran las pruebas del coprocesador en su versión software que se implemento en Maple para los números, en base 10, que se mostraron anteriormente, además en las pruebas también se puede observar el tiempo de ejecución de cada uno de los cálculos.

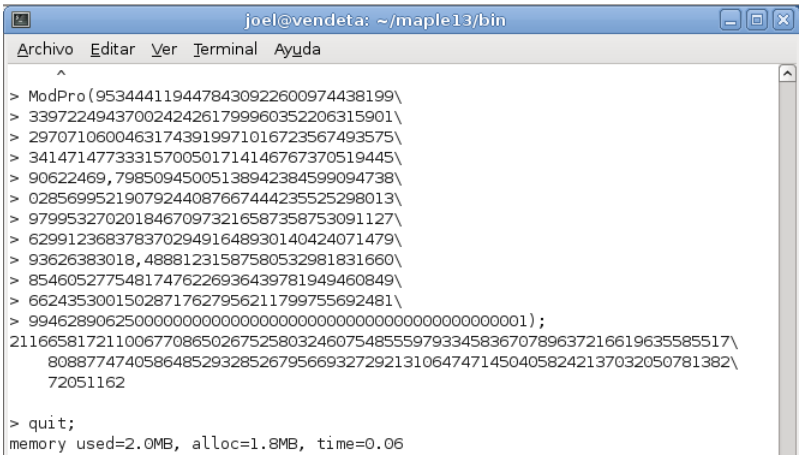

Figura 5.3: Simulación en Maple para los números de 512 bits

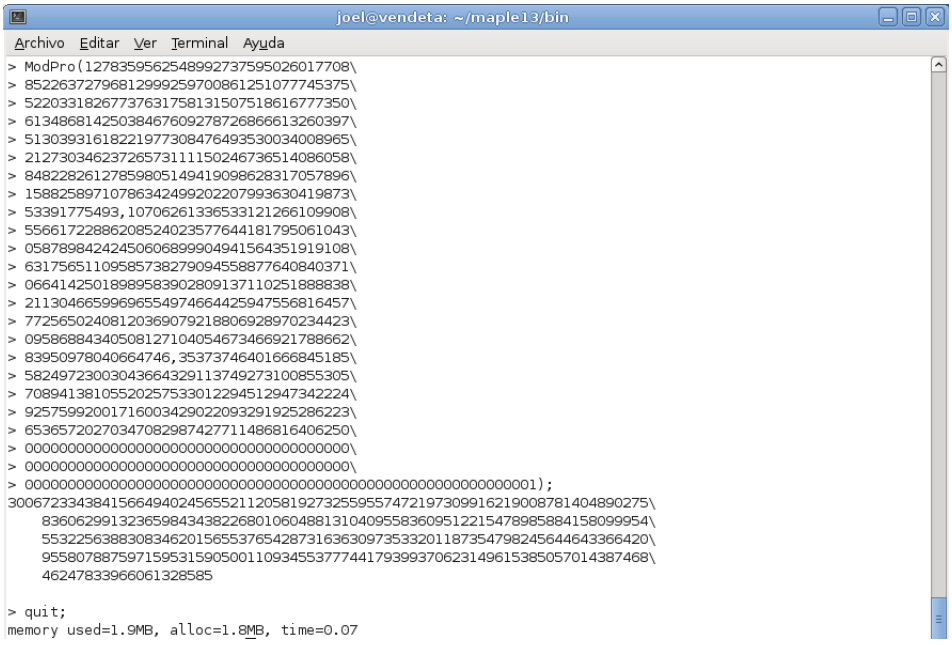

Figura 5.4: Simulación en Maple para los números de 1024 bits

5.2. Conclusiones

El coprocesador en la versión hardware se diseñó para que trabaje con números de 32 bits, además partiendo de esta base el coprocesador fue diseñado para ser escalable; es decir, que pueda aumentar su capacidad para números de 64, 128, 256, 512 y 1024, siendo los dos últimos rangos de prueba. Para el coprocesador de 32 bits, es muy secillo escarlo, ya que los operandos se pueden dividir en palabras de 32 bits, formando i arreglos de palabras de 32 bits formando así los números de 2^n bits. El coprocesador puede escalarse a cualquier número de bits, en este caso se usaron números de 2^n bits. El coprocesador alcanzará su máximo desempeño y eficiencia si los operandos, con los cuales se desea realizar la multiplicación modular, son muy grandes (números de 1024 ó 2048 bits por ejemplo) esto es debido a que el algoritmo de Montgomery trabaja a base de desplazamientos a la izquierda y a la derecha, por lo tanto el coprocesador puede realizar divisiones y multiplicaciones muy rápido. Por el contrario si los operandos son pequeños (números de 32 ó 64 bits por ejemplo) tardará más tiempo en calcular la multiplicación modular en comparación si utilizamos una calculadora aritmética. Esto se debe a que se necesitan algunos cálculos previos a utlizar el algoritmo de multiplicación modular de Montgomery, los cuales son muy costosos en tiempo de ejecución, específicamente el inverso modular del número primo. Para trabajos futuros, los módulos internos de coprocesador se pueden mejorar; es decir, se pueden hacer nuevas arquitecturas para mejorar el tiempo de ejecución. La versión software fue muy sencillo implementarla en comparación con la versión hardware. Para esta versión no fue necesario diseñar una base de 32 bits ni mucho menos escalar el coprocesador, internamente Maple se encarga de hacer arreglos de registros para poder representar los números con los que se está trabajando.
Se hicieron pruebas en Hardware y Software y se obtuvieron los mismos resultados, también se obtuvieron los tiempos de ejecución de las dos versiones y se hizo una tabla comparativa *hardware vs software*. Se comparó [8] con el coprocesador hecho en este trabajo, el coprocesador en hardware ganó en tiempo de ejecución al trabajo citado y la versión hecha en Maple.

