COMPARACIÓN DE LA EFICIENCIA EN HARDWARE DE LOS CIFRADORES DE FLUJO GRAIN, MICKEY-128 Y TRIVIUM DE ECRYPT.

por

Pérez Camacho Blanca Nydia
L.C.C., BUAP

Tesis sometida como requisito parcial
para obtener el grado de

**MAESTRO EN CIENCIAS
EN CIENCIAS COMPUTACIONALES**

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica
Febrero, 2008
Tonantzintla, Puebla. MÉXICO

Supervisada por:

Dr. René A. Cumplido Parra
Investigador del INAOE

©INAOE 2008
Derechos reservados
El autor otorga al INAOE el permiso de reproducir y
distribuir copias de esta tesis en su totalidad o en partes

COMPARACIÓN DE LA EFICIENCIA EN HARDWARE DE LOS CIFRADORES DE FLUJO GRAIN, MICKEY-128 Y TRIVIUM DE ECRYPT.

Estudiante

Pérez Camacho Blanca Nydia

Supervisor

Dr. René A. Cumplido Parra

Departamento de Ciencias Computacionales

Instituto Nacional de Astrofísica, Óptica y Electrónica

Tonantzintla, Puebla. MÉXICO

Febrero, 2008

Abstract

Actually is increasing the use of mobile devices which are used to send information. These devices are portable and give facilities to send and receive information. When these devices send the information by these ways this is more vulnerable to different attacks and could corrupt it. To keep the information secure there are cryptography standards, which are used for cipher the information. The cryptographic algorithms are divided in three families: Hash algorithms, asymmetric algorithms and symmetric algorithms. The most important algorithms in each family are MD5 and SHA1 Hash algorithms; RSA asymmetric algorithm; DES, IDEA and AES (OFB mode) symmetric algorithm. Ecrypt has the main objective of use stream ciphers new standards with eStream project which focus is stream ciphers with software and/or hardware implementation, stream ciphers are compare with AES on OFB mode, because this works like stream cipher. The finalist stream ciphers will be proposed as standards. Stream ciphers of hardware implementation which are on fourth phase are Grain, Mickey-128 and Trivium, which are worked on the present thesis with the objective to compare them on same conditions and to define the restrictions under each one are feasible their work. For these stream ciphers have done ascending design on VHDL language in ModelSim program the synthesis have done in Xilinx ISE 8.2i program using Spartan3 development target. Optimizing by velocity and given the highest parallel grade each one stream cipher, was Grain which gave the highest frequency, following Trivium and last was Mickey-128; optimizing by area on ascending order was Grain, following Trivium and the last was Mickey-128 because it needs most area than others.

Resumen

Actualmente, se está incrementando el uso de los dispositivos móviles para realizar el envío de información. Ya que éstos son portátiles y dan la facilidad de hacer el envío y la recepción de la información. Pero, cuando se hace el envío de la información por estos medios, ésta es más susceptible a diversos ataques en los que se pueda corromper la misma. Para mantener segura la información existen estándares en criptografía, los cuales son usados para cifrar la información. Los algoritmos criptográficos se dividen en tres familias: algoritmos Hash, algoritmos asimétricos y algoritmos simétricos. Entre los algoritmos más importantes de cada familia encontramos MD5 y SHA1 son algoritmos Hash; RSA es un algoritmo asimétrico; DES, IDEA y AES (en modo OFB) son algoritmos simétricos. Con el objetivo de usar nuevos estándares en cifradores de flujo es que la organización Ecrypt diseñó el proyecto eStream, el cual está enfocado en cifradores de flujo con implementación en software y / o hardware, los cifradores de flujo son comparados con el algoritmo AES en modo OFB, ya que se comporta como cifrador de flujo. Los algoritmos que lleguen a la final serán propuestos como estándares. De entre los algoritmos de implementación hardware que han pasado a la cuarta fase se encuentran Grain, Mickey-128 y Trivium, los cuales se toman en el presente trabajo con el objetivo de compararlos bajo las mismas condiciones y definir las restricciones en las cuales es factible el uso de cada uno de ellos. Para estos algoritmos se hizo un diseño ascendente en el lenguaje VHDL en ModelSim, la síntesis se realizó en el programa Xilinx ISE 8.2i usando la tarjeta de desarrollo Spartan3. Cuando se optimizó por velocidad el algoritmo y colocando en su máximo grado de paralelización a cada uno de los cifradores de flujo, fue Grain el que presentó la frecuencia más alta, siguiéndole Trivium y por último Mickey-128; en la optimización por área en orden ascendente por uso de la misma es Grain, después Trivium y Mickey-128 es el que requiere mayor.

Agradecimientos

Al Dr. René Cumplido Parra

Por sus consejos, su apoyo y por darme el respaldo que necesitaba para continuar con mis proyectos.

A la Dra. Claudia Feregrino Uribe

Por que una de las personas que más me ayudo y apoyo en todos los sentidos.

A mis compañeros

Por su apoyo y amistad. En especial a Néstor, Sayde y Esteban, a quiénes admiro y nunca olvidaré.

A mi hijo Yael Antonio

Por la sonrisa y alegría con la que me recibía al llegar a casa, por darme la fuerza necesaria para continuar, y que con sólo recordarlo me iluminaba la vida. Te amo muñequito.

A mi esposo José Antonio

Por su amor y por cada uno de los abrazos que me daba en los momentos en los que más los necesitaba. También, por saber escucharme y aconsejarme.

A mis padres Blanca y José Luis

Quienes me han apoyado día a día sabiendo escucharme y aconsejarme. Además, me han ayudado a madurar, crecer y enfrentar las adversidades. Gracias por su perseverancia y su amor.

A mis hermanos Luis, Laura y Rafael

Quienes me alegraban el día con sus ocurrencias, por saber escucharme y no dejarme sola.

Índice

Lista de Tablas 5
Lista de Figuras 7
1. Introducción 9
1.1 Introducción 9
1.2 Objetivo general 14
1.3 Objetivos específicos 14
1.4 Metodología 15
1.5 Contenido de los capítulos 15
2. Cifradores de flujo: Grain, Mickey-128 y Trivium 17
2.1 Criptografía 17
2.1.1 Historia 17
2.1.2 Tipos de Criptografía 19
2.1.3 Especificaciones de los cifradores de flujo 26
2.1.4 Cifradores de flujo candidatos a estándares en Hardware 28
2.2 Grain 28
2.2.1 Algoritmo de Grain 28
2.2.2 Resultados de trabajos realizados sobre Grain 30
2.3 Mickey-128 32

2.3.1 Algoritmo de Mickey-128 33

2.3.2 Resultados de trabajos realizados sobre Mickey-128 ...38

2.4 Trivium 39

2.4.1 Algoritmo de Trivium 40

2.4.2 Resultados de trabajos realizados sobre Trivium 42

2.5 Resultados de trabajos, comparativas de Grain, Mickey-128 y Trivium 44

2.6 Conclusión del capitulo y panorama del capítulo tres 45

3. Arquitecturas de Grain, Mickey-128 y Trivium 47

3.1 Paralelización de los cifradores 47

3.2 Grain 49

3.2.1 Relación entre algoritmo y componentes 50

3.3 Mickey-128 52

3.3.1 Relación entre algoritmo y componentes 54

3.4 Trivium 56

3.4.1 Relación entre algoritmo y componentes 58

3.5 Conclusión del capítulo y panorama del capítulo cuatro 60

4. Resultados de implementación 61

4.1 Ámbito de desarrollo de la tesis 61

4.2 Resultados de Grain 62

4.2.1 Comparación con otros trabajos 63

4.3 Resultados de Mickey-128 64

4.3.1 Comparación con otros trabajos 65

4.4 Resultados de Trivium 66

4.4.1 Comparación con otros trabajos 68

4.5 Comparación de los resultados de las arquitecturas Grain, Mickey-128 y Trivium 69

4.6 Conclusión del capítulo y panorama del capítulo cinco 71

5. Conclusiones 73

5.1 Conclusiones 73

5.2 Revisión de objetivos 74

Bibliografía 79

Apéndice A. Grain 83

Apéndice B. Mickey-128 103

Apéndice C. Trivium 127

Lista de Tablas

2.1 Resultados de diferentes grados de paralelización en el dispositivo
xc3s50pq208-5 31
2.2 Comparación de los resultados para Grain con 16 bits en paralelo 32
2.3 Resultados de síntesis del dispositivo xcv50ecs144 39
2.4 Comparación de los resultados para Mickey-128 con 1 bit 39
2.5 Resultados de implementación del cifrador de flujo Trivium en el
dispositivo xc3s400fg320-5 43
2.6 Comparación de los resultados para Trivium con 64 bits en paralelo 43
2.7 Comparación de las arquitecturas optimizadas en mínimo de área
en una Xilinx Spartan3 44
2.8 Comparación de arquitecturas optimizadas en base a la velocidad
de procesamiento / área 45
2.9 Comparación de los cifradores de flujo Mickey-128 y Trivium con
respecto a la velocidad de procesamiento / área 45
3.1 Características básicas de los cifradores de flujo 48
4.1 Resultados de síntesis de la arquitectura Grain optimizados por
velocidad 62
4.2 Eficiencia de la arquitectura Grain con diferentes grados de
paralelización optimizadas por velocidad 63
4.3 Comparación con trabajos anteriores 64

4.4 Resultados de síntesis de la arquitectura Mickey-128 optimizados por velocidad 64
4.5 Eficiencia de la arquitectura Mickey-128 con diferentes tamaños de iv optimizadas por velocidad 65
4.6 Comparación con trabajos anteriores 66
4.7 Resultados de síntesis de la arquitectura Trivium optimizadas por velocidad 66
4.8 Eficiencia de la arquitectura Trivium con diferentes grados de paralelización 67
4.9 Comparación con trabajos anteriores sobre Trivium 68
4.10 Comparación de las arquitecturas optimizadas por el mínimo de área 69
4.11 Comparación de las arquitecturas optimizadas por el máx. de velocidad y máx. grado de paralelización 69
4.12 Comparación en diferentes grados de paralelización de Grain y Trivium 70

Lista de Figuras

1.1 Esquema general de cifrado 10
1.2 Esquema del cifrador de Flujo 11
2.1 Esquema de los algoritmos asimétricos 22
2.2 Esquema de los algoritmos simétricos 23
2.3 Esquema de un mensaje en el envío y recepción 27
2.4 Esquema general de Grain 29
2.5 Grain en etapa de inicialización 29
2.6 Esquema general de Mickey-128 32
2.7 Control_bit=0 ………..36
2.8 Control_bit=1 36
2.9 Registro S 38
2.10 Esquema de Trivium 40
3.1 Interfaz de los cifradores de flujo 48
3.2 Arquitectura de Grain en el caso base 49
3.3Arquitectura de Grain para dos bits en paralelo 50
3.4 Arquitectura ascendente de Mickey-128 53
3.5 Arquitectura del componente compo_clock_kg 53
3.6 Arquitectura del componente compo_clockr 53
3.7 Arquitectura del componente compo_clocks 54
3.8 Arquitectura de Trivium en caso base 57
3.9 Arquitectura de Trivium con dos bits en paralelo 57

A.1 Componente Grain generado por la herramienta Xilinx ISE 8.2i 83
A.2 Diagrama RTL de Grain generado por la herramienta Xilinx ISE 8.2i 83
B.1 Componente Mickey-128 generado por la herramienta Xilinx
ISE 8.2i 103
B.2 Diagrama RTL de Mickey-128 generado por la herramienta Xilinx
ISE 8.2i 103
C.1 Componente Trivium generado por la herramienta Xilinx ISE 8.2i 127
C.2 Diagrama RTL de Trivium generado por la herramienta Xilinx
ISE 8.2i 127

Capítulo 1

Introducción

En este capítulo se da una breve descripción de los cifradores de flujo y de la importancia que representan en el mundo de la telecomunicación, de ahí el porqué buscar nuevos estándares en este tipo de cifradores. En lo anterior radica la base del proyecto eCrypt, el cual busca definir los candidatos a ser estándares para ser utilizados en aparatos móviles tal como lo son los teléfonos celulares.

1.1 Introducción.

En el mundo actual se maneja una gran cantidad de información, la cual es enviada y recibida a través de medios inalámbricos o alambrados, como por ejemplo enviar un mensaje de un teléfono celular a otro, o el hacer un traspaso de cuenta usando Internet. Gran parte de la información que se maneja es confidencial, de ahí que es necesario mantenerla protegida de los intrusos y / o de terceras personas interesadas en obtener algún beneficio de ella.

Para mantener protegida la información en el momento de su transferencia se deben cubrir cuatro factores, los cuales son: confidencialidad, autenticación, no repudio e integridad.

Los estándares usados para cifrar la información cubren varios de estos factores y son usados en la transferencia de la información que se hace por medio de la red. En las redes alambradas la trasmisión de la información es segura, en primera, debido a que es más difícil que terceras personas

puedan interferir la información, y en segunda, por que se cifra la información en el envío, pero al mismo tiempo resultan ser imprácticas si lo que se requiere es hacer el envío de la información a un medio móvil. En el caso de los móviles resulta ser más práctico el uso de las redes inalámbricas, las cuales son susceptibles a ataques, ya que es posible para terceras personas interceptar los mensajes y modificarlos o bien usarlos en beneficio propio.

Existen varios tipos de información de entre los cuales encontramos: números y mensajes confidenciales, transacciones bancarias (EFT) y tarjetas ATM, contraseñas, comercio electrónico, correos electrónicos y acceso remoto, video y voz (VoIP), y el video digital. Para poder hacer el envío de la información es que se ha ido incrementando el uso de medios inalámbricos. Una forma de proveer de confidencialidad, y en general de seguridad a la información es mediante el uso de la criptografía.

La criptografía tiene varios objetivos, por un lado, haciendo uso de las funciones hash para el intercambio seguro de llaves y envío de documentos con firma digital, y por el otro lado es el control de los accesos a información confidencial. Los cifradores se pueden clasificar como cifradores de flujo y de bloque. En la Fig. 1.1 se muestra un esquema general de cifrado, el algoritmo de cifrado toma el texto plano y llave con los cuales trabaja para generar el texto cifrado.

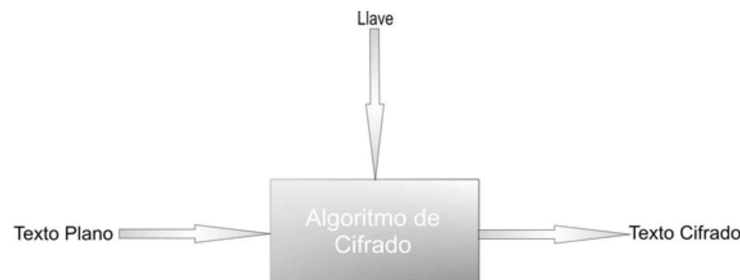

Fig. 1.1 Esquema general de cifrado

Los cifradores de bloque, van formando bloques de texto, cada uno de ellos es cifrado con la llave (*key*) y con el vector de inicialización (*IV*) usando la función que describa el algoritmo usado; mientras que, un cifrador de flujo

genera un flujo de llave (*keystream*), el cual se obtiene con un procedimiento que hace uso de la llave y del vector de inicialización, dicho keystream se combina con el texto plano por medio de una función XOR, ver Fig. 1.2, “los cifradores de flujo tienen diferentes propiedades de implementación con relación a los cifradores de bloque lo cual restringe el criptoanálisis”[1]

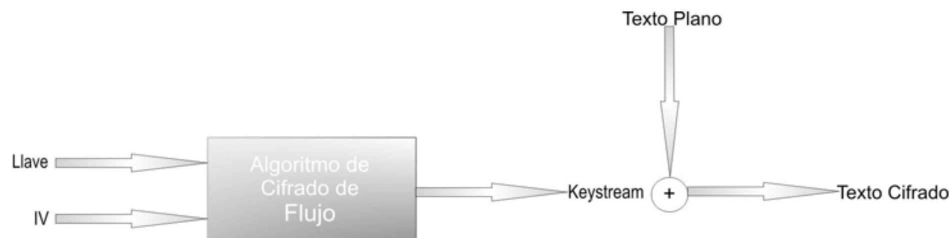

Fig. 1.2 Esquema del Cifrador de Flujo

En “el descifrado se toma la diferencia del keystream con el texto cifrado, lo cual es la misma operación de la función XOR”[1]. Además, los cifradores de flujo no sufren de errores de propagación, esto debido a que se hace manejo individual de bits para realizar el cifrado y descifrado.

Debido a la velocidad de procesamiento y simplicidad es que los cifradores de flujo son usados por varios estándares de comunicación, especialmente los inalámbricos como IEEE 802.11b y Bluetooth.

Algunos de los algoritmos de cifrado de flujo usados en los dispositivos móviles son A5/1 en GSM (telefonía celular), E0 en Bluetooth (radiocomunicaciones de corto alcance) y RC4 en IEEE 802.11b (redes inalámbricas), además una de las ventajas de los cifradores de flujo ante los de bloque es que “la generación del keystream de un cifrador de flujo es significativamente más rápida que la de un cifrador de bloque” [1]. Otra ventaja que tienen los cifradores de flujo con respecto a los cifradores de bloque es que sólo se toma una vez la llave y el vector de inicialización, esto en el momento de cifrar, por lo que puede ir generando el keystream bit a bit de manera consecutiva. Existen vulnerabilidades en los cifradores de flujo al momento de la transmisión de los datos, debido a ello es que muchas

empresas e institutos han puesto atención en encontrar alternativas en algoritmos que no tengan las vulnerabilidades que presentan los estándares actuales. Para ello han surgido varios proyectos como lo son NESSIE en Europa, el cual recomienda estándares como ISO; CRYPTREC del gobierno Japonés y el más reciente es el eSTREAM de Ecrypt.

Ecrypt (European Network of Excellence for Cryptography) es una organización europea, la cual está compuesta por 32 instituciones y compañías. Esta organización tiene ya cuatro años de consolidación gracias al esfuerzo de la Sociedad de Tecnologías de la información (*IST*) programa de la sexta comisión europea (*FPG*) bajo el número de contrato IST-2002-507932. Ecrypt fue dado a conocer el primero de febrero de 2004[2].

Los objetivos de Ecrypt son:

- Intensificar la colaboración de los investigadores europeos en el ámbito de la seguridad, y en particular en criptografía y marcas de agua digitales.
- Asegurar que sea durable la integración de los investigadores europeos tanto de la academia y la industria, y mantener además de fortalecer la excelencia europea en criptología y marcas de agua.

Ecrypt está compuesto por cinco laboratorios:

- Algoritmos de llaves simétricas (STVL).
- Algoritmos de llave pública (AZTEC).
- Protocolos (PROVILAB).
- Implementaciones seguras y eficientes (VAMPIRE).
- Marcas de agua (WAVILA).

Los laboratorios fueron creados para diseñar mejores algoritmos criptográficos, al igual que protocolos e implementaciones en los siguientes aspectos: veloces, un bajo costo y alta seguridad.

En el laboratorio Vampire se buscan nuevas técnicas para determinar una implementación eficaz y segura, además de brindar el puente entre investigadores y usuarios. En este laboratorio es donde van y han sido evaluados los algoritmos enviados al proyecto eStream.
eStream es el nombre del proyecto de Ecrypt con el que se busca definir nuevos estándares en cifradores de flujo, el cual ha sido conceptualizado desde hace años[2]. La convocatoria de eStream fue abierta en noviembre de 2004, la misma será cerrada y totalmente evaluada para mayo de 2008. Para este proyecto los algoritmos de cifrado de flujo no necesariamente deben funcionar en todas las plataformas y en todas las situaciones. Uno de los objetivos de este proyecto es el de encontrar algoritmos que sean capaces de remplazar al algoritmo AES en modo OFB, AES es un cifrador de bloques pero al ponerlo en su modo de operación OFB (Output Feedback) se comporta como algoritmo de cifrado de flujo, tanto en velocidad de procesamiento en software como el de bajo uso de los recursos de hardware. Otro de los objetivos es encontrar un cifrador de flujo que sea generalizable.