Apéndice A

Códigos fuente

Archivo: `modpro.mws` (*Multiplicación Modular*)

```
1 # ##########################################################################
2 #
3 # Noyola Bautista Joel
4 #
5 # Algoritmo de multiplicación modular de Montgomery a*b mod n en Maple
6 #
7 # ##########################################################################
8
9 ModPro:=proc(a,b,p)
10      global n,r,ni,ri;
11      local k,am,bm,c;
12      n:=p;
13      k:=ceil(log[2](n));
14      r:=2^k;
15      igcdex(r,n,'ri','ni');
16      ni:=ni*(-1);
17      if ni<0 then
18           ni:=ni+r;
19      end if;
20      am:=a*r mod n;
21      bm:=b*r mod n;
22      c:=MonPro(am,bm);
23      c:=MonPro(c,1);
24      return c;
25 end proc;
```

Archivo: `monpro.mws` (*Algoritmo de reducción*)

```
1# ##############################################################################
2#
3# Noyola Bautista Joel
4#
5# Algoritmo de reducción de Montgomery en Maple
6#
7# ##############################################################################
8
9 MonPro:=proc(am,bm)
10      local t,m,u;
11      t:=am*bm;
12      m:=t*ni mod r;
13      u:=(t+m * n)/r;
14      if u>= n then
15          return (u-n);
16      else
17          return u;
18      end if;
19 end proc;
```

Archivo: `modmult.vhdl` (*Multiplicación Modular*)

```
1 # ##############################################################################
2 #
3 # Noyola Bautista Joel
4 #
5 # Algoritmo de multiplicación modular de Montgomery a*b mod n en vhdl,
6 # para números d n-bits
7 #
8 # ##############################################################################
9
10 library ieee;
11 use ieee.std_logic_1164.all;
12 use ieee.numeric_std.all;
13
14 use work.paquete.all;
15
16 entity modmult is
17         generic(bits: integer:=1023);
18     port(
19             clk,st:in std_logic;
20             a,b,n:in std_logic_vector(bits downto 0);
21             abmodn:out std_logic_vector(bits downto 0);
22             done:out std_logic
23             );
24 end entity modmult;
25
26 architecture beh of modmult is
27
28 component euclides is
29         generic(bits: integer:=1023);
30     port(
31             clk,st:in std_logic;
32             A,B:in std_logic_vector(bits downto 0);
33             ai,bi,d:out std_logic_vector((2*bits)+1 downto 0);
34             done:out std_logic
35         );
36 end component euclides;
37
38 component dominio is
39         generic(bits: integer:=1023);
40     port(
41             clk,st:in std_logic;
42             x,y,p:in std_logic_vector(bits downto 0);
43             r:in natural;
44             xr,yr:out std_logic_vector(bits downto 0);
45             done:out std_logic
46             );
47 end component dominio;
48
49 component monpro is
50         generic(bits: integer:=1023);
51     port(
52             clk,st:in std_logic;
53             r:in natural;
54             xr,yr:in std_logic_vector(bits downto 0);
55                 p:in std_logic_vector(bits downto 0);
56             pi:in std_logic_vector((2*bits)+1 downto 0);
57             xymodp:out std_logic_vector(bits downto 0);
58             done:out std_logic
59             );
```

```
60 end component monpro;
61
62 signal r:std_logic_vector(bits downto 0):=(others=>'0');
63
64 signal ste:std_logic;
65 signal Ae,Be:std_logic_vector(bits downto 0);
66 signal aie,bie,de:std_logic_vector((2*bits)+1 downto 0);
67 signal donee:std_logic;
68
69 signal std:std_logic;
70 signal xd,yd,pd:std_logic_vector(bits downto 0);
71 signal rd:natural;
72 signal xrd,yrd:std_logic_vector(bits downto 0);
73 signal doned:std_logic;
74
75 signal stm:std_logic;
76 signal rm:natural;
77 signal xrm,yrm:std_logic_vector(bits downto 0);
78 signal pm:std_logic_vector(bits downto 0);
79 signal pim:std_logic_vector((2*bits)+1 downto 0);
80 signal xymodpm:std_logic_vector(bits downto 0);
81 signal donem:std_logic;
82
83 signal estado:integer range 0 to 7;
84
85 begin
86
87     E0:euclides
88         generic map(
89                             bits
90                             )
91     port map(
92             clk,
93             ste,
94             Ae,
95             Be,
96             aie,
97             bie,
98             de,
99             donee
100            );
101
102    D0:dominio
103        generic map(
104                            bits
105                            )
106    port map(
107            clk,
108            std,
109            xd,
110            yd,
111            pd,
112            rd,
113            xrd,
114            yrd,
115            doned
116            );
117
118    M0:monpro
119        generic map(
120                            bits
121                            )
122    port map(
123            clk,
124            stm,
125            rm,
```

```
126             xrm,
127             yrm,
128             pm,
129             pim,
130             xymodpm,
131             donem
132             );
133     process(clk,donee,doned,donem)
134     begin
135         if clk'event and clk='1' then
136             case estado is
137                 when 0 =>
138                     if st='1' then
139                         rd<=calcularR(n);
140                         estado<=1;
141                     end if;
142                 when 1 =>
143                         r(rd)<='1';
144                         estado<=2;
145                 when 2 =>
146                         Ae<=r;
147                         Be<=n;
148                         estado<=3;
149                 when 3 =>
150                     ste<='1';
151                     if donee='1' then
152                         pim<=bie;
153                         ste<='0';
154                         estado<=4;
155                     end if;
156                 when 4 =>
157                     xd<=a;
158                     yd<=b;
159                     pd<=n;
160                     estado<=5;
161                 when 5 =>
162                     std<='1';
163                     if doned='1' then
164                         rm<=rd;
165                         xrm<=xrd;
166                         yrm<=yrd;
167                         pm<=n;
168                         std<='0';
169                         estado<=6;
170                     end if;
171                 when 6 =>
172                     stm<='1';
173                     if donem='1' then
174                         abmodn<=xymodpm;
175                         stm<='0';
176                         estado<=7;
177                     end if;
178                 when 7 =>
179                     done<='1';
180                     estado<=0;
181             end case;
182         end if;
183     end process;
184 end architecture beh;
```