El proyecto eStream se inició con 34 algoritmos. Nueve de los algoritmos han sido diseñados para ser implementados en software: ABC, CryptMT, Dicing, Dragon, Frogbit, Hc-256, Mir-1, py and Sosemanuk; 13 de los algoritmos han sido diseñados para ser implementados en software y hardware: F-Fcsr, Hermes8,Lex, Mag, NLS, Phelix, Polar Bear, Pomaranch, Rabbit, Salsa20, Sss, TRBDK3 Yaea, and Yamb; 12 algoritmos fueron diseñados para ser implementados en hardware: Achterbahn, Decim, Edon 80, **Grain**, **Mickey-128**, Mosquito, Sfinks, **Trivium**, Tsc-3, Vest, wg y Zk-Crypt.

Los rubros a evaluar para los cifradores de flujo con implementación en hardware es:

- Bajo consumo de recursos comparado al Advanced Encryption Standard (*AES*) o al menos el mismo.

- Las consideraciones que se toman en cuenta para evaluar el rendimiento es:
    - Concisión,
    - Ejecución,
    - Consumo de potencia,
    - Flexibilidad / Escalabilidad / Paralelización y
    - Simplicidad / Complitud / Claridad.

El 27 de marzo de 2006 se anunció oficialmente el final de la fase uno de eStream. Iniciándose así el 1ro. de agosto de 2006 la fase dos.

Como parte de los resultados obtenidos de la fase uno se determinó que Grain, Mickey-128 y Trivium conformarían al grupo de los más interesantes a ser implementados en hardware. Lo anterior debido al bajo uso de recursos tales como almacenamiento limitado, número de compuertas y consumo de potencia. Los algoritmos de Grain y Trivium permiten paralelizar el número de bits, lo cuál disminuye el tiempo de procesamiento de la información. Las implementaciones son comparadas en velocidad, flexibilidad y uso de recursos.

1.2 Objetivo general.

Diseñar, implementar y comparar las arquitecturas hardware optimizadas de cada uno de los algoritmos de cifrado de flujo Grain, Mickey-128 y Trivium en velocidad y área.

1.3 Objetivos específicos.

- Diseño e implementación de arquitecturas base para cada uno de los algoritmos.

- Realizar un estudio comparativo de los resultados obtenidos de las implementaciones.
- Definir las condiciones bajo las cuales se adapta mejor cada uno de los algoritmos, esto de acuerdo a las restricciones que presenta cada uno de ellos y de los resultados obtenidos.

1.4 Metodología

- Estudiar cada uno de los cifradores de flujo para entender su funcionamiento y la arquitectura hardware que lo describe.
- Diseñar a bloques las arquitecturas hardware de los cifradores de flujo, obteniendo así las arquitecturas hardware base de cada uno de los cifradores de flujo.
- Validar cada una de las arquitecturas hardware con su respectivo banco de pruebas y estudiar las posibles modificaciones a las arquitecturas.
- Optimizar cada una de las arquitecturas hardware base en área y velocidad.
- Comparación de los resultados obtenidos en cada modificación de cada una de las arquitecturas base.

1.5 Contenido de los capítulos

Para el capítulo 1, se da una reseña de los cifradores de flujo y de su importancia en la seguridad de la información, además de tratar el objetivo general y los objetivos específicos, así como la metodología a seguir para el cumplimiento de los mismos.

En el capítulo dos se abordará el tema de los cifradores de flujo más a fondo y de su interrelación con la seguridad de la información, también se dan las características de tres de los cifradores de flujo de implementación hardware

que pasaron a la tercera etapa del proyecto Ecrypt de eStream, los cuales son: Grain, Mickey-128 y Trivium.

En el capítulo tres se describirán los componentes de Grain, Mickey-128 y Trivium usados para diseñar la arquitectura de cada uno de ellos. Cada componente se diseñó en el lenguaje VHDL en ModelSim y para obtener los resultados de implementación se uso Xilinx ISE.

En el capítulo cuatro se mostrarán los resultados obtenidos de la implementación realizada en esta tesis y se comparan con los resultados obtenidos en trabajos que se han venido realizando hasta este momento.

En el capítulo cinco se encuentran las conclusiones a las que se han llegado con esta investigación y que servirán de apoyo a la selección del cifrador de flujo adecuado.

Capítulo 2

Cifradores de flujo: Grain, Mickey-128 y Trivium

En este capítulo se da una explicación de lo que es la criptografía, de su uso y de su clasificación. Para cada tipo de criptografía se mencionan y describen los algoritmos más usados. Además, de una descripción detallada de tres de los algoritmos de cifradores de flujo, cuya implementación es en hardware, los cuales han pasado a la siguiente fase del proyecto eSTREAM de la organización Ecrypt y son: Grain, Mickey-128 y Trivium. Se mencionan sus características, número de bits en paralelo que se pueden trabajar y la forma en la que se comporta cada uno de estos tres cifradores de flujo.

2.1 Criptografía

La criptografía es la ciencia encargada de mantener la seguridad de la información que es enviada, de manera que sólo le sea posible a las personas que la reciben el poder leer dicha información. Esto se hace cifrando y descifrando la información mediante una técnica matemática.

2.1.1 Historia

La palabra Criptografía viene del griego *kryptos* (ocultar) y *graphos* (escribir), lo cual se traduce como “escritura oculta”. La criptografía fue especialmente usada en tiempo de guerra. Uno de los primeros criptosistemas conocidos es el del historiador griego Polibio, el cual se basaba en el método de sustitución fundado en la posición de las letras de

una tabla. Los romanos usaron un método llamado César, éste era usado por Julio César para enviar mensajes a sus generales. Este método consistía en desplazar las letras un determinado número de posiciones, es decir es un cifrador por sustitución en el que la letra en el texto original es reemplazada por una letra que se encuentra en una posición que está un número determinado de espacios más adelante en el alfabeto. Por ejemplo con un desplazamiento de 3, la A es sustituida por la letra D y la B por la E. Otro método usado por los griegos es el de escítala espartana, método que consiste en la trasposición basada en un cilindro que servía de clave en el que se enrollaba el mensaje para poder cifrar y descifrar.

Leon Battista Alberti inventó un sistema de sustitución polialfabética en el año 1465. En el siglo XVI, el fránces Blaise de Vigenre escribió un tratado sobre "la escritura secreta". Selenus, escribio "Cryptomenytices et Cryptographie", en 1624. En los siglos XVII, XVIII y XIX, los monarcas desarrollaban un gran interés por la criptografía al punto en el que en el reinado de Felipe II utilizaban un alfabeto de 500 símbolos, el cual sólo fue criptoanalizado por el francés François Viète para el rey de Francia. [5]

El holandés Auguste Kerckhoffs y el prusiano Friedrich Kasiski fueron los personajes más importantes en siglo XIX y la segunda guerra mundial. En el siglo XX es cuando surgen las máquinas de cálculo, entre las cuales destaca la máquina alemana Enigma, la cual era una máquina de rotores que automatizaba los cálculos que eran necesarios para realizar las operaciones de cifrado y descifrado de mensajes.

Una vez concluida la segunda guerra mundial, es cuando la criptografía tiene un avance teórico importante; esto se da con las investigaciones sobre teoría de la información realizadas por Claude Shannon. A mediados de los años 70´s el Departamento de normas y estándares norteamericano publica el primer diseño lógico de un cifrador, Estándar de Cifrado de Datos (DES). En el mismo siglo surgieron los sistemas asimétricos, los que permitirían introducir a la criptografía en otros campos el de la firma digital.

2.1.2 Tipos de Criptografía

Los algoritmos criptográficos son usados para cifrar la información que será enviada por algún medio ya sea alambrado o inalámbrico, ya que el cifrar la información permite protegerla de ataques que pudieran alterarla. Se pueden clasificar a los algoritmos criptográficos en tres tipos: criptografía simétrica (clave secreta), criptografía asimétrica (clave pública) y algoritmos Hash (de resumen).

Algoritmos HASH (de resumen)

Los algoritmos HASH, parten de una información de entrada de longitud indeterminada y obtienen como salida un código, que en cierto modo se puede considerar único para cada entrada. La función de estos algoritmos es determinista, es decir que partiendo de una misma entrada siempre se obtiene la misma salida. El interés de estos algoritmos reside en que partiendo de entradas distintas se obtiene salidas distintas.

Un ejemplo son los dígitos de control y los CRC (Cyclic Redundancy Code) que se utilizan para detectar errores de trascripción o de comunicación. Estos algoritmos garantizan que el código generado cambia ante una mínima modificación de la entrada y tienen aplicaciones muy concretas de control de integridad en procesos con perturbaciones fortuitas y poco probables. Son poco robustos y está demostrado que se pueden realizar pequeños conjuntos de modificaciones en un documento de manera que el CRC resulte inalterado. En un buen algoritmo HASH es inadmisible que un conjunto reducido de modificaciones no altere el código resultante, ya que se podrían realizar modificaciones en el documento sin que fuesen detectados, y por lo tanto no se garantiza la integridad.

Dado que el tamaño del código que se genera como salida es de tamaño limitado, (128, 256 ó 512 bits) mientras que el tamaño del documento de entrada es ilimitado, es evidente que se cumplen dos propiedades:

- El algoritmo es irreversible, es decir, no es posible obtener el documento original a partir del código generado.
- Existen varios documentos que dan lugar a un mismo código.

La segunda propiedad es debida a que el número de combinaciones de los códigos de tamaño limitado es menor al número de combinaciones de cualquier archivo grande. Los algoritmos Hash más utilizados son MD5 y SHA1, pero nunca se utilizan códigos CRC en aplicaciones de seguridad. Por ejemplo, los certificados incluyen un campo llamado fingerprint con el resultado de los algoritmos MD5 y SHAI.

MD5 (Message-Digest Algorithm 5) es un algoritmo que produce un código de 128 bits. Fue creado por Ron Rivest en 1991 y se convirtió en el estándar de Internet RFC 1321[3].

El NIST presentó en 1993 un algoritmo basado en las mismas técnicas que MD5 y el denominado SHA (Secure Hash Algorithm). Este algoritmo fue declarado estándar Federal Information Processing Standard PUB 180 en 1993, pero en 1995 la Agencia de Seguridad Nacional (NSA) lo sustituyó por una revisión mejorada que actualmente se conoce como SHA1 y que se considera más seguro que MD5. SHA1 produce un código hash de 160 bits para mensajes de longitud máxima de 264 bits, aunque existen otras variantes poco utilizadas todavía que producen códigos de mayor longitud. En general, SHA1 se considera el mejor algoritmo de esta familia y es el que se aplica en la mayoría de las aplicaciones de firma electrónica. Actualmente, es común aplicar SHAI seguido de RSA para realizar una firma electrónica de un documento, o bien el algoritmo DSA específico para firma electrónica que también utiliza SHA1 internamente.

Digital Signature Algorithm (DSA) es el algoritmo estándar del Gobierno Federal de los Estados Unidos para firma digital. Su desarrollo se atribuye a David W. Kravitz, de la Agencia Nacional de Seguridad. DSA fue presentado al NIST en agosto de 1991, adoptado como estándar en 1993, y con una última revisión en el 2000[4]. El algoritmo DSA es exclusivo de firma

electrónica basado en clave pública (se le llama así a la clave que es conocida tanto por el que envía como por el que recibe el mensaje), pero no vale para comunicaciones confidenciales.

Criptografía asimétrica (clave pública)

Los criptosistemas de este tipo utilizan dos claves distintas para cifrar y para descifrar el mensaje. Ambas claves tienen una relación matemática entre sí, pero la seguridad de esta técnica se basa en que el conocimiento de una de las claves no permite descubrir cuál es la otra clave.

Cada usuario cuenta con una pareja de claves, una se mantiene en secreto y se denomina *clave privada* y otra la distribuye libremente y se denomina *clave pública*. Para enviar un mensaje confidencial sólo hace falta conocer la clave pública del destinatario y cifrar el mensaje utilizando dicha clave. Los algoritmos asimétricos garantizan que el mensaje original sólo puede volver a recuperarse utilizando la clave privada del destinatario. Dado que la clave privada se mantiene en secreto, sólo el destinatario podrá descifrar el mensaje.

Los algoritmos asimétricos pueden trabajar indistintamente con cualquiera de las claves, de manera que un mensaje cifrado con la clave privada sólo puede ser descifrado con la clave pública. Esta característica permite utilizar este método para otras aplicaciones, además de las que sólo requieren confidencialidad, como es el caso de la firma electrónica. En la Fig. 2.1 se muestra el esquema de los algoritmos asimétricos.

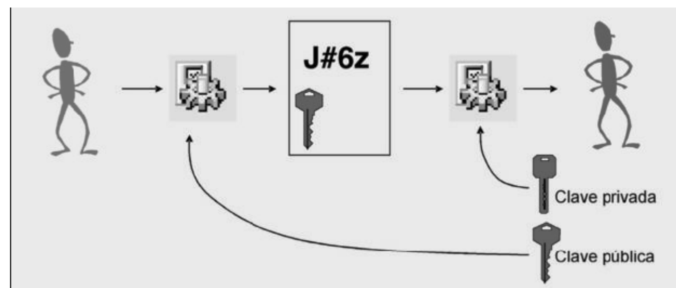

Fig. 2.1 Esquema de los algoritmos asimétricos [5]

Algunas de las características más destacadas de este tipo de algoritmos son las siguientes:

- Se utiliza una pareja de claves denominadas clave pública y clave privada, pero a partir de la clave pública no es posible descubrir la clave privada.
- A partir del mensaje cifrado no se puede obtener el mensaje original, aunque se conozcan todos los detalles del algoritmo criptográfico utilizado y aunque se conozca la clave pública utilizada para cifrarlo.
- Emisor y receptor no requieren establecer ningún acuerdo sobre la clave a utilizar. El emisor se limita a obtener una copia de la clave pública del receptor, lo cual se puede realizar; por cualquier medio de comunicación aunque sea inseguro.

Diffie y Hellman son quienes desarrollaron el primer criptosistema de clave pública en el año de 1976, el cual es conocido como el cambio de clave Diffie-Hellman. Después, fue publicado el primer algoritmo asimétrico denominado RSA, el cual sigue siendo utilizado en la actualidad [4]. RSA fue desarrollado en 1977 por Ron Rivest, Adi Shamir y Len Adleman. El nombre RSA proviene de las iniciales de los apellidos de sus inventores[6]. El algoritmo fue patentado en 1983 por el MIT, el cual expiró desde el 21 de septiembre de 2000 y desde entonces se utiliza libremente. La seguridad de este algoritmo reside en la dificultad que supone la factorización de un

número compuesto por factores primos muy grandes. Si un criptoanalista fuera capaz de encontrar los factores primos sería capaz también de determinar la clave privada y descifrar el mensaje.

Los algoritmos asimétricos se pueden utilizar para cifrado de documentos secretos o para firma electrónica tanto de documentos privados como públicos. Para firmar electrónicamente un documento, el autor lo cifra con su propia clave privada y cualquier usuario puede ver el contenido original descifrando el documento mediante la clave pública correspondiente. El documento es público porque cualquier usuario puede verlo, pero sólo el autor puede modificarlo y volver a cifrarlo ya que él tiene la clave privada. Por lo tanto, no se tiene confidencialidad pero si integridad, y es una herramienta fundamental para generar documentos "oficiales" que sean públicos, pero que vayan debidamente firmados por la autoridad pertinente.

Criptografía simétrica (clave secreta).

En este tipo de sistema tanto el emisor como el receptor cifran y descifran la información con una misma clave (clave secreta), la cual comparten. El proceso consiste en que el emisor cifra el mensaje con la clave k y es enviado al receptor, el cual descifra el mensaje con la clave k. La clave debe ser de al menos 40 bits. Este tipo de sistema de cifrado tiene la ventaja de ser eficiente, ya que los algoritmos utilizados son muy rápidos al poder implementarse tanto en hardware como en software. En la Fig. 2.2 se muestra un esquema de los algoritmos simétricos.

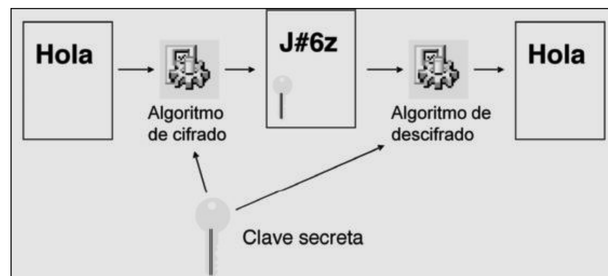

Fig. 2.2 Esquema de los algoritmos simétricos [5]

Algunas de las características de este tipo de algoritmos son:

- A partir del mensaje cifrado no se puede obtener el mensaje original ni la clave que se ha utilizado.
- Se utiliza la misma clave para cifrar el mensaje original que para descifrar el mensaje codificado.
- El emisor y receptor deben haber acordado una clave común por medio de un canal de comunicación confidencial antes de poder intercambiar información confidencial por un canal de comunicación inseguro.

Entre los algoritmos simétricos más conocidos se encuentra DES (Data Encryption Standard), 3DES, RC2 (Rivest´s Cipher), RC4, RC5, IDEA (International Data Encription Algorithm), Blowfish y AES (Advanced Encryption Standard).

DES se basó en Lucifer desarrollado por IBM (1975) y utilizado por las oficinas gubernamentales desde 1977 y denominado como estándar por NIST (National Institute of Standards and Technology). DES utiliza claves de cifrado de 56 de bits de los cuales usa 48 bits, actualmente se le considera poco robusto sobretodo desde que en 1998 la Electrónica Frontier Foundation hizo público un descifrador de código DES con el que se puede obtener el mensaje original en menos de 3 días.[4]

El algoritmo 3DES fue desarrollado por Tuchman en 1978, el cual consiste en aplicar el algoritmo DES tres veces consecutivas. Se puede aplicar con la misma clave cada vez, o con claves distintas y combinando el algoritmo de cifrado con el de descifrado, lo cual da lugar a DES-EEE3, DES-EDE3, DES-EEE2 y DES-EDE2. El resultado es un algoritmo seguro y que es utilizado en la actualidad [7].

En 1989, Ron Rivest desarrolló el algoritmo RC2 (Rivest’s Cipher) para RSA Data Security,Inc. RC2 es un algoritmo de cifrado por bloques, que utiliza

una clave de tamaño variable, este algoritmo es de dos a tres veces más rápido que el algoritmo DES, siendo más seguro[7].

Ron Rivest desarrolló el algoritmo RC4 en 1987 para RSA Data Security, que se hizo público en 1994. Se considera inmune al criptoanálisis diferencial y lineal. Es el algoritmo utilizado en el cifrado WEP de la mayoría de los puntos de acceso WiFi.

Otro algoritmo de Ron Rivest es RC5, publicado en 1994. Se trata de un algoritmo de cifrado por bloques, que utiliza claves de tamaño variable. Se caracteriza por la sencillez del algoritmo, que lo hacen muy rápido y fácil de implementar tanto en software como en hardware.

El International Data Encription Algorithm, IDEA, fue diseñado por Xuejia Lai y James L. Massey de ETH-Zurich. IDEA es un algoritmo de cifrado por bloques de 64 bits que emplea claves de128 bits, que se presentó por primera vez en 1991. IDEA es dos veces más rápido que DES, a pesar de utilizar claves mucho más largas[5].

Blowfish es un algoritmo de cifrado por bloques de 64 bits diseñado por Bruce Schneier en 1993. Blowfish utiliza claves de longitud variable entre 32 y 448 bits. Es considerado como un algoritmo seguro y es más rápido que DES. Twofish es una variante de Blowfish, utiliza bloques de 128 bits y claves de 256 bits, fue diseñado por Bruce Schneier, en colaboración con John Kelsey, Doug Whiting, David Wagner, Chris May y Niels Ferguson. Twofish fue uno de los cinco finalistas para sustituir a DES como algoritmo estándar en la selección de NIST[5].

Advanced Encryption Standard, AES, es el estándar para cifrado simétrico del NIST desde el 26 de mayo de 2002 en sustitución de DES. AES fue desarrollado por dos criptógrafos Belgas, Joan Daemen y Vincent Rijmen, es un algoritmo de cifrado por bloques con longitud de bloque y longitud de clave variables. Los valores adoptados para el estándar son bloques de 128 bits y claves de longitud de 128, 192 ó 256 bits.[8]

Actualmente la organización Ecrypt tiene un proyecto llamado eStream, con el cual quieren seleccionar algoritmos de cifrado de flujo, de implementación en software y/o hardware, para que sean candidatos a ser los nuevos estándares de cifrado, y en específico, para implementación móvil, de ahí radica el porqué cifradores de flujo.

2.1.3 Especificaciones de los cifradores de flujo.

En los cifradores de simétricos encontramos a los cifradores de flujo y los cifradores de bloques. Los cifradores de bloque son algoritmos que van cifrado el texto plano por bloques de bits mediante una llave; mientras que los cifradores de flujo van cifrado bit a bit usando el llamado keystream generado por el algoritmo.

Los cifradores de flujo no sufren de errores de propagación, esto debido a que cada bit es cifrado o descifrado independientemente de los otros, por lo que son más rápidos que los cifradores de bloque. Debido a lo anterior se usan para aplicaciones de tiempo crítico o procesos que tienen recursos limitados.

La Fig.2.3 muestra el esquema general de cifrado. Cuando se envía un mensaje (texto plano), éste es cifrado por un algoritmo que usa una llave de la cual tiene conocimiento tanto la persona que envía el mensaje como la que lo recibe; una vez que ha llegado el mensaje cifrado a su destino éste es descifrado con ayuda de la llave y el algoritmo que se usó en un principio para cifrar así como el algoritmo que se uso para cifrar dando como resultado el mismo texto plano que se tenía al inicio.

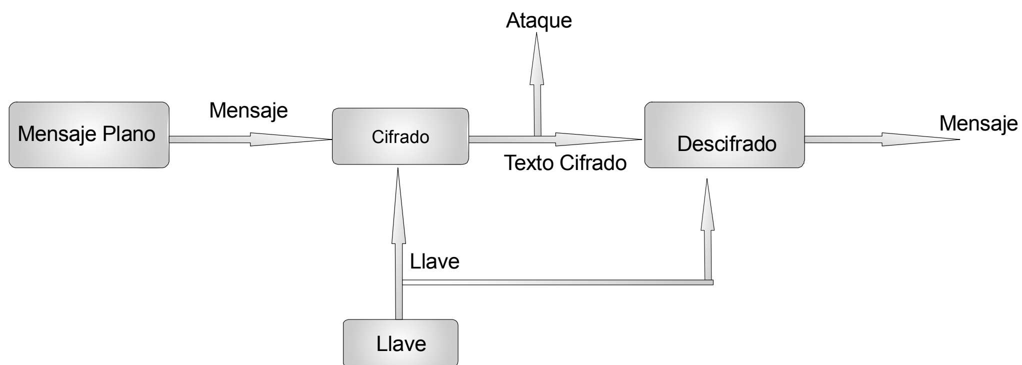

Fig. 2.3 Esquema de un mensaje en el envío y recepción [9].

Existen un gran número de cifradores de flujo que han sido tomados como estándares y usados en protocolos, algunos de ellos son A5/1 en GSM, E0 en Bluetooth, RC4 en IEEE.

Los cifradores de flujo son usados en muchas aplicaciones, algunas de ellas son: cifrado de voz y video en comunicaciones, transmisión de voz por IP, transmisión de video digital, comunicaciones móviles que requieran de alta velocidad de transmisión, área mínima y consumo mínimo de potencia.

RC4 es un algoritmo estándar usado en redes inalámbricas IEEE 802.11 que sirve para proteger los datos durante la transmisión inalámbrica, el protocolo de seguridad se llama Wired Equivalent Privacy (WEP). Otro protocolo que usa a RC4 y que sirve para proteger el tráfico de internet es Secure Sockets Layer (SSL).

A5/1, A5/2, A5/3 son estándares usados para comunicación celular en Global System for Mobile Communications (GSM) y permiten la privacidad de la información en el aire.

E0 es el estándar usado en Bluetooth, E0 permite la comunicación vía radio a corta distancia con bajo consumo de potencia. Una de las aplicaciones que tiene este protocolo es la transferencia de archivos entre celulares, computadoras y PDAs vía inalámbrica.

2.1.4 Cifradores de flujo candidatos a estándares en hardware.

La organización Ecrypt, es una organización europea constituida por 32 instituciones y universidades, tiene un proyecto llamado eSTREAM cuya convocatoria se abrió en el año 2004 con el objetivo de elegir nuevos algoritmos para que sean los nuevos estándares a usar en los protocolos y aplicaciones debido a la sencillez de su implementación pero que al mismo tiempo sea seguros criptoanalíticamente. Estos algoritmos deben de consumir baja potencia, bajo uso de recursos, ser veloces y sencillos.

Las categorías se dividieron en dos: cifradores de flujo a ser implementados en software y a ser implementados en hardware, algunos de los cifradores de flujo que fueron aceptados en la convocatoria pueden entrar en ambas categorías.

Tres de los cifradores de flujo a ser implementados en hardware que han pasado a la tercera etapa son : Grain, Mickey-128 y Trivium. A continuación se explican éstos con mayor detalle.

2.2 Grain.

Grain es un cifrador de flujo de implementación hardware, éste fue diseñado por Martín Hell, Thomas Johansson y Milli Meier. Grain tiene como objetivo cumplir con las siguientes primitivas criptográficas: velocidad, seguridad y simplicidad, además de cumplir con las características de usar un bajo consumo de recursos y la facilidad de implementación. Éste cifrador de flujo es síncrono y genera el keystream independientemente del texto plano.

2.2.1 Algoritmo Grain.

Grain usa dos parámetros de entrada: *key* (llave) de 80 bits e *IV* (vector de inicialización) de 64 bits. El algoritmo usa dos registros de 80 bits

cada uno, los cuales son *LFSR* (linear feedback shift register) y *NFSR* (non-linear feedback shift register), y tres funciones, de las cuales dos son de retroalimentación y una de generación del bit para el keystream. El esquema de Grain es el que se presenta en la Fig. 2.4.

Para obtener el keystream, el cual tiene como máximo 2^{64}, se siguen dos etapas: una de inicialización y otra de generación del keystream.

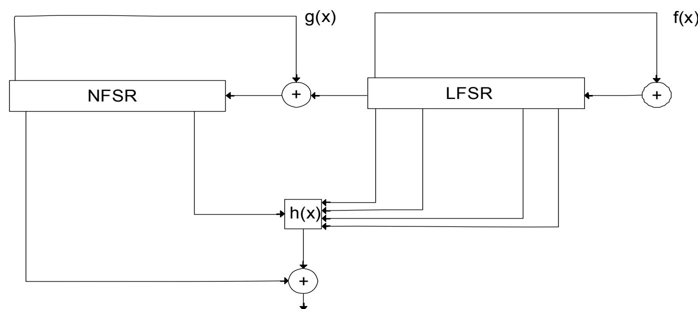

Fig. 2.4 Esquema general de Grain

Etapa de inicialización.

Para esta etapa se usan: *key* e *IV*, los cuales son usados para inicializar los registros *NFSR* y *LFSR* respectivamente. Cada uno de estos registros van de 0 a 79, por lo que el registro *LFSR* es rellenado por 1´s desde la posición 64 a la 79.

Para estabilizar estos registros, la función *h(x)* hace un XOR con determinados bits del registro *NFSR*, los cuales son usados para otra función de retroalimentación, cuyo bit de resultado entra como otro parámetro a las funciones *g(x)* y *f(x)* durante 160 ciclos, vea Fig. 2.5.

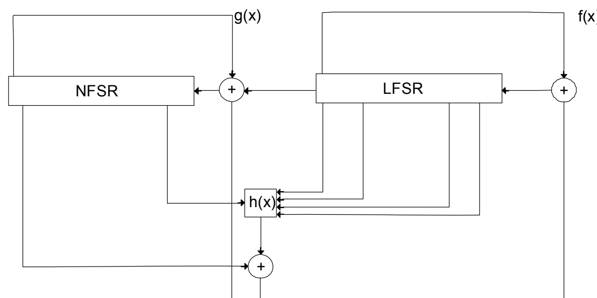

Fig. 2.5 Grain en etapa de inicialización.

Las funciones *g(x)*, *f(x)* y *h(x)* están definidas de la siguiente manera:

$$g(x) = s_i + b_{i+62} + b_{i+60} + b_{i+52} + b_{i+45} + b_{i+37} + b_{i+33} + b_{i+28} + b_{i+21} + b_{i+14} + b_{i+9} + b_i +$$

$$b_{i+63}b_{i+60} + b_{i+37}b_{i+33} + b_{i+15}b_{i+9} + b_{i+60}b_{i+52}b_{i+45} + b_{i+33}b_{i+28}b_{i+21} + b_{i+63}b_{i+45}b_{i+28}b_{i+9} +$$

$$b_{i+60}b_{i+52}b_{i+37}b_{i+33} + b_{i+63}b_{i+60}b_{i+21}b_{i+15} + b_{i+63}b_{i+60}b_{i+52}b_{i+45}b_{i+37} + b_{i+33}b_{i+28}b_{i+21}b_{i+15}b_{i+9} +$$

$$b_{i+52}b_{i+45}b_{i+37}b_{i+33}b_{i+28}b_{i+21}$$

$$f(x) = s_{i+62} + s_{i+51} + s_{i+38} + s_{i+23} + s_{i+13} + s_i.$$

$$h(x) = s_{i+3} + s_{i+25} + s_{i+46} + s_{i+64} + b_{i+63}.$$

$$z_i = keystream = h(X) + \left(b_{i+1} + b_{i+2} + b_{i+4} + b_{i+10} + b_{i+31} + b_{i+43} + b_{i+56}\right).$$

Los subíndices indican la posición del registro correspondiente, donde las b's son del registro *NFSR* y las s's del registro *LFSR*.

El bit que resulta de la función *g(x)* y *f(x)* es colocado en la posición 79 de su respectivo registro, lo que implica corrimientos a la izquierda sin retroalimentación.

Etapa de generación de keystream.

En esta etapa ya no se usa el resultado de la función XOR de *h(x)* con determinados bits del registro *NFSR* como retroalimentación a los registros, sino que ya es enviado como una salida; tal como se muestra en la Fig. 2.4.

2.2.2 Resultados de trabajos realizados sobre Grain.

En esta sección se muestran los resultados reportados en la literatura con respecto a estos tres algoritmos. Las métricas utilizadas son la velocidad de procesamiento (Mbps), frecuencia (MHz), tiempo (ns), área (slices) y velocidad de procesamiento por área (Mbps/slices).

En el trabajo [10] se implementó una arquitectura en la tarjeta XC2S15-5 Spartan2 con la herramienta Xilinx, la simulación se realizó usando un bit y bits en paralelo. Para un bit los resultados de frecuencia fueron de 105 MHz con 48 slices una velocidad de procesamiento de 105 Mbps; mientras que, para 16 bits en paralelo la frecuencia es de 105 MHz y la velocidad de procesamiento es de 1680 Mbps.

También en [11] se implementó el algoritmo de Grain con un grado de paralelización de 16 bits por ciclo, concluyendo que Grain, Mickey y Trivium son los que requieren menos área en comparación a Achterbahn, Mosquito, Sfinks, Vest y ZK-Crypt que son los otros algoritmos que analizaron. Obteniendo también que, los algoritmos Grain, Trivium, Vest, ZK-crypt tienen una alta velocidad de procesamiento con respecto al algoritmo de referencia AES. Concluyendo con respecto a la tarjeta FPGA usada, para la cual no se mencionan sus especificaciones, que: Grain, Trivium y ZK-Crypt resultan ser 20 veces más eficientes que AES. Los datos obtenidos para el cifrador de flujo Grain con 16 bits en paralelo fueron una frecuencia de 300 MHz una velocidad de procesamiento de 4475 Mbps.

En [12] se implementó el algoritmo Grain en una tarjeta Xilinx Spartan3 utilizando la XC3S50PQ208-5 y el sintetizador Synplicity Synplify Pro para Xilinx. Se efectuaron diferentes grados de paralelización que van desde el caso base hasta el máximo grado de paralelización (16 bits), ver tabla 2.1.

Grado de paralelización	Frecuencia máx. de reloj	Tiempo de setup para la llave		Vel. de proc. máx.		area		Eficiencia	
	MHz	ciclos	ns	Mbps	X básico	CLB slices	X básico	Mbps/ CLB slices	X básico
1 (básico)	193	304	1575	193	1.0	122	1.0	1.58	1.0
2	168	152	905	336	1.7	147	1.2	2.29	1.4
4	170	76	447	680	3.5	173	1.4	3.93	2.5
8	161	38	236	1288	6.72	244	2.0	5.28	3.3
16	155	19	123	2480	12.8	356	2.9	6.97	4.4

Tabla 2.1 Resultados para diferentes grados de paralelización en el dispositivo XC3S50PQ208-5 [12]

En la tabla 2.2 se hace una comparación de los resultados de [10], [11] y [12], con respecto a la frecuencia, la velocidad de procesamiento, el área y la eficiencia que se mide como velocidad de procesamiento/área. En ésta tabla se puede notar que [11] es el trabajo que mejores resultados ha reportado con respecto a la frecuencia y la velocidad de procesamiento, pero no es posible determinar en qué dispositivo se realizaron las pruebas ya que no se encuentra disponible ese dato (n.d).

Trabajo	Frecuencia MHz	Vel. de Proc. Mbps	Area CLB slices	Eficiencia Mbps/slices
[10]	105	1680	n.d	n.d
[11]	300	4475	n.d	37.346
[12]	155	2480	356	6.97

Tabla 2.2 Comparación de los resultados para Grain con 16 bits en paralelo

2.3 Mickey-128.

Mickey-128 fue diseñado por Steve Babbage y Matthew Dodd. El propósito del diseño de Mickey-128 “es tener una baja complejidad en hardware, mientras se ofrece un alto grado de seguridad”[13]. Mickey-128 “usa ciclos de reloj irregulares para sus registros”[13]; además requiere de dos entradas para funcionar, las cuales son *key* de 128 bits y el *IV* de entre 0 y 128 bits. El texto cifrado se obtiene de implementar la función XOR del texto plano con el keystream. Un esquema general de Mickey-128 es el que se muestra en la Fig.2.6.

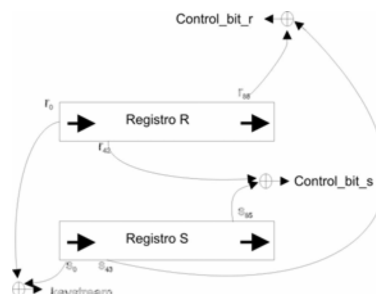

Fig.2.6 Esquema general de Mickey-128

2.3.1 Algoritmo Mickey-128

Mickey-128 está compuesto por dos registros *R* (registro lineal) y *S* (registro no-lineal) de 160 bits cada uno, cuenta con dos señales de control *Mixing* e *Input_bit*, las cuales van a modificar junto con posiciones de bits específicos de los registros a las señales de decisión de los procesos que modificarán a los registros. Los parámetros de entrada son *key* de 128 bits y una variable de inicialización (*IV*) que es de longitud variable que va de 0 a 128 bits.

Etapa de inicialización.

En esta etapa los registros son inicializados con ceros, la señal *Mixing* siempre va a tomar el valor de true, y el valor de *Input_bit* va a ir tomando el valor del bit en la posición correspondiente del *IV* y de *key*. Es decir, que los siguientes ciclos for se van a ejecutar, haciendo el llamado del procedimiento CLOCK_KG con los valores correspondientes. En el primer ciclo de llamados se usa el bit de la posición *i* que corresponda en el *IV*; en el segundo ciclo de 127, usando el bit que corresponden a la posición *i* de la llave.

For $0 \leq i \leq$ IVLENGTH -1 :

CLOCK_KG (R , S , Mixing =TRUE ,Input_bit =IV_i)

For $0 \leq i \leq 127$:

CLOCK_KG (R , S , Mixing =TRUE ,Input_bit =key_i)

Después viene una etapa de estabilización de registros mediante 160 ciclos, en la cual el INPUT_BIT es definido como cero.

For $0 \leq i \leq 159$:

CLOCK_KG (R , S , Mixing =TRUE , Input_bit =0)

Etapa de generación de keystream.

Se pueden generar hasta 2^{64} bits de keystream, y se deben seguir los siguientes pasos:

$Z_i = r_0$ xor s_0

CLOCK_KG (R , S , Mixing = FALSE , Input_bit = 0)

Procedimientos.

CLOCK_KG.

Es el procedimiento principal de control, es decir, es donde se toma la decisión de lo que debe realizarse con los datos y la manera en la cual se trabajarán los registros, los parámetros de entrada son los datos que halla en el registro *R*, *S*, el valor correspondiente de *Mixing* e *Input_bit*. En el procedimiento CLOCK_KG se definen los parámetros de entrada de los procedimientos que son llamados, los parámetros que se definen son: el valor del bit de las señales de control de *CLOCK_R* y *CLOCK_S*, las señales de control *Input_bit_r* e *Input_bit_s*.

CLOCK_KG (R ,S ,Mixing, Input_bit)

Control_bit_r = s_{54} xor r_{106}

Control_bit_s = s_{106} xor r_{53}

If Mixing =TRUE ,

CLOCK_R (R , Input_bit_r = Input_bit XOR s_{80} ,

Control_bit_r =Control_bit)

CLOCK_S(S , Input_bit_s = Input_bit,

Control_bit_s =Control_bit)

ELSEIF Mixing = FALSE ,

CLOCK_R (R , Input_bit_r = Input_bit,

Control_bit_r =Control_bit)

CLOCK_S (S , Input_bit_s = Input_bit,

Control_bit_s =Control_bit)

CLOCK_R

El procedimiento está definido para operar los elementos del registro *R* de acuerdo a los parámetros de entrada que se tengan. En RTAPS son definidas ciertas posiciones que serán manipuladas mediante la función XOR con el valor que se define para *FEEDBACK_BIT*. Por la manera en la que son manipulados los bits del registro resulta imposible adelantarse al valor que contendrá determinada posición en el siguiente ciclo.

RTAPS ={0, 4, 5, 8, 10, 11, 14, 16, 20, 25, 30, 32, 35, 36, 38, 42, 43, 46, 50, 51, 53, 54, 55, 56, 57, 60, 61, 62, 63, 65, 66, 69, 73, 74, 76, 79, 80, 81, 82, 85, 86, 90, 91, 92, 95, 97, 100, 101, 105, 106, 107, 108, 109, 111, 112, 113, 115, 116, 117, 127, 128, 129, 130, 131, 133, 135, 136, 137, 140, 142, 145, 148, 150, 152, 153, 154, 156, 157 }

$r_0 \ldots r_{159}$ es el registro antes del procedimiento
$r'0 \ldots r'_{159}$ es el registro después del procedimiento

CLOCK_R (R , Input_bit_r , Control_bit_r)

FEEDBACK *BIT* $= r_{159}$ *XOR* Input_bit_r

For $1 \leq i \leq 159$, $r'_i = r_{i-1}$ $r'_0=0$

For $0 \leq i \leq 159$, if $i \subseteq$ RTAPS , $r'_i = r_i$ XOR FEEDBACK_BIT

If Control_bit_r= 1 :

For $0 \leq i \leq 159$, $r'_i = r'_i$ XOR r_i

En la Fig.2.7 se muestra cómo se modifica el registro *R* cuando en el procedimiento CLOCK_R la señal *Control_bit* es igual a cero, y en la Fig. 2.8 como se realiza la modificación del mismo registro con el mismo procedimiento cuando la señal de *Control_bit* es igual a uno.