Archivo: `monpro.vhdl` (*Reducción modular de Montgomery*)

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use ieee.std_logic_arith.all;
5 use ieee.std_logic_unsigned.all;
6
7 use work.paquete.all;
8
9 entity monpro is
10         generic(bits: integer:=1023);
11     port(
12             clk,st:in std_logic;
13             r:in natural;
14             xr,yr:in std_logic_vector(bits downto 0);
15                 p:in std_logic_vector(bits downto 0);
16             pi:in std_logic_vector((2*bits)+1 downto 0);
17             xymodp:out std_logic_vector(bits downto 0);
18             done:out std_logic
19             );
20 end entity monpro;
21
22 architecture beh of monpro is
23
24 signal estado:integer range 0 to 11;
25
26 signal ni:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
27 signal tni:std_logic_vector((4*bits)+3 downto 0):=(others=>'0');
28 signal suma:std_logic_vector(bits downto 0):=(others=>'0');
29
30 signal a,b,bi,c:std_logic_vector(bits downto 0):=(others=>'0');
31 signal n:std_logic_vector(bits downto 0):=(others=>'0');
32 signal modr:std_logic_vector(bits downto 0):=(others=>'0');
33 signal t:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
34 signal rn:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
35 signal u:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
36 signal ui:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
37 signal carry,bandera:std_logic;
38 signal shift:std_logic_vector(31 downto 0):=(others=>'0');
39
40 begin
41     xymodp<=c;
42     process(clk,r,ui,shift)
43     begin
44         if clk'event and clk='1' then
45             case estado is
46                 when 0 =>
47                     if st='1' then
48                         a<=xr;
49                         b<=yr;
50                         n<=p;
51                         ni<=pi;
52                         bi(0)<='1';
53                         bandera<='0';
54                         done<='0';
55                         estado<=1;
56                     else
57                         done<='0';
58                         estado<=0;
59                     end if;
```

```
60                  when 1 =>
61                      t<=a*b;
62                      estado<=2;
63                  when 2 =>
64                      tni<=t*ni;
65                      estado<=3;
66                  when 3 =>
67                      modr(r-1 downto 0)<=tni(r-1 downto 0);
68                      estado<=4;
69                  when 4 =>
70                      rn<=modr*n;
71                      estado<=5;
72                  when 5 =>
73                                          shift<=std_logic_vector(to_unsigned(r,32));
74                      ui<=t+rn;
75                      estado<=6;
76                  when 6 =>
77                                          u<=sr(ui,shift(9 downto 0));
78                      estado<=7;
79                  when 7 =>
80                      sumav(u(bits downto 0),not n,'1',suma,carry,bits+1);
81                      estado<=8;
82                  when 8 =>
83                      if carry='1' then
84                          c<=u(bits downto 0)-n;
85                      else
86                          c<=u(bits downto 0);
87                      end if;
88                      estado<=9;
89                  when 9 =>
90                      a<=c;
91                      b<=bi;
92                      estado<=10;
93                  when 10 =>
94                      if bandera='1' then
95                          estado<=11;
96                      else
97                          bandera<='1';
98                          estado<=1;
99                      end if;
100                 when 11 =>
101                     done<='1';
102                     estado<=0;
103             end case;
104         end if;
105     end process;
106 end architecture beh;
```

Archivo: `dominio.vhdl` (*Dominio de Montgomery*)

```
1 library ieee;
2 use ieee.std_logic_1164.ALL;
3 use ieee.numeric_std.all;
4
5 use work.paquete.all;
6
7 entity dominio is
8         generic(bits: integer:=1023);
9     port(
10             clk,st:in std_logic;
11             x,y,p:in std_logic_vector(bits downto 0);
12             r:in natural;
13             xr,yr:out std_logic_vector(bits downto 0);
14             done:out std_logic
15             );
16 end entity dominio;
17
18 architecture beh of dominio is
19
20 component divisorn is
21         generic(bits: integer:=1023);
22     port(
23                         clk,st:in std_logic;
24             dividendo:in std_logic_vector((2*bits)+1 downto 0);
25             divisor:in std_logic_vector(bits downto 0);
26                         cociente,residuo:out std_logic_vector(bits downto 0);
27             v,done:out std_logic
28                         );
29 end component divisorn;
30
31 signal ste:std_logic;
32 signal dividendoe: std_logic_vector((2*bits)+1 downto 0);
33 signal divisore,cocientee,residuoe:std_logic_vector(bits downto 0);
34 signal ve,donee:std_logic;
35
36 signal estado:integer range 0 to 5;
37 signal a,b:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
38 signal ar,br:std_logic_vector(bits downto 0):=(others=>'0');
39 signal do:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
40 signal shift:std_logic_vector(31 downto 0):=(others=>'0');
41
42 begin
43
44     div:divisorn
45         generic map(
46                         bits
47                         )
48     port map(
49             clk,
50             ste,
51             dividendoe,
52             divisore,
53             cocientee,
54             residuoe,
55             ve,
56             donee
57             );
58
59     dividendoe<=do;
```

```
60      divisore<=p;
61      xr<=ar;
62      yr<=br;
63
64      process(clk,donee,a,b,shift)
65      begin
66          if clk'event and clk='1' then
67              case estado is
68                  when 0 =>
69                      if st='1' then
70                                                      shift<=std_logic_vector(to_unsigned(r,32));
71                          a(bits downto 0)<=x;
72                          b(bits downto 0)<=y;
73                          estado<=1;
74                      else
75                          done<='0';
76                          estado<=0;
77                      end if;
78                  when 1 =>
79                                              a<=sl(a,shift(9 downto 0));
80                                              b<=sl(b,shift(9 downto 0));
81                      estado<=2;
82                  when 2 =>
83                      do<=a;
84                      ste<='1';
85                      if donee='1' then
86                          ar<=residuoe;
87                          ste<='0';
88                          estado<=3;
89                      end if;
90                  when 3 =>
91                      estado<=4;
92                  when 4 =>
93                      do<=b;
94                      ste<='1';
95                      if donee='1' then
96                          br<=residuoe;
97                          ste<='0';
98                          estado<=5;
99                      end if;
100                 when 5 =>
101                     done<='1';
102                     estado<=0;
103             end case;
104         end if;
105     end process;
106 end architecture beh;
```

Archivo: `euclides.vhdl` (*Algoritmo Extendido de Euclides*)

```
1 library ieee;
2 use ieee.std_logic_1164.ALL;
3 use ieee.std_logic_arith.ALL;
4 use ieee.std_logic_unsigned.all;
5 use ieee.numeric_std.all;
6
7 use work.paquete.all;
8
9 entity euclides is
10         generic(bits: integer:=1023);
11     port(
12             clk,st:in std_logic;
13             A,B:in std_logic_vector(bits downto 0);
14             ai,bi,d:out std_logic_vector((2*bits)+1 downto 0);
15             done:out std_logic
16         );
17 end entity euclides;
18
19 architecture beh of euclides is
20
21 component divisorn is
22         generic(bits: integer:=512);
23     port(
24                         clk,st:in std_logic;
25             dividendo:in std_logic_vector((2*bits)+1 downto 0);
26             divisor:in std_logic_vector(bits downto 0);
27                         cociente,residuo:out std_logic_vector(bits downto 0);
28             v,done:out std_logic
29                         );
30 end component divisorn;
31
32 signal ste:std_logic;
33 signal dividendoe: std_logic_vector((2*bits)+1 downto 0);
34 signal divisore,cocientee,residuoe:std_logic_vector(bits downto 0);
35 signal ve,donee:std_logic;
36 signal cero:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
37 signal menos:std_logic_vector((2*bits)+2 downto 0):=(others=>'1');
38 signal estado:integer range 0 to 9;
39 signal m:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
40 signal n:std_logic_vector(bits downto 0):=(others=>'0');
41 signal x1,y1:std_logic_vector(bits downto 0):=(others=>'0');
42 signal x,y:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
43 signal x2,y2:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
44 signal q:std_logic_vector(bits downto 0):=(others=>'0');
45 signal rest:std_logic_vector(bits downto 0):=(others=>'0');
46 signal qx,qy:std_logic_vector((2*bits)+1 downto 0):=(others=>'0');
47 signal carry:std_logic;
48 signal r:natural;
49 begin
50
51     div:divisorn
52         generic map(
53                         bits
54                         )
55     port map(
56             clk,
57             ste,
58             dividendoe,
59             divisore,
```