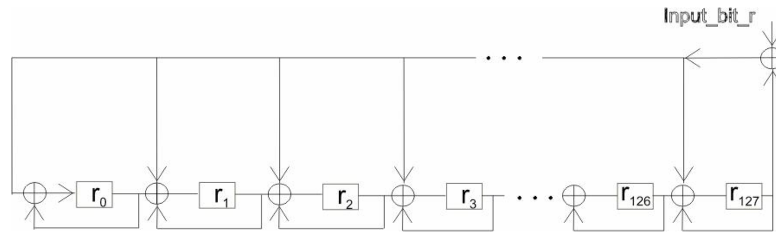

Fig. 2.7 Control_bit=0

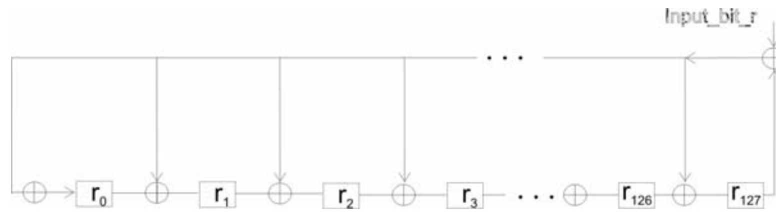

Fig.2.8 Control_bit=1

CLOCK_S

Para este procedimiento se definen cuatro secuencias $COMP0_i$, $COMP1_i$, $FB0_i$ y $FB1_i$ de la siguiente manera:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
COMP0i		1	1	1	1	0	1	0	0	1	0	0	1	1	1
COMP1i		0	0	0	1	1	0	0	1	1	1	1	1	0	0
FB0i	1	1	1	1	0	1	0	1	1	1	1	1	1	0	0
FB1i	1	1	0	1	0	1	0	1	1	1	1	0	1	1	1
i	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
COMP0i	1	0	1	1	0	1	0	1	1	1	0	1	1	1	0
COMP1i	0	1	0	0	1	1	0	0	0	1	0	1	1	1	1
FB0i	0	0	0	1	1	1	1	0	0	0	0	1	0	0	0
FB1i	0	0	0	1	0	1	1	1	1	1	1	0	1	1	0
i	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
COMP0i	1	0	1	0	1	0	1	0	1	0	1	0	0	1	0
COMP1i	1	0	0	0	0	1	1	0	0	1	0	0	1	1	1
FB0i	1	1	0	1	0	0	0	1	0	0	1	1	0	0	0
FB1i	0	1	0	0	0	0	1	0	0	1	0	0	1	1	0
i	45	46	47	48	49	50	51	52	53	108	109	110	111	112	113

COMP0i	0	0	0	0	1	1	0	0	1	1	0	1	1	1	1
COMP1i	1	0	0	0	1	1	0	1	1	0	1	1	1	0	0
FB0i	1	0	1	1	1	1	1	0	1	0	0	1	0	0	0
FB1i	0	0	1	1	0	0	1	1	1	1	1	1	0	1	1
i	117	118	122	123	124	125	126	127	128	129	130	131	132	133	134
COMP0i	1	0	1	0	1	1	0	0	0	1	1	1	1	1	0
COMP1i	1	1	1	0	0	1	1	0	1	0	1	0	1	1	0
FB0i	0	1	1	0	1	0	1	1	1	0	0	0	0	0	1
FB1i	0	1	1	1	0	1	1	0	1	1	1	1	0	1	1
i	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149
COMP0i	1	0	1	1	0	0	0	0	0	0	1	1	1	1	1
COMP1i	1	1	1	0	1	1	0	1	0	0	0	1	0	1	1
FB0i	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1
FB1i	1	0	0	0	0	0	0	0	1	1	1	1	0	0	1
i	150	151	152	153	154	155	156	157	158	159					
COMP0i	0	1	1	1	1	1	0	0	0						
COMP1i	1	1	1	1	1	1	1	1	1						
FB0i	1	0	1	1	0	0	0	0	0	1					
FB1i	0	1	1	0	0	0	1	0	0	0					

Los elementos de estas secuencias son utilizados en el procedimiento CLOCK_S, el cual es usado para manipular el registro *S*. Los nuevos bits que definirán el nuevo estado del registro *S* son calculados mediante funciones XOR y AND de entre valores específicos de determinadas posiciones definidas por el algoritmo del procedimiento.

$s_0 ... s_{159}$ es el estado del registro S antes del procedimiento.

$\hat{s}_0 … \hat{s}_{159}$ es un estado intermedio.

$S'_0 ... s'_{159}$ es el estado del registro S después del procedimiento.

CLOCK_S (S , Input_bit_s , Control_bit_s)

FEEDBACK *BIT* $= s_{159}$ *XOR* Input_bit_s

For $1 \leq i \leq 158$, $\hat{s}(( ) ( ))_i = s_{i-1}$ XOR ((s_i XOR $COMP0_i$)*(s_{i+1} XOR $COMP1_i$)) $\hat{s}_0=0$; $\hat{s}_{159}=s_{158}$

If Control_bit_s = 0 :

For $0 \leq i \leq 159$, $s'_i = \hat{s}_i$ XOR ((FB0$_i$* FEEDBACK_BIT)

If Control_bit_s = 1 :

For $0 \leq i \leq 159$, $s'_i = \hat{s}_i$ XOR (FB1$_i$* FEEDBACK_BIT)

En la Fig. 2.9 se muestra lo que ocurre en el registro *S*, en dependencia del valor que tenga la señal *Control_bit_s*. Cuando la señal *Control_bit_s* toma el valor cero, los valores que se toman en cuenta para definir el nuevo valor final en el actual llamado son los que están definidos en FB0; cuando la señal *Control_bit_s* toma el valor de uno, los valores que se toman en cuenta son los definidos por FB1.

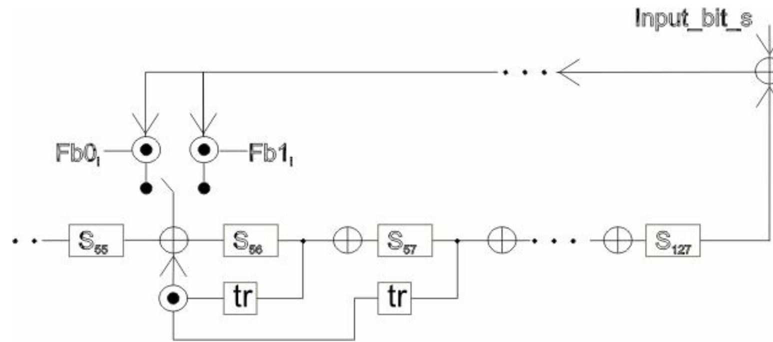

Fig. 2.9 Registro S

2.3.2 Resultados de trabajos realizados sobre Mickey-128.

En el trabajo [1] se describieron los algoritmos usando el lenguaje VHDL mediante una descripción lógica. Los resultados de síntesis de la implementación del cifrador de flujo en la tarjeta Virtex-E V400EFG676 son: 167 CLBs, frecuencia de 166 MHz, velocidad de procesamiento de 166 Mbps y una eficiencia de 0.99 Mbps /área .

En el trabajo [11] se reportan los resultados de la etapa de síntesis dando una frecuencia de 308 MHz, una velocidad de procesamiento de 287 Mbps, concluyendo que el algoritmo Mickey-128 es ligeramente mejor que el algoritmo referencia de AES.

Los resultados de la tabla 2.3 son reportados en [14], estos resultados son obtenidos de la etapa de síntesis en una tarjeta FPGA XCV50ECS144, los cuales son los resultados de síntesis y del análisis de la eficiencia.

Recursos	Usado	Disponible	Utilizado
Recurso	XCV50ECS144		
I/Os	6	94	5%
FGs	333	1536	21.6%
Clb slices	167	768	21.7%
DFFs	235	1818	12.9%
Frecuencia (MHz)	170		
Vel. de proc. (Mbps)	170		

Tabla 2.3 Resultados de síntesis del dispositivo XCV50ECS144 [14]

En la tabla 2.4 se muestra la frecuencia, el throughput, área y la eficiencia con respecto a los recursos usados por cada referencia. Por la forma en la que fue diseñado el algoritmo de Mickey-128 no es posible determinar el valor que contendrá cualquier posición de los registros en un estado posterior, ya que en cada ciclo son tomados y modificados todos los bits de ambos registros. El máximo grado de paralelización que se puede tener con Mickey-128 es un bit por ciclo de reloj.

Trabajo	Frecuencia MHz	Vel. de Proc. Mbps	Area CLB slices	Eficiencia Mbps/CLB slices
[1]	166	166	167	0.99
[11]	308	287	n.d	n.d
[14]	170	170	167	1.018

Tabla 2.4 Comparación de los resultados para Mickey-128 con 1 bit

2.4 Trivium.

Trivium es un algoritmo síncrono diseñado por Christoph De Cannière y Bart Preneel para ser flexible sin dejar de ser compacto en ambientes

restringidos, con eficiencia en potencia y de gran velocidad en la encriptación.

Trivium está diseñado para generar un keystream de hasta 2^{64} bits, usando como entradas una llave (*key*) de 80 bits y un vector de inicialización (*IV*) de 80 bits.

Este algoritmo está compuesto por un registro de 288 bits los cuales van de la posición 0 a la 287, la Fig. 2.10 muestra un esquema del cifrador de flujo Trivium.

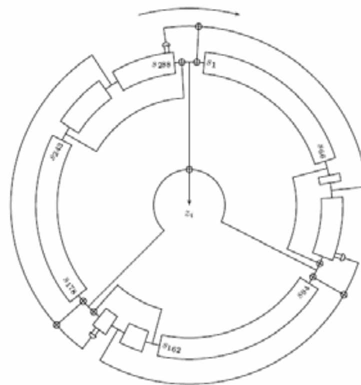

Fig. 2.10 Esquema de Trivium

2.4.1 Algoritmo Trivium.

Este cifrador de flujo usa un registro de 288 bits, su funcionamiento se divide en dos etapas. La primera etapa es la de inicialización y la segunda etapa es de generación del keystream. Una vez que se ha modificado un bit éste no vuelve ha ser usado hasta después de 64 iteraciones, por lo que se puede tener una paralelización de hasta 64 bits.

Etapa de inicialización.

En esta etapa el algoritmo hace uso del *Key* (llave) y del *IV* (vector de inicialización), los cuales son colocados en determinadas posiciones del registro, de la posición 1 a la 80 la llave y de la 94 a la 173 el vector de

inicialización, poniendo a cero el resto de las posiciones ha excepción de las tres ultimas que se ponen a uno, para poder inicializar el registro.

El siguiente es el pseudo-código de esta etapa, en la cual se contempla un ciclo de 4*288 para terminar de inicializar al registro. Se generan nuevos valores *t*, los cuales son usados en las posiciones 1, 94 y 178 ; se realizan corrimientos a la derecha sin retroalimentación.

$(s_1, s_2, ..., s_{93})$ $(K_1,...,K_{80}, 0, ..., 0)$

$(s_{94}, s_{95},..., s_{177})$ $(IV_1,..., IV_{80}, 0,..., 0)$

$(s_{178}, s_{279}, ..., s_{288})$ $(0,..., 0, 1, 1, 1)$

for *i* = 1 to 4 * 288 do

$t_1 \leftarrow s_{66} + s_{91} * s_{92} + s_{93} + s_{171}$

$t_2 \leftarrow s_{162} + s_{175} * s_{176} + s_{177} + s_{264}$

$t_3 \leftarrow s_{243} + s_{286} * s_{287} + s_{288} + s_{69}$

$(s_1, s_2, ..., s_{93}) \leftarrow (t_3, s_1,..., s92)$

$(s_{94}, s_{95}, ..., s_{177}) \leftarrow (t_1, s_{94},..., s_{176})$

$(s_{178}, s_{279}, ..., s_{288}) \leftarrow (t_2, s_{178},..., s_{287})$

end for

Etapa de generación del Keystream

En la etapa de generación del *keystream* se realiza un proceso iterativo en el cual se usan 15 bits específicos para poder generar los 3 bits correspondientes al estado en el que se encuentra, estos bits son usados para generar un bit del keystream z_i usando la función XOR. Los bits son rotados, y se continua el proceso hasta ejecutarlo $N \leq 2^{64}$ de veces. El siguiente es el pseudo-código que describe el proceso de generación del keystream.

for *i* = 1 to N do

$t_1 \leftarrow s_{66} + s_{93}$

$t_2 \leftarrow s_{162} + s_{177}$

$t_3 \leftarrow s_{243} + s_{288}$

$z_i \leftarrow t_1 + t_2 + t_3$

$t_1 \leftarrow t_1 + s_{91} * s_{92} + s_{171}$

$t_2 \leftarrow t_2 + s_{175} * s_{176} + s_{264}$

$t_3 \leftarrow t_3 + s_{286} * s_{287} + s_{69}$

$(s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$

$(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$

$(s_{178}, s_{279}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$

end for

Este pseudo-código es casi igual al que se describe en la etapa de inicialización, con la excepción de que aquí ya se determina el bit del keystream. El keystream es determinado por posiciones específicas del registro, las cuales son consideradas para determinar el valor de los bits que se colocarán en el registro.

2.4.2 Resultados de trabajos realizados sobre Trivium.

En [10] se realizó una implementación del algoritmo Trivium usando una máquina de estados para indicar las señales de control necesarias y usando señales que trabajen en paralelo. Para un bit se usó la tarjeta XC2S15-5 obteniendo una frecuencia de 102 MHz, y una velocidad de procesamiento de 102 Mbps. Para la implementación de 64 bits en paralelo se usó la tarjeta Spartan2 dando una frecuencia de 102 MHz, y una velocidad de procesamiento de 6528 Mbps.

En [8] se implementó en la tarjeta Virtex-E V400EFG676 usando 144 CLBs, una frecuencia de 211MHz y una velocidad de procesamiento de 211Mbps. La implementación del algoritmo se realizó usando una descripción lógica estructural usando el lenguaje VHDL.

En [11], se realizó una selección de los algoritmos de cifrado de flujo entre los cuales se encuentra Trivium. Trivium fue implementado en distintas tarjetas FPGA ocupando poca área en conjunto con Grain y Mickey. Los

resultados que se obtuvieron para 64 bits en paralelo en el trabajo fueron una frecuencia de 312 MHz y una velocidad de procesamiento de 18,568Mbps.

Los resultados de implementación que se muestran en la tabla 2.5 son del trabajo [12], los cuales se obtuvieron de la implementación del algoritmo en una tarjeta Xilinx Spartan3 XC3S400FG320-5.

Grado de paralelización	Frecuencia máx. de reloj	Tiempo de setup para la llave		Vel. de proc. máx.		Area		Eficiencia	
	MHz	ciclos	ns	Mbps	X básico	CLB slices	X básico	Mbps/ CLB slices	X básico
1 (básico)	201	1312	6527	201	1.0	1888	1.00	1.07	1.00
2	202	656	6248	404	2.0	189	1.01	2.14	2.00
4	203	328	1616	812	4.0	199	1.06	4.08	3.82
8	193	164	850	154	7.7	199	1.06	7.76	7.26
16	191	82	429	3056	15.2	227	1.21	13.46	12.59
32	202	41	203	6464	32.2	264	1.40	24.48	22.90
64	190	21	108	12160	60.5	388	2.06	31.34	29.31

Tabla 2.5 Resultados de implementación del cifrador de flujo Trivium en el dispositivo XC3s400FG320-5[12]

En la tabla 2.6 se comparan los resultados de los trabajos [9], [14], [11] y [12] con respecto a la frecuencia, velocidad de procesamiento, área y eficiencia con 64 bits en paralelo.

Trabajo	Frecuencia MHz	Vel. de proc. Mbps	Area CLB slices	Eficiencia Mbps/CLB slices
[9]	102	6528	n.d	n.d
[1]	211	211	144	1.465
[11]	312	18568	n.d	n.d
[12]	190	12160	388	31.34

Tabla 2.6 Comparación de los resultados para Trivium con 64 bits en paralelo.

2.5 Resultados de trabajos, comparativas de Grain, Mickey-128 y Trivium.

Los resultados obtenidos de la implementaciones en [11] denotan que Grain y Trivium son al menos 20 veces más eficientes que el algoritmo AES. Los resultados obtenidos en [14] demuestran que Mickey-128 tiene dos ventajas:

- complejidad baja en hardware y
- proveer un alto nivel de seguridad.

Además, es factible su implementación en un recurso FPGA, tarjetas inteligentes y recursos RFID.

En el mismo trabajo se realizó una comparación de la eficiencia entre el algoritmo de referencia AES y el cifrador de flujo Mickey-128 implementado. Obteniendo que AES tiene una frecuencia de 67 MHz y una velocidad de procesamiento de 2.2 Mbps usando la tarjeta XC2S16-6, y para Mickey-128 se obtuvo una frecuencia de 170 MHz y una velocidad de procesamiento de 170 Mbps usando la tarjeta XC-50ECS14.

En [6], se realiza una comparación de las arquitecturas optimizadas por mínimo de área usando una tarjeta de la familia Xilinx Spartan3, donde el parámetro *d* es el factor de paralelización que determina el número de bits del keystream producidos por ciclo de reloj; el parámetro *k* es el número de bits de la llave y del IV utilizados por ciclo de reloj. Las comparaciones se muestran en la tabla 2.7.

Cifrador de Flujo	Frecuencia de reloj	Tiempo de setup de la llave		Velocidad de procemiento máx.		Area		Eficiencia	
	MHz	ciclos	ns	Mbps	/ Grain	CLB slices	/ Grain	Mbit/s/ CLB slices	/ Grain
Grain (d=1, k=1)	193	304	1575	193	1.00	122	1.00	1.58	1.00
Mickey-128 (d=1, k=1)	156	416	2667	156	0.81	261	2.14	0.60	0.38

Trivium (d=1, k=1)	201	1312	6527	201	1.04	188	1.54	1.07	0.68

Tabla 2.7 Comparación de las arquitecturas optimizadas en mínimo de área en una Xilinx Spartan3 [12]

Los resultados de comparación de las arquitecturas optimizadas por su velocidad de procesamiento y eficiencia se muestran en la tabla 2.8.

Cifrador de Flujo	Frecuencia de reloj	Tiempo de setup de la llave		Velocidad de procesamiento máx.		Area		Eficiencia	
	MHz	ciclos	ns	Mbps	/ Trivium	CLB slices	/ Trivium	Mbps/CLB slices	/ Trivium
Grain (d=16, k=16)	155	19	123	2480	4.9	356	0.92	6.97	4.5
Mickey-128 (d=1, k=1)	156	416	2667	156	77.9	261	0.67	0.60	52.4
Trivium (d=64, k=64)	190	21	108	12160	1.0	388	1.00	31.34	1.0

Tabla 2.8 Comparación de arquitecturas optimizadas en base a la velocidad de procesamiento / área [12]

En [16] se realiza una comparación de los algoritmo de los cifradores de flujo Mickey-128 y Trivium, cuya implementación se realizó en el lenguaje VHDL y se uso el recurso Xilinx Virtex-II XC2V6000-4FF1152. Los resultados se muestran en la tabla 2.9.

Cifrador de flujo	Recurso usado	Area Slices	Frecuencia (MHz)	Vel. de Proc. (Mbps)	Eficiencia (Mbps/slice)
Mickey-128	Virtex-II	190	200	200	1.05
Trivium	Virtex-II	41	207	207	5.05

Tabla 2.9 comparación de los cifradores de flujo Mickey-128 y Trivium con respecto a la velocidad de procesamiento y eficiencia.

2.6 Conclusión del capítulo y panorama del capítulo tres.

En el presente capítulo se dió una revisión de lo que es la criptografía, de su historia y del como se divide. También se analizaron los trabajos relacionados a los cifradores de flujo Grain, Mickey-128 y Trivium. Se hizo

una comparación de los resultados reportados por los mismos concluyendo que:

- Grain es el algoritmo más fácil de entender y de implementar, le sigue en complejidad Trivium y en último Mickey-128, lo cual queda determinado por la naturaleza del diseño de los algoritmos.
- Por la forma en la cual se diseñó Grain es posible trabajar 16 bits en paralelo, ya que no se usa ninguno de los primeros 16 bits de cada registro como parámetro de entrada a las funciones de retroalimentación.
- Para Mickey-128 no es posible trabajar con mas de un bit a la vez, pues la forma en la cual se realiza la actualización de los registros no permite saber cual será el valor que tomará cualquiera de los bits que definen un parámetro de control.
- Trivium, por su parte está definido de tal manera que se puede lograr un grado de paralelización de 64 bits. Aunque está definido como un registro, éste se puede considerar como si fueran tres registros. De cada uno de los registros se obtienen los bits para definir el bit del keystream y de los bits que ocuparan la posición de uno de los registros, es decir que existe una retroalimentación de unos registros a otros.

En el capítulo tres se propone una arquitectura basada en un diseño ascendente de los cifradores de flujo Grain, Mickey-128 y Trivium. Las arquitecturas permiten trabajar con un bit o bits en paralelo, el grado máximo de paralelización queda determinado por la naturaleza del algoritmo de cada uno de los cifradores de flujo.

Capítulo 3

Arquitecturas de Grain, Mickey-128 y Trivium

En este capítulo se describen las arquitecturas que se proponen para cada uno de los cifradores de flujo Grain, Mickey-128 y Trivium. El diseño que se siguió para las arquitecturas de los cifradores de flujo fue el diseño ascendente (bottom-up). Esto, con el propósito de identificar a los componentes que definen a la arquitectura y las señales de control; además de obtener una arquitectura paralelizable desde un bit hasta el grado máximo que permite cada uno de los cifradores de flujo Grain, Mickey-128 y Trivium.

Para diseñar los componentes se usó el lenguaje VHDL en ModelSim, la síntesis se realizó en el programa Xilinx ISE 8.2i montando una tarjeta de desarrollo Spartan3 development kit.

3.1 Paralelización de los cifradores.

La metodología que se siguió para realizar la paralelización de los cifradores de flujo: Grain, Mickey-128 y Trivium, fue el diseño ascendente, es decir conceptuar hasta la unidad mínima de cada cifrador de flujo para poder manipularla de acuerdo al grado de paralelización requerida.

Para codificar cada uno de los bloques que componen a los cifradores de flujo fue usando el lenguaje VHDL. Para la etapa de síntesis se usó el programa Xilinx ISE 8.2i utilizando la tarjeta de desarrollo Spartan3 development kit XC3S400-5FG320. En la tabla 3.1 se muestra un sumario de los cifradores de flujo Grain, Mickey-128 y Trivium.

Cifrador de Flujo	Autores	Tamaño de la Llave	Tamaño del IV	Secuencia máx de Keystream	Núm. De bits internos	Componentes básicos
Grain	Matin Hell, Thomas Johansson and Willi Meier	80-bit	64-bit	2^{64}	160	LFSR, NFSR, función de salida
Mickey-128	Steve Babbage and Matthew Dodd	128-bit	128-bit	2^{64}	320	LFSR, NFSR
Trivium	Christophe De Canniere and Bart Prenell	80-bit	80-bit	2^{64}	288	LFSR, NFSR

Tabla 3.1 Características básicas de los cifradores de flujo

La interfaz usada de los cifradores de flujo es la siguiente:

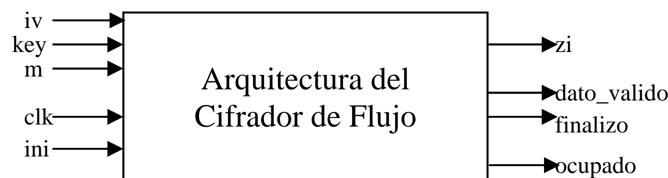

Fig. 3.1 Interfaz de los cifradores de flujo.

En *iv* entra el valor del vector de inicialización del ancho requerido para el cifrador de flujo con el cual se va a trabajar. En la señal *key* entra la llave, y al igual que en el vector de inicialización éste debe ser del ancho especificado por cada cifrador de flujo.

Para la señal *m* se define el número de bits de salida que se deseen y que además no sobrepase el máximo definido por el cifrador de flujo.

Las señales *clk* e *ini* son señales de control, la primera define la señal de reloj que se necesita para que funcione la arquitectura y la segunda es la señal de activación para que trabaje la arquitectura.

Las señales de salida se muestra el estado en el que se encuentra el cifrador de flujo. La señal *zi* se puede usar cuando lo indiquen las otras señales de salida.

Las señales *dato_valido*, *finalizo* y *ocupado* muestran el estado en el que se encuentra el cifrador de flujo. La señal *dato_valido* se activa en alto y las otras dos en bajo, lo que muestra que se encuentra trabajando; la señal *ocupado* la pondrá en alto y las otras en bajo, quiere decir que está generando datos *zi* validos, y por ultimo la señal *finalizo* se pondrá en alto una vez terminada la generación del keystream y las otras señales se pondrán en bajo.

3.2 Grain

Para diseñar la arquitectura de Grain se siguió un diseño ascendente, para definir los componentes y las señales que los controlarán. En cada uno de los bloques se encuentra la unidad mínima a la cual se podía llegar y que a la vez permitirá paralelizar el número de bits según las necesidades del usuario y al grado que lo permite la definición del algoritmo. Los grados de paralelización que se pueden manejar con el algoritmo Grain son de 1,2,4,8 y 16 bits por cada ciclo de reloj. A continuación en la Fig. 3.2 se muestran los componentes y las señales que se manejan cuando se tiene el caso base, es decir cuando se maneja sólo un bit por ciclo de reloj de la arquitectura de Grain. Y en la Fig.3.3, se tiene la arquitectura de Grain cuando se manejan dos bits en paralelo.

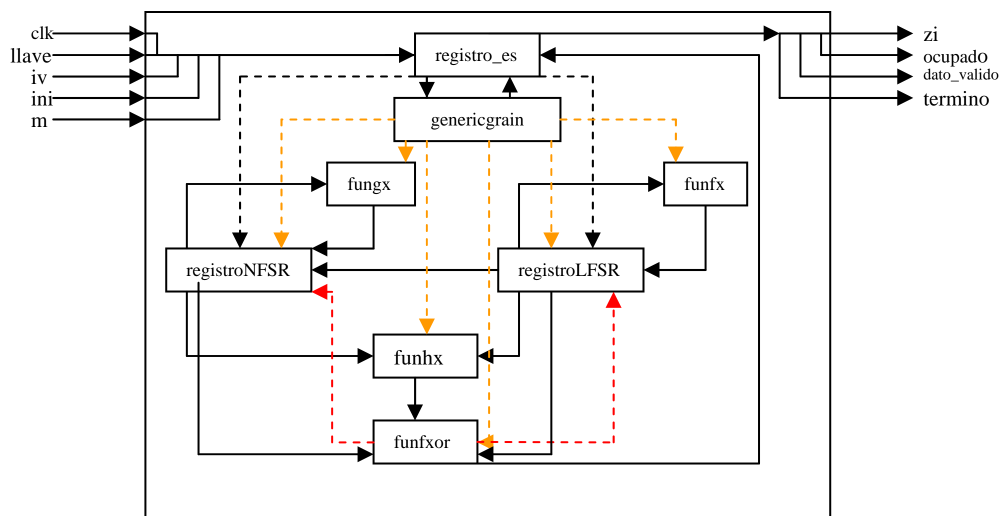

Fig. 3.2 Arquitectura de Grain en el caso base.

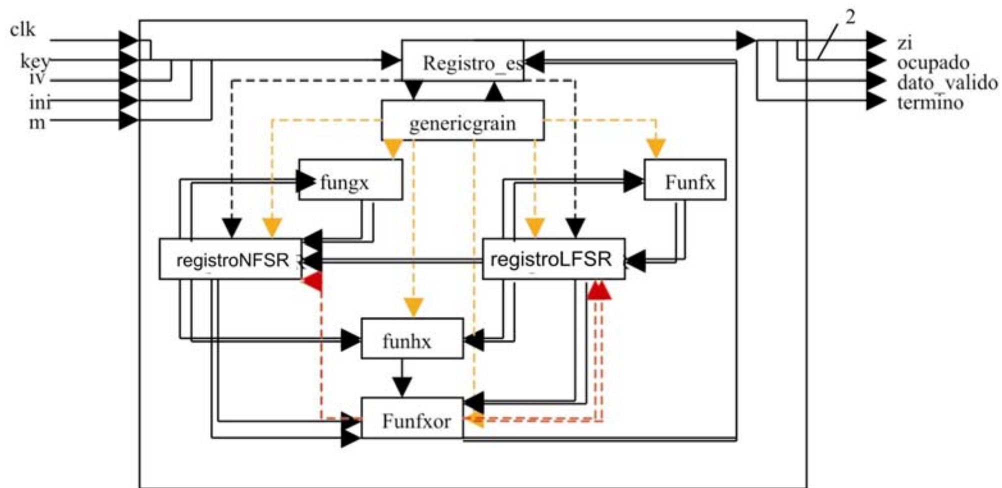

Fig. 3.3 Arquitectura de Grain para dos bits en paralelo

3.2.1 Relación entre algoritmo y componentes

El cifrador de flujo Grain está definido por dos registros y tres funciones de retroalimentación. Los registros *NFSR* y *LFSR* son de 80 bits cada uno y las funciones *g(x)* y *f(x)* son de retroalimentación. La función *h(x)* en la etapa de inicialización es una función de retroalimentación, una vez que se han estabilizado los registros la función sirve para generar el keystream.

La arquitectura está compuesta por los siguientes componentes: controlgenerico, regNFSR, regLFSR, funcionfx, funciongx, funcionhx, funcionxor y reg_es. Al componente principal, el que contiene a todos los componentes, entran las siguientes señales: *iv*, *llave*, *m*, *clk* e *ini*; y salen las: *zi*, *dato_valido*, *ocupado* y *termino*.

En el componente controlgenerico se encuentra definida la máquina de estados que controla la forma de operar de la arquitectura. En regNFSR y regLFSR se encuentran definidos los registros NFSR y LFSR respectivamente. En los componentes funcionfx, funciongx, funcionhx se encuentran definidas las funciones de retroalimentación de los registros y la función que determina el o los bits para generar el keystream, es decir, f(x), g(x) y h(x) respectivamente. En el componente funcionxor se define a la función que genera el keystream. Las

señales que entran a la arquitectura son los parámetros necesarios para generar el keystream y controlar la arquitectura. Las señales de salida son de monitoreo, para saber en que estado se encuentra la arquitectura, y el keystream.

En los siguientes párrafos se detalla la relación entre los componentes de la arquitectura de Grain y su algoritmo.

En la etapa de inicialización pasa por una serie de pasos, en el primer paso se inicializa la arquitectura, introduciendo la llave en la señal *llave* (80 bits), el IV en la señal *iv* (64 bits), la señal *ini* se pone en alto, en *m* el número de bits que se quiere que tenga el keystream que no debe exceder 2^{64} bits. El componente que contiene el algoritmo de control es el que se encargará de pasar de una etapa a otra. Una vez, que se produce el primer ciclo de reloj el *iv* y la *llave* son colocados en los registros que se encuentran en los componentes regLFSR y regNFSR respectivamente. En el siguiente ciclo de reloj, se empieza la etapa de estabilización de los registros el cual está compuesto por (160-n)/n, donde *n* es el número de bits que se trabajan en paralelo.

Durante la estabilización de los registros los componentes funcionfx, funciongx, funcionhx y la funcionxor, que definen a las funciones f(x), g(x), h(x) que calcula el valor de salida *zi* y la que calcula el o los bits que se guardarán en el registro NFSR, reciben los bits de las posiciones correspondientes de cada uno de los registros. Los bits que recibe el componente funcionfx (0 a n-1), (13 a 13+n-1), (23 a 23+n-1), (38 a 38+n-1), (51 a 51+n-1) y (62 a 62+n-1) del registro que se encuentra en el componente regLFSR; para el componente funciongx los bits son (0 a n-1), (9 a 9+n-1), (14 a 14+n-1), (15 a 15+n-1), (21 a 21+n-1), (28 a 28+n-1), (33 a 33+n-1), (37 a 37+n-1), (45 a 45+n-1), (52 a 52+n-1), (60 a 60+n-1), (62 a 62+n-1) y (63 a 63+n-1) que recibe del registro que se encuentra en el componente regNFSR. En el componente funcionhx los bits de entrada son: (3 a 3+n-1), (25 a 25+n-1), (46 a 46+n-1), (64 a 64+n-1) del componente regLFSR y (63 a 63+n-1) del componente regNFSR, también recibe los bits (1 a n), (2 a 2+n-1), (4 a 4+n-1), (10 a 10+n-1), (31 a 31+n-1), (43 a 43+n-1) y (56 a 56+n-1) del

componente regNFSR que el algoritmo determina para calcular el valor de salida de *zi*.

En la etapa de inicialización el o los bits que salen del componente funcionhx sirven como otro parámetro de entrada a los componentes regLFSR y regNFSR para actualizar los registros y lograr la estabilización de los mismos. En este proceso de estabilización la señal de salida *ocupado* está activada en alto y las otras dos señales de monitoreo se encuentran activadas en bajo.

Una vez terminado el proceso de estabilización, se empiezan a producir las salidas del *keystream* y las señales de monitoreo se activan en bajo a excepción de la señal *dato_valido* que se encuentra activada en alto. La señal *zi* empieza a mostrar bits validos de keystream.

En la etapa de generación del *keystream*, a diferencia de la etapa de inicialización, el componente funcionhx ya no funciona como parámetro para calcular los nuevos bits que se guardarán en los registros, la salida de éste funcionará como salida válida para *zi*. En esta etapa de generación se detendrá hasta calcular los m bits que definen en la señal de entrada *m*.

Una vez calculados los m bits del keystream, las señales de monitoreo pasarán a estar activadas en bajo con excepción de la señal *termino* que estará en alto.

3.3 Mickey-128

Al igual que para Grain, la metodología que se siguió para obtener la arquitectura de Mickey-128 fue el diseño ascendente, con la cual se abstrajo hasta la unidad mínima que sirviera para paralelizar en lo posible el número de bits. Durante el proceso del diseñó fue posible concluir que no pueden paralelizar operaciones, esto debido a la manera en la que se comporta el algoritmo. El algoritmo modifica totalmente los valores de los registros después de cada ciclo for que se ejecute. La arquitectura de Mickey-128 no es paralelizable a más de un bit por ciclo de reloj debido a que no se puede determinar cuál va a ser el valor

que tomarán los bits en los registros en el siguiente ciclo for. En la Fig. 3.4 se muestra la arquitectura a bloques de Mickey-128, y en las Fig. 3.5, 3.6 y 3.7 sus componentes internos.

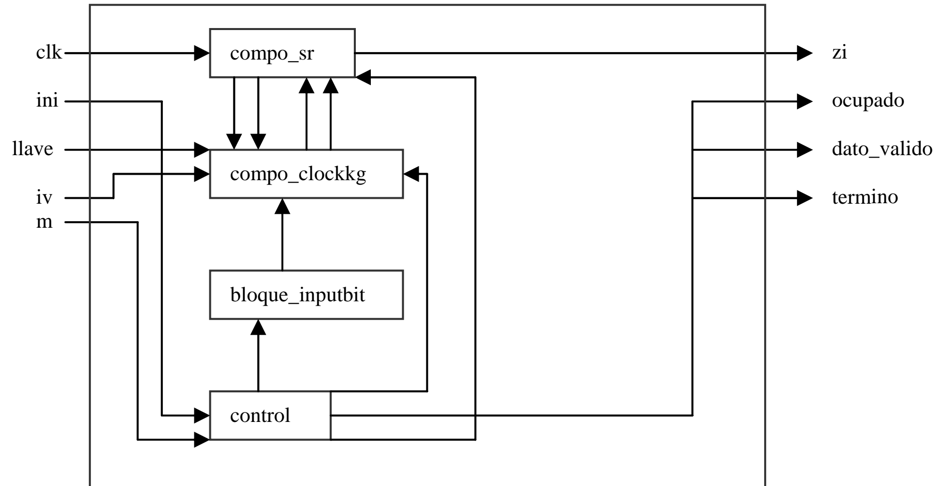

Fig. 3.4 Arquitectura ascendente de Mickey-128

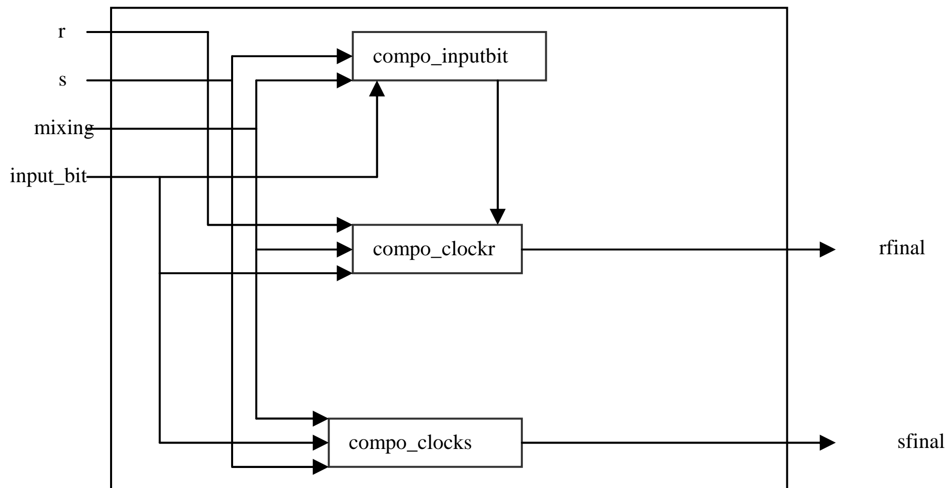

Fig. 3.5 Arquitectura del componente compo_clock_kg

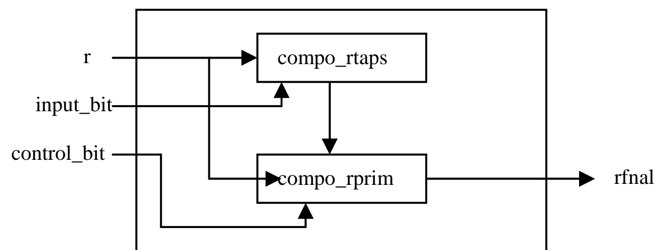

Fig. 3.6 Arquitectura del componente compo_clockr

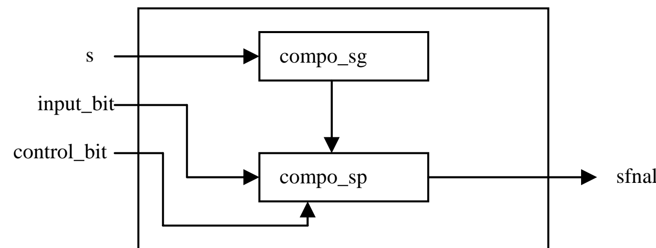

Fig. 3.7 Arquitectura del componente compo_clocks

3.3.1 Relación entre algoritmo y componente.

El algoritmo de Mickey-128 fue diseñado en forma de procedimientos, los cuales se llaman entre si para generar el keystream. En los procedimientos que hacen los llamados se definen parámetros de control que determinarán la manera en la cuál se trabajará en los procedimientos que han sido llamados. Los parámetros de control se usan para determinar la manera en la cual se tratarán los bits que componen a cada uno de los registros, que en hardware es la definición del estado de las señales que actuarán como señales de control para decidir la manera en la cual se definirán las señales de control para otros componentes p la manera en la que se modificarán los registros.

El algoritmo de Mickey-128 está definido por dos registros *R* y *S*, éstos son de 160 bits cada uno, los parámetros de entrada son la *llave* de 128 bits y el *iv* que va desde 0 a 128 bits, este número de bits lo define el usuario en el momento de usar el algoritmo, el keystream puede ser de hasta 2^{64} bits.