```
60              cocientee,
61              residuoe,
62              ve,
63              donee
64              );
65
66      ai<=x2;
67      bi<=y2;
68
69      dividendoe<=m;
70      divisore<=n;
71
72      process(clk,donee)
73      begin
74          if clk'event and clk='1' then
75              case estado is
76                  when 0 =>
77                      if st='1' then
78                          m(bits downto 0)<=A;
79                          n<=B;
80                          estado<=1;
81                      else
82                          done<='0';
83                          estado<=0;
84                      end if;
85                  when 1 =>
86                      if n=0 then
87                          d<=m;
88                          x(0)<='1';
89                          estado<=0;
90                      else
91                          y1(0)<='1';
92                          x2(0)<='1';
93                          r<=calcularR(A);
94                          estado<=2;
95                      end if;
96                  when 2 =>
97
98                      --Habilitación del divisor
99                      ste<='1';
100                     if donee='1' then
101                         q<=cocientee;
102                         rest<=residuoe;
103                         ste<='0';
104                         estado<=3;
105                             end if;
106                     ------------------------------
107
108                 when 3 =>
109                     qx<=q*x1;
110                     qy<=q*y1;
111                     estado<=4;
112
113                 when 4 =>
114
115                                         sumavec1(x2,not qx,'1',x,carry);
116                                         sumavec1(y2,not qy,'1',y,carry);
117                     estado<=5;
118
119                 when 5 =>
120                     m(bits downto 0)<=n;
121                     n<=rest;
122                     x2(bits downto 0)<=x1; x1<=x(bits downto 0);
123                     y2(bits downto 0)<=y1; y1<=y(bits downto 0);
124                     estado<=6;
125                 when 6 =>
```

```
126                     if n>0 then
127                         done<='0';
128                         estado<=2;
129                     else
130                         menos((2*bits)+1 downto 0)<=y2;
131                         estado<=7;
132                     end if;
133                 when 7 =>
134                     sumav1(cero,not menos((2*bits)+1 downto 0),'1',y2,carry,(2*bits)+2);
135                     estado<=8;
136                 when 8 =>
137                     cero(r)<='1';
138                     if y2<0 then
139                         sumav1(cero,y2,'0',y2,carry,(2*bits)+2);
140                     end if;
141                     estado<=9;
142                 when 9 =>
143                     done<='1';
144                     estado<=0;
145             end case;
146         end if;
147     end process;
148 end architecture beh;
```

Archivo: `divisorn.vhdl` (*Divisor*)

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 use work.paquete.all;
6
7 entity divisorn is
8         generic(bits: integer:=1023);
9         port(
10                      clk,st:in std_logic;
11             dividendo:in std_logic_vector((2*bits)+1 downto 0);
12             divisor:in std_logic_vector(bits downto 0);
13                      cociente,residuo:out std_logic_vector(bits downto 0);
14             v,done:out std_logic
15                      );
16 end entity divisorn;
17
18 architecture beh of divisorn is
19 signal estado:integer range 0 to 5;
20 signal sl:std_logic_vector((2*bits)+2 downto 0):=(others=>'0');
21 alias p:std_logic_vector(bits+1 downto 0) is sl((2*bits)+2 downto bits+1);
22 alias a:std_logic_vector(bits downto 0) is sl(bits downto 0);
23 signal suma,b:std_logic_vector(bits+1 downto 0);
24 signal cero:std_logic_vector(bits+1 downto 0):=(others=>'0');
25 signal contador:integer range 0 to bits;
26 signal carry:std_logic;
27
28 begin
29         residuo<=p(bits downto 0);
30         cociente<=a;
31         sumavec(p,not b,'1',suma,carry);
32         process(clk)
33         begin
34         if clk'event and clk='1' then
35             case estado is
36                 when 0 =>
37                         if st='1' then
38                         sl((2*bits)+1 downto 0)<=dividendo;
39                         b<='0' & divisor;
40                         v<='0';
41                         contador<=0;
42                         estado<=1;
43                     else
44                         done<='0';
45                             estado<=0;
46                     end if;
47                 when 1 =>
48                     if carry='1' then
49                         v<='1';
50                         estado<=0;
51                     else
52                         sl<=p(bits downto 0) & a & '0';
53                         contador<=contador+1;
54                         estado<=2;
55                     end if;
56                     if st='0' then
57                             done<='0';
58                             estado<=0;
59                         end if;
```

```
60
61                  when 2 | 3 =>
62                      if carry='1' then
63                          p<=suma;
64                          sl(0)<='1';
65                      else
66                          sl<=p(bits downto 0) & a & '0';
67                          contador<=contador+1;
68                          estado<=estado+1;
69                      end if;
70                      if st='0' then
71                              done<='0';
72                              estado<=0;
73                          end if;
74                  when 4 =>
75                      if carry='1' then
76                          p<=suma;
77                          sl(0)<='1';
78                      else
79                          sl<=p(bits downto 0) & a & '0';
80                          if contador=bits then
81                              estado<=5;
82                              contador<=0;
83                          else
84                              contador<=contador+1;
85                          end if;
86                      end if;
87                      if st='0' then
88                              done<='0';
89                              estado<=0;
90                          end if;
91                  when 5 =>
92                      if carry='1' then
93                          p<=suma;
94                          sl(0)<='1';
95                          estado<=0;
96                      else
97                          estado<=0;
98                      end if;
99                      done<='1';
100                     if st='0' then
101                             done<='0';
102                             estado<=0;
103                         end if;
104             end case;
105         end if;
106         end process;
107 end architecture beh;
```