La arquitectura se diseñó de manera ascendente, ésta está compuesta por cuatro componentes principales: compo_sr, compo_clockkg, bloque_inputbit y control. Dentro del componente compo_clockkg se encuentran los componentes: compo_inputbit, compo_clockr y compo_clocks, compo_clockr está compuesto por compo_rtaps y compo_rprim; y compo_clocks tiene a compo_sg y compo_sp. Internamente para el registro *S* que está definido en el bloque compo_clocks, se

encuentran definidas cuatro secuencias de 128 bits cada una, estas secuencias son COMP0, COMP1 (definidas en el componente compo_sg,),FB0 y FB1 (que se encuentran en el componente compo_sp).

Las señales que se requieren de entrada a la arquitectura son: *iv* longitud determinada por el usuario, *k* de 128 bits, *n* el número de bits que se esperan de salida e *ini* para iniciar la arquitectura. Y las señales de salida son: *zi* que son los bits del keystream, *dato_valido*, *ocupado* y *termino* que son señales de monitoreo.

El nombre que tiene cada uno de los componentes es similar al que se usa en el algoritmo descrito en [13], en cada componente se encuentra descrito el procedimiento al que hace referencia.

A continuación se da una descripción más detallada de la relación existente entre el algoritmo y la arquitectura de Mickey-128, así como del manejo y función de las señales de entrada y salida.

La arquitectura pasa por las etapas de inicialización y la de generación del keystream. En la etapa de inicialización se siguen varios pasos, los cuales consisten en la inicialización de los registros *R* y *S*, que se encuentran definidos en los componentes compo_clockr y compo_clocks respectivamente. Después pasa por tres ciclos for, el primero dura de acuerdo a la longitud del iv y las señales que usan de control, *Mixing* que es enviada en alto y el *Input_bit* que toma el valor de la posición *i* del *iv*, donde *i* es número del ciclo for en el que se encuentra; el segundo ciclo for es de 128, la señal *mixing* es enviada en alto y la señal *Input_bit* toma el valor de la posición i de la llave, donde *i* va a depender del ciclo for en el cuál va; el tercero ciclo for es de 128 y es para terminar de estabilizar a los registros, la señal *Mixing* es enviada en alto y la señal *Input_bit* es enviada en bajo.

Durante el proceso de inicialización las señales se envían al componente compo_clockkg, éste componente pasa la señal *control_bit_r* al componente compo_clockr, el i*nput_bit* al componente compo_inputbit el cual manipula la entrada de acuerdo al valor de la señal *Mixing* y la señal resultante la envía al

componente compo_clockr. Las señales de *control_bit_r* e *Input_bit_s* son enviadas al componente compo_clocks.

El componente compo_clockr redefine al registro *R* y la señal *feedback*, señales que se pasan a los componentes compo_rtaps y compo_rprim. En compo_rtaps se vuelve a recalcular el valor de los bits del registro *R,* la modificación de los bits depende de si la posición que se está analizando está definida en el parámetro rtaps. El registro *R* que se obtuvo se pasa a compo_rprim en el que si la señal *control_bit_r* es igual a uno, entonces se vuelve a calcular cada uno de los bits del registro *R*.

En compo_clocks se determinan la señal *feedback* que se define como la función XOR entre la posición 159 del registro *S* y de la señal de entrada *Input_bit_s*, se pasa el registro *S* a compo_sg, en el cual se encuentran definidos los parámetros comp1 y comp0 con los que se obtienen los nuevos valores que se guardarán en el registro *S*. El actual registro *S* es enviado a compo_sp, en el cual se vuelve a recalcular el valor de los bits del registro s de acuerdo a la señal *control_bit_s* que elige entre usar el parámetro *fb1* o *fb0*. Los registros *R* y *S* finales son salidas del compo_clockkg.

En la etapa de generación del keystream es un proceso de generación que depende del número bits que se requieran, el keystream se va generando en el compo_sr con la función XOR de los bits de la posición cero de cada uno de los registros *R* y *S*. La señal *Mixing* se activa en bajo y la señal *Input_bit* se activa en bajo, estas señales se pasan a compo_clockg y se sigue el proceso ya antes mencionado.

3.4 Trivium

La arquitectura de Trivium se diseñó dividiendo el registro principal en tres partes, ya que actúan de manera independiente. De cada una de las tres partes se define un bit que servirá de retroalimentación para otra parte del registro, tomando a cada uno de los bits generados y aplicándoles la función XOR se obtiene el bit

del *keystream*. En la Fig. 3.8 se muestra la arquitectura a bloques de Trivium, en está se puede apreciar cómo cada uno de los registros manda los bits para calcular las t's, las cuales sirven para formar parte de uno de los otros dos registros y que además sirven para definir el bit del keystream.

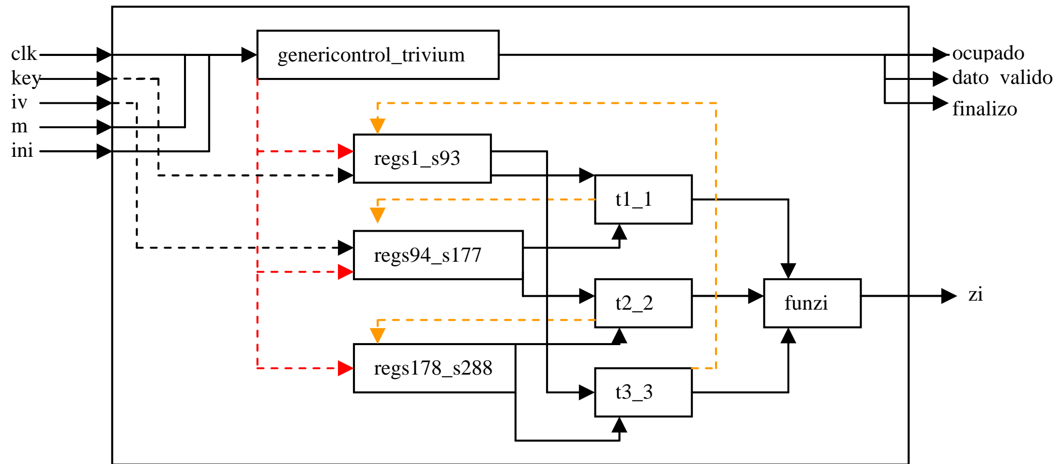

Fig. 3.8 Arquitectura de Trivium en caso base

En la Fig. 3.9 se muestra la arquitectura de Trivium cuando se manejan dos bits en paralelo, pues por la forma en la cual se encuentra diseñado el algoritmo de Trivium es posible la paralelización de 1, 2 ,4 ,8 16, 32 y 64 bits.

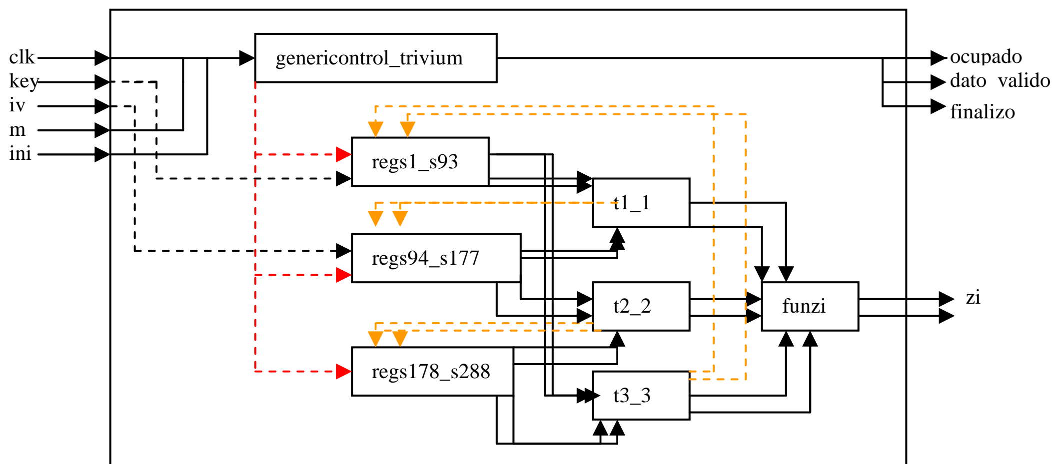

Fig. 3.9 Arquitectura de Trivium con dos bits en paralelo.

3.4.1 Relación entre algoritmo y componentes

El cifrador de flujo Trivium fue diseñado como un sólo registro de 288 bits, entran el iv de 80 bits, y la llave de 80 bits, y tiene definidas cuatro funciones. t1, t2 y t3 son las funciones que sirven para determinar los valores de las t's, la cuarta función es una función XOR sirve para definir el bit que forma parte del keystream. Éste cifrador de flujo se divide en dos etapas: inicialización y generación del keystream.

En la arquitectura que se propone para el cifrador de flujo Trivium se dividió á el registro en tres partes. Los componentes que se utilizaron en arquitectura son: genericontrol_Trivium, registro_es, regs1_193, regs94_s177, regs178_s288, t1_1, t2_2, t3_3 y funzi. Al componente principal entran las siguientes señales: *iv*, *llave*, *m*, *clk* e *ini*; y salen las siguientes señales: *zi*, *dato_valido*, *ocupado* y *finalizo*.

La manera en la cual se actualizan los bits del registro permite dividirlo en tres partes, cada parte es descrita por los componentes regs1_193, regs94_s177, regs178_s288. En los componentes t1_1, t2_2 y t3_3 son calculados los bits de retroalimentación de cada una de las tres partes del registro. En el componente funzi se genera el keystream que se muestra en la señal *zi*. Las señales de entrada son las requeridas para generar el keystream y para controlar la arquitectura, y las señales de salida sirven para monitorear el estado en el que se encuentra la arquitectura.

A continuación se profundizará en la relación de la arquitectura de Trivium con su algoritmo, del manejo y significado de las señales de monitoreo y control.

La etapa de inicialización pasa por varios pasos, el primero consiste en introducir la *llave*, *iv*, *m* e *ini* , ésta señal debe ser activada en alto. La *llave* e *iv* son de 80 bits cada uno, la *m* define el número de bits que componen al keystream e i*ni* sirven para inicializar a la arquitectura. En el componente regs1_s93, se encuentra definida la primera parte del registro, del bit 1 al 93, que es inicializado con la *llave* y los últimos trece bits son rellenados con ceros. La segunda parte del

registro, del bit 94 al 177, se encuentra definida en el componente regs94_s177 y es inicializado con el *iv* y los últimos cuatro bits con cero. Y la tercera parte del registro, del bit 178 al 288, se encuentra definida en el componente regs178_s288, éste es inicializado con ceros a excepción de los tres últimos bits que son inicializados con unos. La inicialización ocurre en el primer ciclo de reloj, a partir del segundo ciclo de reloj hasta completar (1152-n)/n ciclos de reloj, donde *n* define el número de bits en paralelo que se están manejando, y sirve para estabilizar a los registros de la arquitectura. Durante este proceso de estabilización, las señales de salida se encuentran activadas en bajo excepto *ocupado* que se encuentra activada en alto. En éste proceso de estabilización los componentes t1_1, t2_2 y t3_3 calculan mediante la función XOR, la cual toma como parámetros todos los bits que le envía el registro correspondiente, el valor del bit o los bits que servirán para determinar el nuevo valor que será guardado en los registros. El o los bits que produce el componente t1_1 es para el componente regs94_s177, los que produce el componente t2_2 es para regs178_s288 y los t3_3 son para regs1_s93. El componente regs1_s93 envía los bits (65 a 65-n+1), (92 a 92-n+1), (90 a 90-n+1), (91 a 91-n+1), (68 a 68-n+1), regs94_s177 envía (68 a 68-n+1), (83 a 83-n+1), (81 a 81-n+1), (82 a 82-n+1), (77 a 77-n+1) y de regs178_s288 son (65 a 65-n+1), (110 a 110-n+1), (108 a 108-n+1), (109 a 109-n+1) y (86 a 86-n+1).

Una vez que se ha completado el proceso de estabilización de los registros las señales de salida se activan en bajo con excepción de la señal *dato_valido* la cual se activa en alto, esto quiere decir que los bits que se encuentran en la señal *zi* son válidos y factibles de usar. Las señales se mantendrán así hasta tener en el keystream los m bits que el usuario requiere, el número total de bits del *keystream* no pueden ser más de 2^{64}.

En la etapa de generación del keystream, los componentes regs1_s93, regs94_177 y regs178_288 envían sus respectivos bits a los componentes t1_1, t2_2 y t3_3 para calcular los bits que se almacenaran en los registros y los bits

que servirán para el keystream. En el componente t1_1 se usan las posiciones (0 a n-1) y (n a 2*n-1) de la señal de entrada para calcular los bits que se envían a la funzi, este mismo resultado es utilizado junto con los bits del resto de las posiciones de la señal de entrada para calcular el valor de los bits que se enviarán al componente regs94_s177. El mismo procedimiento es el que siguen los componentes t2_2 , para funzi y regs178_s288, y t3_3, para funzi y regs1_s93.
El componente funzi recibe los bits correspondientes de t1_1, t2_2 y t3_3, y mediante una función xor se determina los bits del keystream.
Una vez que se han generado el número de bits determinados por la señal de entrada *m*, es que las señales de monitoreo se activan en bajo y la señal *finalizo* se activa en alto.

3.5 Conclusión del capítulo y panorama del capítulo cuatro

En este capítulo se han descrito las arquitecturas propuestas para cada uno de los cifradores de flujo Grain, Mickey-128 y Trivium y la manera en la cual describen al algoritmo que define a cada uno de los cifradores de flujo, así como la forma en la cual operan las señales de entrada, salida, keystream y monitoreo. Las señales de monitoreo servirán de apoyo al usuario para determinar el estado en el cual se encuentra la arquitectura. El número de bits en paralelo es determinado por el usuario y se realiza en el archivo principal de cada una de las arquitecturas. El cifrador de flujo Mickey-128 no es paralelizable a más de un bit.
En el capítulo cuatro se muestran los resultados obtenidos de la implementación de cada una de las arquitecturas en Xilinx ISE 8.2i montando la tarjeta de desarrollo Spartan3 y simulando a cada uno de los grados, uno a la vez, de paralelización que permite cada cifrador de flujo.

Capítulo 4

Resultados de implementación

En este capítulo se muestran los resultados de las arquitecturas propuestas para cada uno de los cifradores de flujo, cada una de las cuales se diseñó sin hacer uso de una técnica de optimización avanzada. Lo anterior con el fin de poder comparar las arquitecturas bajo las mismas circunstancias, es decir, sin usar sintetizadores ni dispositivos diferentes. Las arquitecturas fueron diseñadas usando el lenguaje VHDL en la herramienta de simulación ModelSim usando un diseño estructural. El código VHDL fue sintetizado con XST de Xilinx ISE 8.2i usando un recurso FPGA. La tarjeta usada fue la Spartan3 development kit con el dispositivo XC3S400-5FG320 para el cual se sintetizó cada una de las arquitecturas de los cifradores de flujo. Se mostrarán los resultados de síntesis y de eficiencia de cada una de las arquitecturas, dichos resultados serán comparados entre sí y como propósito de referencia también serán comparados con los de otros trabajos.

4.1 Ámbito de desarrollo de la tesis.

Para desarrollar la siguiente tesis se utilizó una computadora con las siguientes características:

- Procesador Pentium 4 a 2.93 GHz
- 1 Gb en RAM
- Sistema operativo Windows XP Home Edition v. 2002

El lenguaje utilizado para describir a la arquitectura de cada uno de los algoritmos de cifrado de flujo fue VHDL en el programa ModelSim usando un diseño estructural. El código de la arquitectura fue sintetizado en Xilinx ISE 8.2i. La tarjeta usada fue la Spartan3 development kit con el dispositivo XC3S400-5FG320.

Para verificar el correcto funcionamiento de las arquitecturas se realizó una comparación con aplicaciones java de los algoritmos, estas aplicaciones fueron desarrolladas por los diseñadores de cada algoritmo, y con archivos de texto que se encuentran disponibles en la página del proyecto eStream, en estos archivos se definen los vectores de prueba y la secuencia de salida esperada.

4.2 Resultados de Grain.

En la tabla 4.1 se muestran los resultados de síntesis indicando el número de Slices, LUTs e IOBs.

Grado de paralelización	Area Slices	Area LUTs	Entradas y Salidas IOBs
1	156	280	182
2	220	402	183
4	252	463	185
8	305	564	189
16	425	794	197

Tabla 4.1 Resultados de síntesis de la arquitectura Grain optimizados por velocidad

En la tabla 4.1 se observa que el área no aumenta de manera proporcional al número de bits que son procesados en paralelo, ya que se reutilizan componentes y sólo son agregados los necesarios.

La eficiencia en cada grado de paralelización se muestra en la tabla 4.2. En la tabla se muestra la frecuencia, tiempo, velocidad de procesamiento, área y eficiencia para los siguientes grados de paralelización 1,2,4,8 y 16.

# de bits en paralelo	Frecuencia máx. de reloj MHz	Tiempo ns	Vel. de Proc. Mbps	Area Slices	Eficiencia Mbps/slices
1	140.531	7.116	140	156	0.897
2	140.531	7.116	280	220	1.272
4	140.531	7.116	560	252	2.222
8	140.531	7.116	1120	305	3.672
16	140.531	7.179	2240	425	5.270

Tabla 4.2 Eficiencia de la arquitectura Grain con diferentes grados de paralelización optimizados por velocidad

En la tabla anterior se mantiene la frecuencia y el tiempo, a pesar de que se aumenta el grado de paralelización. La velocidad de procesamiento aumenta rápidamente mientras que el área requerida no crece aceleradamente.

Con esta arquitectura si lo que se requiere es acelerar la transmisión sin sacrificar por mucho el área a ocupar se pueden elegir cuatro bits en paralelo lo cual da una velocidad de procesamiento de 560 Mbps y un área de 252 slices.

4.2.1 Comparación con otros trabajos

En la tabla 4.3 se hace una comparación de este trabajo con respecto a otros realizados sobre este cifrador de flujo. La comparación que se puede realizar con [11], no es del todo justa debido a que varios de los datos son omitidos en el reporte revisado, y estos son el dispositivo usado para la implementación y el número de Slices. Con respecto a la frecuencia, ésta se comparara sólo con los trabajos que reportan el dispositivo usado, quedando en el siguiente orden [10] la frecuencia más baja, la del presente trabajo y la más alta [12]. En [12], se sintetizó

con Synplicity Synplify Pro para Xilinx. En este trabajo se usó el sintetizador XST [VHDL/Verilog]. El área que ocupa en Grain es de 59 Slices mayor que la de [12].

Trabajo con 16 bits en paralelo	Dispositivo	Frecuencia máx. de reloj MHz	Vel. de Proc. Mbps	Area Slices
[10]	Xc2s15-5	105	1680	n.d
[11]	n.d	300	4475	n.d
[12]	Xc3s50pq208-5	155	2480	356
Este trabajo	xc3s400-5fg320	140.531	2240	425

Tabla 4.3 Comparación con trabajos anteriores

4.3 Resultados de Mickey-128

La tabla 4.4 muestra los resultados de síntesis indicando el número de Slices, LUTs e IOBs.

Tamaño del IV	Área Slices	Área LUTs	Entradas y salidas IOBs
1	315	568	135
2	316	570	136
4	317	572	138
8	321	581	142
16	374	677	150
32	353	645	166
64	368	668	198
128	385	701	262

Tabla 4.4 Resultados de síntesis de la arquitectura Mickey-128 optimizados por velocidad

En la tabla 4.4 el crecimiento del área no es proporcional al tamaño del iv, es decir, no se va duplicando el área tal como lo hace el tamaño del iv.

La eficiencia con respecto al tamaño del iv se muestra en la tabla 4.5. En la tabla se muestra el tamaño del iv 1,2,4,8,16, 32, 64 y 128 para los cuales se muestran la frecuencia, tiempo , eficiencia , área y eficiencia /área.