Archivo: `paquete.vhdl` (*Funciones y procedimientos*)

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 package paquete is
6
7         procedure sumav(
8                 add1, add2:in std_logic_vector(1023 downto 0);
9                 cin:in std_logic;
10                signal suma:out std_logic_vector(1023 downto 0);
11                signal co:out std_logic;
12                n:in natural);
13
14        procedure sumavec(
15                add1, add2:in std_logic_vector(1024 downto 0);
16                cin:in std_logic;
17                signal suma:out std_logic_vector(1024 downto 0);
18                signal co:out std_logic
19        );
20
21        procedure sumav1(
22                add1, add2:in std_logic_vector(2047 downto 0);
23                cin:in std_logic;
24                signal suma:out std_logic_vector(2047 downto 0);
25                signal co:out std_logic;
26                n:in natural);
27
28        procedure sumavec1(
29                add1, add2:in std_logic_vector(2047 downto 0);
30                cin:in std_logic;
31                signal suma:out std_logic_vector(2047 downto 0);
32                signal co:out std_logic
33        );
34
35        function sl(
36        signal inp:std_logic_vector(2047 downto 0);
37                signal shift:std_logic_vector(9 downto 0)
38                ) return std_logic_vector;
39
40    function sr(
41        signal inp:std_logic_vector(2047 downto 0);
42                signal shift:std_logic_vector(9 downto 0)
43        ) return std_logic_vector;
44
45    function calcularR(
46        signal s:std_logic_vector(1023 downto 0)
47        ) return natural;
48
49 end package paquete;
50
51 package body paquete is
52
53        procedure sumav(
54                add1, add2:in std_logic_vector(1023 downto 0);
55                cin:in std_logic;
56                signal suma:out std_logic_vector(1023 downto 0);
57                signal co:out std_logic;
58                n:in natural) is
59                        variable c:std_logic;
```

```
60                      variable i:integer;
61              begin
62                      c:=cin;
63                      for i in 0 to n-1 loop
64                              suma(i)<=add1(i)xor add2(i) xor c;
65                              c:=(add1(i)and add2(i))or (add1(i)and c)or (add2(i)and c);
66                      end loop;
67                      co<=c;
68      end sumav;
69
70      procedure sumavec(
71              add1, add2:in std_logic_vector(1024 downto 0);
72              cin:in std_logic;
73              signal suma:out std_logic_vector(1024 downto 0);
74              signal co:out std_logic
75      ) is
76                      variable c:std_logic;
77                      variable i:integer;
78              begin
79                      c:=cin;
80                      for i in 0 to 1024 loop
81                              suma(i)<=add1(i)xor add2(i) xor c;
82                              c:=(add1(i)and add2(i))or (add1(i)and c)or (add2(i)and c);
83                      end loop;
84                      co<=c;
85      end sumavec;
86
87      procedure sumav1(
88              add1, add2:in std_logic_vector(2047 downto 0);
89              cin:in std_logic;
90              signal suma:out std_logic_vector(2047 downto 0);
91              signal co:out std_logic;
92              n:in natural) is
93                      variable c:std_logic;
94                      variable i:integer;
95              begin
96                      c:=cin;
97                      for i in 0 to n-1 loop
98                              suma(i)<=add1(i)xor add2(i) xor c;
99                              c:=(add1(i)and add2(i))or (add1(i)and c)or (add2(i)and c);
100                     end loop;
101                     co<=c;
102     end sumav1;
103
104
105     procedure sumavec1(
106             add1, add2:in std_logic_vector(2047 downto 0);
107             cin:in std_logic;
108             signal suma:out std_logic_vector(2047 downto 0);
109             signal co:out std_logic
110     ) is
111                     variable c:std_logic;
112                     variable i:integer;
113             begin
114                     c:=cin;
115                     for i in 0 to 2047 loop
116                             suma(i)<=add1(i)xor add2(i) xor c;
117                             c:=(add1(i)and add2(i))or (add1(i)and c)or (add2(i)and c);
118                     end loop;
119                     co<=c;
120     end sumavec1;
121
122     function sl(
123     signal inp:std_logic_vector(2047 downto 0);
124             signal shift:std_logic_vector(9 downto 0)
125             ) return std_logic_vector is
```

```
126                 VARIABLE temp1: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
127                 VARIABLE temp2: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
128                 VARIABLE temp3: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
129                 VARIABLE temp4: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
130                 VARIABLE temp5: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
131                 VARIABLE temp6: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
132                 VARIABLE temp7: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
133                 VARIABLE temp8: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
134                 VARIABLE temp9: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
135         VARIABLE outp: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
136                 begin
137         ----- 1er sl -----
138         IF (shift(0)='0') THEN
139         temp1 := inp;
140     ELSE
141         temp1(0) := '0';
142         FOR i IN 1 TO inp'HIGH LOOP
143                 temp1(i) := inp(i-1);
144         END LOOP;
145     END IF;
146         ----- 2do sl -----
147         IF (shift(1)='0') THEN
148                 temp2 := temp1;
149         ELSE
150         FOR i IN 0 TO 1 LOOP
151                 temp2(i) := '0';
152         END LOOP;
153         FOR i IN 2 TO inp'HIGH LOOP
154                 temp2(i) := temp1(i-2);
155         END LOOP;
156         END IF;
157         ----- 3ro sl -----
158         IF (shift(2)='0') THEN
159         temp3 := temp2;
160         ELSE
161         FOR i IN 0 TO 3 LOOP
162                 temp3(i) := '0';
163         END LOOP;
164         FOR i IN 4 TO inp'HIGH LOOP
165                 temp3(i) := temp2(i-4);
166         END LOOP;
167         END IF;
168         ----- 4to sl -----
169         IF (shift(3)='0') THEN
170         temp4 := temp3;
171         ELSE
172         FOR i IN 0 TO 7 LOOP
173                 temp4(i) := '0';
174         END LOOP;
175         FOR i IN 8 TO inp'HIGH LOOP
176                 temp4(i) := temp3(i-8);
177         END LOOP;
178         END IF;
179         ----- 5to sl -----
180         IF (shift(4)='0') THEN
181         temp5 := temp4;
182         ELSE
183         FOR i IN 0 TO 15 LOOP
184                 temp5(i) := '0';
185         END LOOP;
186         FOR i IN 16 TO inp'HIGH LOOP
187                 temp5(i) := temp4(i-16);
188         END LOOP;
189         END IF;
190         ----- 6to sl -----
191         IF (shift(5)='0') THEN
```

```
192         temp6 := temp5;
193         ELSE
194         FOR i IN 0 TO 31 LOOP
195                 temp6(i) := '0';
196         END LOOP;
197         FOR i IN 32 TO inp'HIGH LOOP
198                 temp6(i) := temp5(i-32);
199         END LOOP;
200         END IF;
201         ----- 7to sl -----
202         IF (shift(6)='0') THEN
203         temp7 := temp6;
204         ELSE
205         FOR i IN 0 TO 63 LOOP
206                 temp7(i) := '0';
207         END LOOP;
208         FOR i IN 64 TO inp'HIGH LOOP
209                 temp7(i) := temp6(i-64);
210         END LOOP;
211         END IF;
212         ----- 8to sl -----
213         IF (shift(7)='0') THEN
214         temp8 := temp7;
215         ELSE
216         FOR i IN 0 TO 127 LOOP
217                 temp8(i) := '0';
218         END LOOP;
219         FOR i IN 128 TO inp'HIGH LOOP
220                 temp8(i) := temp7(i-128);
221         END LOOP;
222         END IF;
223         ----- 9to sl -----
224         IF (shift(8)='0') THEN
225         temp9 := temp8;
226         ELSE
227         FOR i IN 0 TO 255 LOOP
228                 temp9(i) := '0';
229         END LOOP;
230         FOR i IN 256 TO inp'HIGH LOOP
231                 temp9(i) := temp8(i-256);
232         END LOOP;
233         END IF;
234
235         ----- 10to sl -----
236         IF (shift(9)='0') THEN
237                 outp := temp9;
238     ELSE
239         FOR i IN 0 TO 511 LOOP
240                 outp(i) := '0';
241         END LOOP;
242         FOR i IN 512 TO inp'HIGH LOOP
243                         outp(i) := temp9(i-512);
244         END LOOP;
245         END IF;
246                 return outp;
247         end sl;
248
249     function sr(
250         signal inp:std_logic_vector(2047 downto 0);
251                 signal shift:std_logic_vector(9 downto 0)
252         ) return std_logic_vector is
253         VARIABLE temp1: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
254         VARIABLE temp2: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
255         VARIABLE temp3: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
256         VARIABLE temp4: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
257         VARIABLE temp5: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
```