# de bits del iv	Frecuencia máx. de reloj MHz	Tiempo ns	Vel. de Proc. Mbps	Area Slices	Eficienica Mbps/slices
1	96.584	10.354	96	315	0.0104
2	96.584	10.354	192	316	0.608
4	95.938	10.423	380	317	1.199
8	95.344	10.488	760	321	2.368
16	80.217	12.466	1280	374	3.422
32	84.217	11.874	2688	353	7.615
64	83.471	11.980	5312	368	14.435
128	89.165	11.215	11392	385	29.590

Tabla 4.5 Eficiencia de la arquitectura Mickey-128 con diferentes tamaños de iv optimizada por velocidad

En la tabla anterior se aprecia el cambio en la frecuencia y el tiempo con respecto al tamaño del iv. La velocidad de procesamiento aumenta rápidamente mientras que el área crece relativamente poco de un tamaño de iv a otro.

La arquitectura de Mickey-128 se puede usar con cualquiera de los tamaños de iv permitidos por las restricciones del mismo algoritmo, ya que el área a ocupar no crece en proporción al tamaño del iv. La frecuencia va disminuyendo conforme va aumentando el tamaño del iv.

4.3.1 Comparación con otros trabajos

En la tabla 4.6 se hace mención de trabajos reportados sobre Mickey-128 manejando un bit a la vez, al igual que de los datos que se proporcionan sobre el mismo. La comparación no es del todo posible debido a que, como se puede

apreciar en la tabla, el tipo de dispositivos en los trabajos [1] y [14] son Virtex, lo cual no da condiciones similares para realizar la evaluación de una manera justa. Denotando que el único trabajo que hace uso de un Spartan3 es el presente dando una frecuencia de 95 MHz y 315 Slices.

Trabajo	Dispositivo	Frecuencia máx. de reloj MHz	Vel. de Proc. Mbps	Area Slices
[1]	V400efg676	166	166	167
[11]	n.d.	308	287	n.d.
[14]	Xcv50ecs144	170	170	167
Este trabajo	Xc3s400-5fg320	95.584	96	315

Tabla 4.6 Comparación con trabajos anteriores

4.4 Resultados de Trivium

En la tabla 4.7 se muestran los resultados de síntesis indicando el número de Slices, LUTs e IOBs.

Grado de paralelización	Slices	LUTs	IOBs
1	217	281	198
2	251	384	199
4	256	399	201
8	257	414	205
16	266	463	213
32	300	559	229
64	398	755	261

Tabla 4.7 Resultados de síntesis de la arquitectura Trivium optimizados por velocidad

En la tabla 4.7, se denota que la diferencia entre el área que ocupan 2,4,8,16 y 32 bits en paralelo es pequeña comparada a ir de un bit a 64 bits, que aunque es apreciablemente mayor hay que considerar el hecho de que el área no crece en proporción al número de bits en paralelo que se usan, al igual que los recursos de la tarjeta que se usa.

La eficiencia en cada grado de paralelización se muestran en la tabla 4.8. En la tabla se muestran los grados de paralelización 1,2,4,8,16,32 y 64, para cada grado de paralelización es reportada la frecuencia, tiempo , velocidad de procesamiento, área y eficiencia.

# de bits en paralelo	Frecuencia máx. de reloj MHz	Tiempo ns	Vel. de Proc. Mbit/s	Área Slices	Eficiencia Mbps/slices
1	136.606	7.320	136	217	0.627
2	136.606	7.320	272	251	1.083
4	136.606	7.320	544	256	2.125
8	136.606	7.320	1088	257	4.233
16	136.606	7.320	2176	266	8.180
32	136.606	7.320	4352	300	14.505
64	136.606	7.320	8704	398	21.869

Tabla 4.8 Eficiencia de la arquitectura Trivium con diferentes grados de paralelización

En la tabla 4.8, tanto la frecuencia como el tiempo se mantienen constantes para todos los grados de paralelización. El throghput aumenta proporcionalmente a la frecuencia y grado de paralelización, mientras que el área no aumenta de manera proporcional ni a la frecuencia ni al grado de paralelización.

El área que ocupa la arquitectura de Trivium aumenta ligeramente cuando se pasa de 2 a 4 ó de 4 a 8 ó de 8 a 16 bits en paralelo, esto permite aumentar el grado de paralelización sin sacrificar el área.

4.4.1 Comparación con otros trabajos

En la tabla 4.9 se hace una comparación del presente trabajo con respecto a otros ya realizados sobre Trivium. En [11] no se reportan ni el dispositivo usado para la implementación al igual que el área ocupada por la misma. Así, se puede comparar el presente trabajo sólo con [10], [1] y [12], pues reportan el dispositivo usado. El orden de frecuencias que van desde la más baja a la más alta queda de la siguiente manera [10], la de este trabajo, [12] y [1], de estos resulta más justa la comparación con [12] ya que hace uso de un Spartan3, en [10] de un Spartan2 y en [1] de un Virtex. Para [12] el sintetizador usado fue un Synplicity Synplify Pro para Xilinx y el usado para este trabajo fue XST [VHDL/Verilog]. De entre estos dos últimos trabajos la diferencia con respecto al área es muy pequeña, de diez Slices.

Trabajo con 64 bits en paralelo	Recurso	Frecuencia máx. de reloj MHz	Vel. De Proc. Mbps	Slices
[10]	Xc2s15-5	102	6528	n.d
[1]	V400efg676	211	211	144
[11]	n.d	312	18568	n.d.
[12]	Xc3s400fg320-5	190	12160	388
Este trabajo	Xc3s400-5fg320	136.606	8704	398

Tabla 4.9 Comparación con trabajos anteriores sobre Trivium

4.5 Comparación de los resultados de las arquitecturas Grain, Mickey-128 y Trivium.

A continuación se muestra en la tabla 4.10 la comparación de los resultados de las arquitecturas Grain, Mickey-128 y Trivium optimizadas por el mínimo de área y, además, se manejan en sus casos base, en el caso de Mickey-128 con el máximo número de bits que puede tener el iv.

Cifrador de Flujo	Frecuencia máx. de reloj MHz	Tiempo ns	Vel. de Proc. Mbps	Area Slices	Eficiencia Mbps/slices
Grain (1 bit)	116.220	8.604	116	156	0.744
Mickey-128 (1 bit)	48.307	20.701	48	398	0.121
Trivium (1 bit)	126.844	7.884	126	216	0.583

Tabla 4.10 Comparación de las arquitecturas optimizadas por el mínimo de área

En los casos base que se muestran en la tabla 4.10 para cada una de las arquitecturas se muestra que Grain es la que presenta la mejor frecuencia y la que menos área requiere, la arquitectura que le sigue es la de Trivium y la última es la de Mickey-128.

En la tabla 4.11 se hace la comparación de los resultados de la optimización de las arquitecturas con respecto a la velocidad.

Cifrador de Flujo	Frecuencia máx. de reloj MHz	Tiempo ns	Vel. de Proc. Mbps	Area Slices	Eficicencia Mbps/slices
Grain (16 bits)	140.531	7.155	2240	428	5.234

Mickey-128 (1 bit)	89.165	11.215	89	385	0.213
Trivium (64 bits)	136.606	7.320	8704	398	21.869

Tabla 4.11 Comparación de las arquitecturas optimizadas por el máx. de velocidad y máx grado de paralelización

Con respecto a la optimización por velocidad, la mejor arquitectura es la de Grain pues muestra una frecuencia mayor comparada a las otras dos arquitecturas, pero con respecto al área es la que más ocupa comparándola con Trivium. En este caso no se puede comparar totalmente a Grain y a Trivium con Mickey-128, pues ésta última no puede manejar más que un bit por ciclo de reloj. Mientras que Grain y Trivium si pueden trabajar con bits en paralelo. Trivium es la mejor opción para manejar un alto grado de paralelización sin sacrificar el área siguiéndole Grain.

En la tabla 4.12 se hace una comparación exclusiva de los resultados de las arquitecturas de Grain y Trivium, debido a que con éstas se puede paralelizar el número de bits con los que se puede trabajar la arquitectura.

# de bits en paralelo	Frecuencia máx. de reloj MHz		Tiempo ns		Vel. de Proc. Mbps		Área Slices		Eficiencia Mbps/slices	
	Grain	Trivium	Grain	Trivium	Grain	Trivium	Grain	Trivium	Grain	Trivium
1	140.531	136.606	7.116	7.320	140	136	156	217	0.897	0.627
2	140.531	136.606	7.116	7.320	280	272	220	251	1.272	1.083
4	140.531	136.606	7.116	7.320	560	544	252	256	2.222	2.125
8	140.531	136.606	7.116	7.320	1120	1088	305	257	3.672	4.233
16	140.531	136.606	7.179	7.320	2240	2176	425	266	5.270	8.180

Tabla 4.12 Comparación en diferentes grados de paralelización de Grain y Trivium

El análisis de la tabla 4.12 se puede hacer desde dos puntos de vista, el primero de ellos es con base a la velocidad de procesamiento, con respecto a la cual la

arquitectura de Grain es la más práctica con el fin de hacer el envío más rápido con cualquier grado de paralelización; el segundo es con respecto al área que ocupa la arquitectura, siendo la de Trivium la más factible de usar. La eficiencia de Trivium es notablemente mayor que la de Grain debido a que ocupa menos área y su velocidad de procesamiento está ligeramente por debajo de la de Grain, además es capaz de trabajar con 32 y 64 bits en paralelo.

4.6 Conclusión del capítulo y panorama del capítulo cinco

En este capítulo se mostraron los resultados obtenidos por cada una de las arquitecturas con ayuda de la herramienta Xilinx ISE 8.2i. Las comparaciones realizadas se hicieron con respecto a la velocidad y el área, además de que se comparó el presente trabajo con los trabajos que se han presentado hasta este momento.

Grain es la arquitectura más veloz de las tres manejando desde 1 a 16 bits en paralelo. Trivium es la arquitectura que ocupa menos área de las tres y puede manejar 1, 2, 4, 8, 16, 32 ó 64 bits en paralelo. De las tres arquitecturas, la de Trivium es la más viable a elegir debido a que su velocidad de procesamiento esta ligeramente debajo de Grain, mientras que el área a ocupar si es notablemente menor, además es posible paralelizar hasta 64 bits.

Con lo que respecta a Mickey-128, en las dos comparaciones realizadas éste es el que está en mayor desventaja pues no permite paralelizar, y tiene una frecuencia menor a las reportadas para Grain y Trivium, y el área que usa es mayor con respecto a las otras dos arquitecturas.

En el siguiente capítulo se darán las conclusiones que se han obtenido con la realización del presente trabajo.

Capítulo 5

Conclusiones

En este capítulo se mencionan las conclusiones a las cuales se llegaron en el proceso de la realización del presente trabajo. Las conclusiones obtenidas se basan en la complejidad del diseño y entendimiento, la velocidad de procesamiento que tiene cada una de las arquitecturas y el área que ocupa cada una de ellas. También, se hace una revisión de los objetivos tanto del general como de los específicos.

5.1 Conclusiones

- Grain es un cifrador de flujo fácil de entender y de implementar por la forma en la cual se encuentra diseñado el algoritmo, siguiéndole en complejidad el cifrador de flujo Trivium y por último Mickey-128. Mickey-128 es complejo de implementar, debido a la manera en la cual se encuentra definido, que es mediante procesos que se llaman unos a otros, y por la manera en la cual trabaja el mismo, ya que se modifica totalmente cada uno de los registros en cada ciclo de reloj.
- La arquitectura de Grain tiene una mejor velocidad de procesamiento que la arquitectura de Trivium cuando se manejan 1, 2 4, 8 y 16 bits en paralelo. Mickey-128 es omitido en la comparación debido a que sólo puede trabajar con un bit a la vez.
- Cuando se requiera consumir poca área y trabajar con 1, 2 ó 4 bits en paralelo, la opción es trabajar con la arquitectura de Grain; en caso de requerir poca área y de trabajar con 4, 8 ó 16 bits en

paralelo se debe trabajar con la arquitectura de Trivium, la cual también trabaja con 32 ó 64 bits en paralelo.

Para trabajos futuros se propone aplicar técnicas de optimización avanzada a las arquitecturas propuestas, con el objetivo de mejorar la velocidad de procesamiento y la eficiencia. Reducir el área que ocupa cada una de las arquitecturas, en especial la arquitectura de Mickey-128, pues ésta ocupa muchos recursos por lo cuál requiere de más área que las arquitecturas de Grain y Trivium.

5.2 Revisión de objetivos

El objetivo principal es comparar las arquitecturas hardware optimizadas de cada uno de los algoritmos de cifrado de flujo Grain, Mickey-128 y Trivium en velocidad y área.

- Colocando a su máximo grado de paralelización a cada una de las arquitecturas se obtuvo que fue la arquitectura de Grain la que presentó la frecuencia más alta que es de 140.531MHz siguiéndole la arquitectura de Trivium con 136.606MHz y por último se encuentra la arquitectura de Mickey-128 que tiene 89.165MHz.
- Los resultados obtenidos optimizando por el mínimo de área en el caso base de un bit, Grain requiere de un área de 156 Slices siguiéndole Trivium con 216 Slices y Mickey-128 con 398 Slices.

Uno de los objetivos particulares fue el diseño e implementación de arquitecturas base para cada uno de los algoritmos.

- La parte del diseño e implementación se realizó en dos etapas, la primera consistió en diseñar una arquitectura para cada uno de los algoritmos de cifrado de flujo (Grain, Mickey-128 y Trivium) sin hacer uso de una estrategia de diseño, esto con el propósito de analizar y entender a cada uno de los algoritmos. En la segunda etapa se

siguió un diseño ascendente (bottom-up) para cada uno de los cifradores de flujo, el cual se definió mediante componentes. Siguiéndole el proceso de hacer una arquitectura genérica. Para la verificación del correcto funcionamiento de cada una de las arquitecturas se usaron los vectores de prueba.

- En las arquitecturas genéricas es posible modificar los parámetros de entrada, esto con el fin de definir el número de bits en paralelo que se manejarán (el número de bits en paralelo sólo puede ser modificado en el código, ya que lo que se representa es hardware y no puede modificarse algo que en teoría es físico), en caso de que el cifrador de flujo lo permita. Una vez obtenida la arquitectura genérica se optimizó a cada una de las arquitecturas definidas tanto en área como en velocidad. En área, Grain es la mejor arquitectura, esto en la optimización por velocidad y con un máximo grado de paralelización, también es la mejor para la optimización en velocidad en el caso base.

Otro de los objetivos particulares fue el de realizar un estudio comparativo de los resultados obtenidos de las implementaciones.

- Se comparó a cada una de las arquitecturas tomando en cuenta la complejidad para ser implementadas (por la manera en la cual fueron diseñados cada uno de sus algoritmos), la velocidad en MHz y el área en Slices que requiere cada una de las arquitecturas. La arquitectura de Grain es fácil de implementar y es la que presenta una frecuencia mayor en comparación a las otras dos arquitecturas. Trivium es fácil de entender y de implementar, y de cuya arquitectura se obtuvo que hace un menor uso de recursos que Grain, y que su frecuencia, aunque es muy poca la diferencia, es menor a la de Grain. Mickey-128 es el cifrador de flujo más complejo de entender y de implementar, por lo que su arquitectura es más compleja que la de los otros dos

cifradores de flujo, requiere de una mayor área y la frecuencia es menor.

El último de los objetivos particulares fue el definir las condiciones bajo las cuales se adapta mejor cada uno de los algoritmos, esto de acuerdo a las restricciones que presenta cada uno de ellos y de los resultados obtenidos.

- Grain es un cifrador de flujo que puede implementarse en hardware, cuyo algoritmo está definido mediante el uso de dos registros y funciones de retroalimentación, puede manejar 1, 2 , 4 , 8 ó 16 bits en paralelo. Esta arquitectura se definió mediante componentes y los de los resultados obtenidos se obtiene una frecuencia mayor en comparación a las arquitecturas de los cifradores de flujo Grain y Mickey-128. La arquitectura de Grain presenta un bajo consumo de recursos, los cuales no aumentan de manera proporcional. Así, el tipo de aplicaciones para las cuales es factible el uso de esta arquitectura es en aquellas en las cuales se requiera del envío de la información de manera rápida y de un bajo consumo de recursos.
- Mickey-128 es el algoritmo más complejo de los tres analizados en el presente trabajo. La arquitectura presentó una frecuencia menor en comparación a las arquitecturas de Grain y Trivium. La arquitectura se diseñó con componentes y algunos de estos están definidos por otros componentes, de ahí el porque requiere de una mayor área que la que ocupan las otras dos arquitecturas. Otra restricción que presenta este algoritmo es el de sólo poder manejar un bit. Por lo que, el tipo de aplicaciones para las cuales es factibles usar la arquitectura de Mickey-128 es en aquellas en las que no se requiera de manejar bits en paralelo pero si de una manera segura, tal como el envío de un archivo que contenga un número confidencial o información restringida.

- Trivium, es un cifrador de flujo con vías a ser implementado en hardware. Su algoritmo esta definido por un registro circular y que puede manejar 1, 2, 4, 8, 16 y 32 bits en paralelo. La arquitectura se basó en el concepto de dividir el registro circular en tres partes, ya que por la forma diseño del algoritmo esto es posible. De la arquitectura diseñada e implementada se encontró que maneja una frecuencia relativamente menor a la que maneja la arquitectura de Grain. Lo más importante con relación a ésta arquitectura radica en que el área no aumenta de manera proporcional a como se incrementa el grado de paralelización, más bien lo hace de manera gradual. El tipo de aplicaciones para las cuales resulta ser útil esta arquitectura es en aquellas en las que se requiera de hacer un envío rápido de la información y un bajo consumo de recursos, un ejemplo puede ser en la transmisión por medios móviles.

Bibliografía

[1] P. Kitsos "Hardware Implementations for the ISO/IEC 180033-4: 2005 Standard for Stream Ciphers", International Journal of Signal Processing V. 3 N. 1,2006

[2] Disponible en www.ecrypt.eu.org/stream

[3] Disponible en www.wikipedia.org

[4] National Bureau of Standards, Data Encryption Standard, FIPS-Pub.186-2. National Bureau of Standards, U.S. Department of Commerce, Washington D.C., 2000.

[5] Disponible en https://www.icai.es/contenidos/publicaciones/anales_get.php?id=1210/42-46_introcritogrf_(I-2006)-1210[1].pdf

[6] R. Rivest, et al., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". Communications of the ACM, Vol. 21 (2), pp.120–126. 1978. Previously released as an MIT Technical Memo in April 1977.

[7] http://theory.lcs.mit.edu/~rivest/

[8] Joan Daemen,Vincent Rijmen: "The Rijndael Block Cipher", sep 1999. http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf

[9] Y. Lu, "Applied Stream ciprés in Mobile Communications". Disponible en http://biblion.epfl.ch/EPFL/theses/2006/3491/3491_abs.pdf

[10] T. Good, et al, "Review of stream cipher candidates from a low resource hardware perspective". Disponible en http://www.ecrypt.eu.org/stream/hw.html

[11] F. K. Gurkaynak, et al, "Hardware evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt",

eSTREAM, ECRYPT Stream Cipher Project. Disponible en http://www.ecrypt.eu.org/stream/hw.html

[12] K. Gaj, et al, "Comparison of hardware performance of selected Phase II eSTREAM candidates", eSTREAM, ECRYPT Stream Cipher Project.

[13] S. Babbage y Matthew Dodd, "The stream cipher Mickey-128 2.0", eSTREAM, ECRYPT Stream Cipher Project. Disponible en http://www.ecrypt.eu.org/stream/hw.html

[14] P. Kitsos, "On the Hardware Implementation of the Mickey-128 Stream Cipher", eSTREAM, ECRYPT Stream Cipher Project. Disponible en http://eprint.iacr.org/2005/301

[15] P. Bulens, et al., "FPGA Implementations of eSTREAM Phase-2 focus Candidates with Hardware Profile", eSTREAM, ECRYPT Stream Cipher Project.