```
258         VARIABLE temp6: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
259         VARIABLE temp7: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
260         VARIABLE temp8: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
261         VARIABLE temp9: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
262     VARIABLE outp: STD_LOGIC_VECTOR (2047 DOWNTO 0):=(others=>'0');
263         begin
264                 ---- 1er sr -----
265         IF (shift(0)='0') THEN
266         temp1 := inp;
267     ELSE
268         temp1(2047) := '0';
269         FOR i IN 0 TO inp'HIGH-1 LOOP
270                 temp1(i) := inp(i+1);
271         END LOOP;
272     END IF;
273         ---- 2do sr -----
274         IF (shift(1)='0') THEN
275                 temp2 := temp1;
276         ELSE
277         FOR i IN inp'HIGH TO inp'HIGH-1 LOOP
278                 temp2(i) := '0';
279         END LOOP;
280         FOR i IN 0 TO inp'HIGH-2 LOOP
281                 temp2(i) := temp1(i+2);
282         END LOOP;
283         END IF;
284         ---- 3ro sr -----
285         IF (shift(2)='0') THEN
286         temp3 := temp2;
287         ELSE
288         FOR i IN inp'HIGH TO inp'HIGH-3 LOOP
289                 temp3(i) := '0';
290         END LOOP;
291         FOR i IN 0 TO inp'HIGH-4 LOOP
292                 temp3(i) := temp2(i+4);
293         END LOOP;
294         END IF;
295         ---- 4to sr -----
296         IF (shift(3)='0') THEN
297         temp4 := temp3;
298         ELSE
299         FOR i IN inp'HIGH TO inp'HIGH-7 LOOP
300                 temp4(i) := '0';
301         END LOOP;
302         FOR i IN 0 TO inp'HIGH-8 LOOP
303                 temp4(i) := temp3(i+8);
304         END LOOP;
305         END IF;
306         ---- 5to sr -----
307         IF (shift(4)='0') THEN
308         temp5 := temp4;
309         ELSE
310         FOR i IN inp'HIGH TO inp'HIGH-15 LOOP
311                 temp5(i) := '0';
312         END LOOP;
313         FOR i IN 0 TO inp'HIGH-16 LOOP
314                 temp5(i) := temp4(i+16);
315         END LOOP;
316         END IF;
317         ---- 6to sr -----
318         IF (shift(5)='0') THEN
319         temp6 := temp5;
320         ELSE
321         FOR i IN inp'HIGH TO inp'HIGH-31 LOOP
322                 temp6(i) := '0';
323         END LOOP;
```

```
324         FOR i IN 0 TO inp'HIGH-32 LOOP
325                 temp6(i) := temp5(i+32);
326         END LOOP;
327         END IF;
328         ---- 7to sr -----
329         IF (shift(6)='0') THEN
330         temp7 := temp6;
331         ELSE
332         FOR i IN inp'HIGH TO inp'HIGH-63 LOOP
333                 temp7(i) := '0';
334         END LOOP;
335         FOR i IN 0 TO inp'HIGH-64 LOOP
336                 temp7(i) := temp6(i+64);
337         END LOOP;
338         END IF;
339         ---- 8to sr -----
340         IF (shift(7)='0') THEN
341         temp8 := temp7;
342         ELSE
343         FOR i IN inp'HIGH TO inp'HIGH-127 LOOP
344                 temp8(i) := '0';
345         END LOOP;
346         FOR i IN 0 TO inp'HIGH-128 LOOP
347                 temp8(i) := temp7(i+128);
348         END LOOP;
349         END IF;
350         ---- 9to sr -----
351         IF (shift(8)='0') THEN
352         temp9 := temp8;
353         ELSE
354         FOR i IN inp'HIGH TO inp'HIGH-255 LOOP
355                 temp9(i) := '0';
356         END LOOP;
357         FOR i IN 0 TO inp'HIGH-256 LOOP
358                 temp9(i) := temp8(i+256);
359         END LOOP;
360         END IF;
361         ---- 10to sr -----
362         IF (shift(9)='0') THEN
363                 outp := temp9;
364     ELSE
365         FOR i IN inp'HIGH TO inp'HIGH-511 LOOP
366                 outp(i) := '0';
367         END LOOP;
368         FOR i IN 0 TO inp'HIGH-512 LOOP
369                         outp(i) := temp9(i+512);
370         END LOOP;
371         END IF;
372                 return outp;
373         end sr;
374
375
376     function calcularR(
377         signal s:std_logic_vector(1023 downto 0)
378         ) return natural is
379         begin
380             for i in 1023 downto 0 loop
381                 if s(i)='1' then
382                     return i+1;
383                 end if;
384             end loop;
385                         return 0;
386     end calcularR;
387
388 end package body;
```

Bibliografía

[1] Herbert S. Zuckerman. Niven I. *Introducción a la teoría de números*. Limusa, 1976.

[2] Lawrence C. Washington. Wade Trappe. *Introduction to Cryptography with Coding Theory*. Prentice Hall, 2001.

[3] Hugo Rincón Mejía y César Rincón Orta y Alejandro Bravo Mojica. *Álgebra Superior*. Las prensas de ciencias, 2006.

[4] Richard Johnsonbaugh. *Matemáticas Discretas*. Richard Johnsonbaugh, 2005.

[5] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2006.

[6] Çetin Kaya. Koç. *Cryptographic Engineering*. Springer, 2009.

[7] David A. Patterson and Jhon L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, 2009.

[8] Çetin Kaya Koç and Tolga Acar. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 1996.

[9] D. J Guan. Montgomery algorithm for modular multiplication. *Department of Computer of Science, National Sun Yar-Sen Univeristy, Kaohsiung, Taiwan*, 2003.

[10] Alexandre F. Tenca and Çetin Kaya Koç. Montgomery algorithm for modular multiplication. *IEEE Micro*, 52:1215–1221, 2003.