[16] A. Alex, et al., "Hardware Accelerated Novel Protein Identification", Field-Programmable Logic and Applications, 14th International Conference, September 2004.

[17] "Applied stream ciphers in mobile communications". Disponible en http://lasecwww.epfl.ch/pub/lasec/doc/Lu06.pdf

[18] S. Babbage, "Stream ciphers- what does industry want?". Disponible en http://www.ecrypt.eu.org/stvl/sasc/slides21.pdf

[19] L. Batina, et al., "Testing Framework for eSTREAM Profile II Candidates". Disponible en http://www.cosic.esat.kulevven.be/ecrypt/stream/papersdir/2006/014.pdf

[20] B. L. Buzbee, "The Efficiency of Parallel Procesing". Disponible en http://www.fas.org/sgp/othergov/does/lanl/pubs/00326993.pdf

[21] G. Caracuel Ruiz, "Criptografía, Seguridad informatica"

[22] "Chapter 6. Stream Ciphers". Disponible en http://www.cacr.math.uwaterloo.ca/hac/about/chap6.pdf

[23] C. De Canniere y B. Preneel, “Trivium, A Stream Cipher Construction Inspired by Block Cipher Design Principles”, eSTREAM, ECRYPT Stream Cipher Project.

[24] R. Elbaz, et al., “Hardware Engines for Bus Encryption: a Survey of Existing Techniques”, Desing, Automation and test in Europe, vol. 3, marzo 2005

[25] “Energy, performance, area versus security trade-offs for stream ciphers”. Disponible en http://www.ecrypt.eu.org/stvl/sasc/slides24.pdf

[26] M. Feldhofer, “Comparison of Low-Power Implemenattions of Trivium and Grain”. Disponible en http://www.cosic.esat.kulevven.be/ecrypt/stream/papersdir

[27] M. Galanis, et al., “Comparison of the Hardware Implementation of Stream Ciphers”, the International Arab Journal of Information Technology, V. 2, N. 4, October 2005.

[28] M. Galanis, et al., “Comparison of the Performance of Stream Ciprés for Wireless Communications”, eSTREAM, ECRYPT Stream Cipher Project.

[29] F. K. Gurkaynak, “Recommendations for Hardware Evaluation of Cryptographic Algorithms”, eSTREAM, ECRYPT Stream Cipher Project. Disponible en http://www.ecrypt.eu.org/stream/hw.html

[30] M. Hell, et al., “Grain- A Stream Cipher for Constrained Environments”. Disponible en http://www.ecrypt.eu.org/stream/ciphers/grain/grain.pdf

[31] P. Kitsos, et al., “Hardware implementation of Bluetooth security”, IEEE Pervasive computing, Enero- Marzo 2003.

[32] P. Kocher, et al., ”Security as New Dimension in embedded System Design”. Disponible en http://palms.ee.princeton.edu/PALMSopen/Lee-41stDAC_46_1.pdf

[33] P. Koopman, “Embedded System Security”, Julio 2004.

[34] W. Stallings, “Cryptography an Network Security. Principles an Practices”, ed. 4a. Prentice Hall

[35] H. X. Mel y D. Baker, “Criptography Decrypted”, Addison Wesley.

[36] C. J. Mitchell y A. W. Dent, “International standards for stream ciphers: A progress report”, eSTREAM, ECRYPT Stream Cipher Project.

[37] J. Nakajima, et al., “ Cipher Algorithm Implementation”, reporte técnico.

[38] W. Roelandts, “FPGAs and the Era of field Programmability”, Field-Programmable Logic and Applications, 14th International Conference, September 2004.

[39] M. Sonmez Turan, et al., “Statistical Analysis of Synchronous Stream Ciphers”. Disponible en http://www.ecrypt.eu.org/stream/papersdir/2006/012.pdf

[40] “Stream Ciphers”. Disponible en http://www.quadibloc.com/crypto/co0408.htm

[41] http://www.uninet.edu/6fevu/text/criptografia.htm

[42] “What is a stream cipher?”. Disponible en http://www.rsa.com/rsalabs/node.asp?id=2174

[43] J. Zambreno, et al., “Exploring Area / Delay Tradeoffs in an AES FPGA Implementation”, Field- Programmable Logic and Applications, 14th International Conference, September 2004.

[44] E. Zenner, “Stream Cipher Criteria”, eSTREAM, ECRYPT Stream Cipher Project.

Apéndice A. Grain

En este apéndice se muestra la arquitectura del algoritmo, la descripción de los componentes y el código diseñado en el lenguaje VHDL que define a la arquitectura.

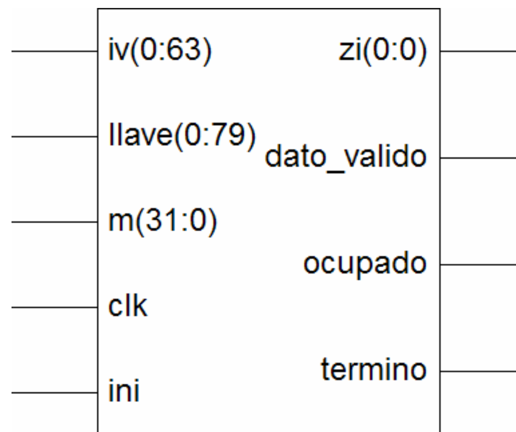

Fig. A.1 Componente Grain generado por la herramienta Xilinx ISE 8.2i

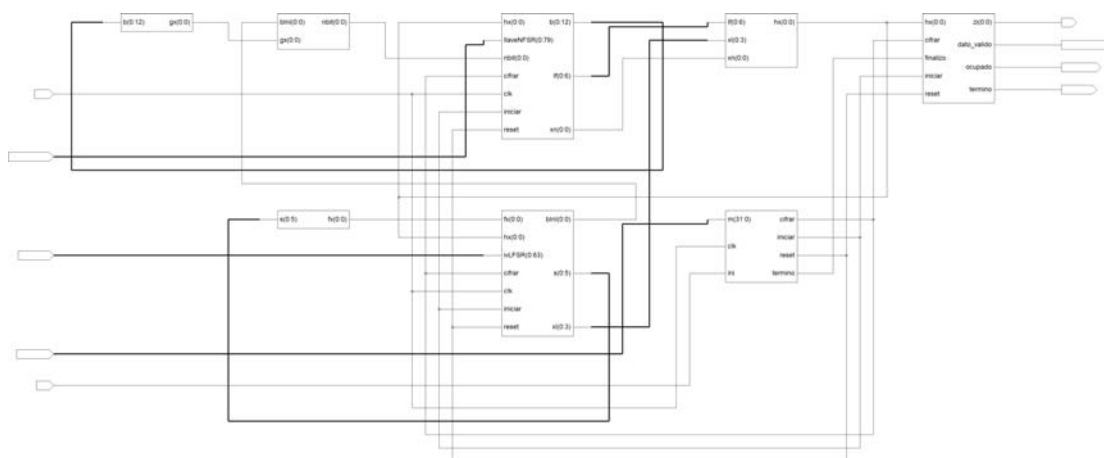

Fig. A.2 Diagrama RTL de Grain generado por la herramienta Xilinx ISE 8.2i

Nombre de componente	Componentes Internos	Señales de Entrada	Señales de Salida	Descripción del componente
Cifrador_grain_generico	registro_es genericgrain registroNFSR registroLFSR funfx fungx funhx funfxor	clk ini llave iv m	zi ocupado dato_valido termino	Componente principal, el cual contiene al a los componentes que conforman la arquitectura de grain.
registro_es		reset iniciar cifrar finalizo hx	ocupado dato_valido termino zi	La función de este componente consiste en servir como registro de las señales de entrada y de las señales de salida, lo cual ayuda a evitar la perdida de la señales y evitar retrazo en la llegada y salida de las señales.
genericgrain		clk ini m cifrar	iniciar reset termino	En este componente se encuentra el algoritmo de control, es decir que es aquí donde se toman las decisiones para pasar a las siguientes etapas por las que esta compuesto el algoritmo para poder obtener el keystream.

registroNFSR		clk reset iniciar cifrar llaveNFSR	b x xn lf hx nbit	Este componente define al registro no lineal que componen al algoritmo, este registro se encuentra conformado por 80 posiciones cuyos valores se van recorriendo y colocando el nuevo valor en la posición cero, este nuevo valor se obtiene de realizar un xor de dos señales de entrada.
registroLFSR		clk reset iniciar cifrar ivLFSR fx hx	s xl blnl	En este componente se encuentra definido el registros lineal, cuyos valores se van recorriendo y dejando en la posición cero el valor del bit calculado de efectuar la operación xor de entre dos señales de entrada.
funfx		s	fx	Aquí se efectúa la función f(x), la cual va a servir de entrada al registro lineal para calcular el valor del nuevo bit.
fungx		b	gx	En este componente se ejecuta la función g(x), la cual va a servir para calcular el valor del nuevo bit de la posición cero en el registro no lineal.

funhx		xn xl lf	hx	Aquí se efectúa la función h(x) que va a servir en la etapa de inicialización como una entrada apara calcular el nuevo bit de la posición cero de los registros, pero en la etapa de generación del keystream va a ser uno delos parámetros a considerar para obtener el elemento del keystream.
funfxor		gx blnl	nbit	La función que obtiene el elemento que forma parte del keystream considerando varias entradas de posiciones especificas de los registros al igual que el bit resultante de la función h(x) ejecutando entre ellas la operación xor.

Código de los componentes.

Componente principal. Cifrador_Grain_generico

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity Cifrador_Grain_generico is-- se puede paralelizar hasta 16 bits
   Generic(n:integer:=16);
   port(clk: in std_logic;
       ini: in std_logic;
       llave: in std_logic_vector(0 to 79);
       iv: in std_logic_vector(0 to 63);
       m: integer;
       zi: out std_logic_vector(0 to n-1);
       ocupado: out std_logic;
       dato_valido: out std_logic;
       termino: out std_logic);
   end;
architecture Grain of Cifrador_Grain_generico is
   component registro_es is
        Generic(n: integer);
              port (
                   reset: in std_logic;--asigna valor al registro
          iniciar: in std_logic;--realiza los ciclos para ajustar registros
                        cifrar: in std_logic;--para iniciar el cifrado
          finalizo: in std_logic;
                        hx: in std_logic_vector(0 to n-1);
                      ocupado: out std_logic;
                      dato_valido: out std_logic;
                      termino: out std_logic;
                      zi: out std_logic_vector(0 to n-1)
                      );
   end component;
   component genericgrain is
        Generic(n: integer);
         PORT(
           clk     : IN     std_logic;
           ini     : IN     std_logic;
           m       : IN     integer;
           cifrar  : OUT    std_logic;
           iniciar : OUT    std_logic;
           reset   : OUT    std_logic;
           termino : OUT    std_logic
```

```
        );
    end component;
    component registroNFSR is
        Generic(n:integer);
        port (
            clk:in std_logic;--señal de reloj
            reset: in std_logic;--asigna valor al registro
                iniciar: in std_logic;--realiza los ciclos para ajustar registros
                cifrar: in std_logic;--para iniciar el cifrado
                llaveNFSR: in std_logic_vector(0 to 79);--valor de la llave
              b: out std_logic_vector(0 to 13*n-1);--
b0,b9,b14,b15,b21,b28,b33,b37,b45,b52,b60,b62, b63                x: out
std_logic_vector(0 to 5);--s3,s25,s46,s64,b63
              xn: out std_logic_vector(0 to 1*n-1);--b63
              lf: out std_logic_vector(0 to 7*n-1);--b1,b2,b4,b10,b31,b43,b56
              hx: in std_logic_vector(0 to n-1);
              nbit: in std_logic_vector(0 to n-1)
          );
    end component;
    component registroLFSR is
        Generic(n: integer);
        port (
            clk:in std_logic;--señal de reloj
            reset: in std_logic;--asigna valor al registro
                iniciar: in std_logic;--realiza los ciclos para ajustar registros
                cifrar: in std_logic;--para iniciar el cifailedfrado
                ivLFSR: in std_logic_vector(0 to 63);--valor de la llave
              s: out std_logic_vector(0 to 6*n-1);--s0, s13, s23, s38, s51, s62
              xl: out std_logic_vector(0 to 4*n-1);--s3, s25, s46, s64
              blnl: out std_logic_vector(0 to n-1);
              fx: in std_logic_vector(0 to n-1);
              hx: in std_logic_vector(0 to n-1)
           );
    end component;
    component funfx is
              Generic( n: integer);
              port (
                  s: in std_logic_vector(0 to 6*n-1);--valores para f(x)
                      fx: out std_logic_vector(0 to n-1)--resultado de las
funciones
                 );
          end component;
          component fungx is
            Generic(n:integer);
        port (
```

```
          b: in std_logic_vector(0 to 13*n-1);--valores para g(x)
              gx: out std_logic_vector(0 to n-1)--resultado de las funciones
          );
   end component;
   component funhx is
     Generic(n:integer);
          port (
          xn: in std_logic_vector(0 to 1*n-1);--b63
          xl: in std_logic_vector(0 to 4*n-1);
               lf: in std_logic_vector(0 to 7*n-1);--valores de la funcion lineal
               hx: out std_logic_vector(0 to n-1)--resultado de las funciones
               );
         end component;
         component funfxor is
     Generic(n: integer);
          port (
               gx: in std_logic_vector(0 to n-1);--
               blnl: in std_logic_vector(0 to n-1);--
                   nbit: out std_logic_vector(0 to n-1)
               );
         end component;
        signal reset: std_logic;
        signal iniciar: std_logic;
        signal cifrar: std_logic;
        signal finalizo: std_logic;
        signal b: std_logic_vector(0 to 13*n-1);
        signal xn: std_logic_vector(0 to 1*n-1);
  signal lf: std_logic_vector(0 to 7*n-1);
  signal hx: std_logic_vector(0 to n-1);
  signal nbit: std_logic_vector(0 to n-1);
  signal s: std_logic_vector(0 to 6*n-1);
  signal xl: std_logic_vector(0 to 4*n-1);
  signal blnl: std_logic_vector(0 to n-1);
  signal fx: std_logic_vector(0 to n-1);
  signal gx: std_logic_vector(0 to n-1);

begin
   icontrolgrain: genericgrain Generic map(n=>n)
        port map
        ( clk => clk,
          ini => ini,
          m => m,
          cifrar => cifrar,
          iniciar => iniciar,
          reset => reset,
```

```
        termino => finalizo);
  iNFSR: registroNFSR Generic map(n=>n)
       port map
       ( clk => clk,
         reset => reset,
                iniciar => iniciar,
                cifrar => cifrar,
                llaveNFSR => llave,
             b => b,
             xn => xn,
             lf => lf,
             hx => hx,
             nbit => nbit);
  iLFSR: registroLFSR Generic map(n=>n)
               port map
               (clk => clk,
                reset => reset,
                       iniciar => iniciar,
                       cifrar => cifrar,
                       ivLFSR => iv,
                    s => s,
                    xl => xl,
                    blnl => blnl,
                    fx => fx,
                    hx => hx);
         ifx:funfx Generic map(n=>n)
               port map
               ( s => s,
                     fx => fx);

         igx:fungx Generic map(n => n)
         port map
         (b => b,
              gx => gx);
   ihx:funhx Generic map(n => n)
                port map
                (xn => xn,
              xl => xl,
                 lf => lf,
                     hx => hx);

         ifxor:funfxor Generic map(n => n)
         port map
         (gx => gx,
                     blnl => blnl,
```

```
                    nbit => nbit);

          ireges:registro_es Generic map(n => n)
         port map
         ( reset => reset,
           iniciar => iniciar,
                      cifrar => cifrar,
          finalizo => finalizo,
                      hx => hx,
                    ocupado => ocupado,
                    dato_valido => dato_valido,
                    termino => termino,
                    zi => zi);
end architecture;
```

Componente genericgrain.

```
-- VHDL Entity graingenerico_lib.genericgrain.interface
--
-- Created:
--          by - Blanca.UNKNOWN (ARTEMISA)
--          at - 12:42:54 13/03/2007
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2004.1 (Build 41)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY genericgrain IS
   Generic(n: integer);
   PORT(
      clk     : IN     std_logic;
      ini     : IN     std_logic;
      m       : IN     integer;
      cifrar  : OUT    std_logic;
      iniciar : OUT    std_logic;
      reset   : OUT    std_logic;
      termino : OUT    std_logic
   );

-- Declarations

END genericgrain ;
```

```
--
-- VHDL Architecture graingenerico_lib.genericgrain.controlgenerico
--
-- Created:
--          by - Blanca.UNKNOWN (ARTEMISA)
--          at - 12:42:54 13/03/2007
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2004.1 (Build 41)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ARCHITECTURE controlgenerico OF genericgrain IS

  -- Architecture Declarations
  signal contador:integer;

  TYPE STATE_TYPE IS (
    Sinicio,
    S_reset,
    S_iniciar,
    S_cifrar
  );

  -- State vector declaration
  ATTRIBUTE state_vector : string;
  ATTRIBUTE state_vector OF controlgenerico : ARCHITECTURE IS
"current_state" ;


  -- Declare current and next state signals
  SIGNAL current_state : STATE_TYPE ;
  SIGNAL next_state : STATE_TYPE ;

BEGIN

  -----------------------------------------------------------------
  clocked : PROCESS(
    clk
  )
  -----------------------------------------------------------------
  BEGIN
    IF (clk'EVENT AND clk = '1') THEN
```

```
      current_state <= next_state;
      -- Default Assignment To Internals
      contador <= 0;

      -- Combined Actions for internal signals only
      CASE current_state IS
      WHEN S_reset =>
        contador <= 0;
      WHEN S_iniciar =>
        contador <= contador + 1;
        IF ( contador = (160-n)/n) THEN
           contador<= 0 ;
        END IF;
      WHEN S_cifrar =>
        contador<=contador+1;
      WHEN OTHERS =>
        NULL;
      END CASE;


    END IF;

  END PROCESS clocked;

  ---------------------------------------------------------------
  nextstate : PROCESS (
    contador,
    current_state,
    ini,
    m
   )
  ---------------------------------------------------------------
  BEGIN
    CASE current_state IS
    WHEN Sinicio =>
      IF (ini = '1' ) THEN
        next_state <= S_reset;
      ELSE
        next_state <= Sinicio;
      END IF;
    WHEN S_reset =>
        next_state <= S_iniciar;
    WHEN S_iniciar =>
      IF ( contador = (160-n)/n) THEN
        next_state <= S_cifrar;
```

```
      ELSE
        next_state <= S_iniciar;
      END IF;
    WHEN S_cifrar =>
      IF (contador= m/n) THEN
        next_state <= Sinicio;
      ELSE
        next_state <= S_cifrar;
      END IF;
    WHEN OTHERS =>
      next_state <= Sinicio;
    END CASE;

  END PROCESS nextstate;

  ------------------------------------------------------------------
  output : PROCESS (
    contador,
    current_state,
    m
  )
  ------------------------------------------------------------------
  BEGIN
    -- Default Assignment
    cifrar <= '0';
    iniciar <= '0';
    reset <= '0';
    termino <= '0';
    -- Default Assignment To Internals

    -- Combined Actions
    CASE current_state IS
    WHEN Sinicio =>
      reset<='0';
      iniciar<='0';
      cifrar<='0';
    WHEN S_reset =>
      reset <= '1' ;
      cifrar <= '0' ;
      iniciar <= '0' ;
      termino <= '0';
    WHEN S_iniciar =>
       iniciar <= '1' ;
      cifrar <= '0' ;
      reset <= '0' ;
```

```
      termino <= '0' ;
    WHEN S_cifrar =>
      cifrar<='1';
      iniciar<='0';
      reset<='0';
      termino<='0';
      IF (contador= m/n) THEN
        termino<='1';
        cifrar<='0';
      END IF;
    WHEN OTHERS =>
      NULL;
    END CASE;


  END PROCESS output;

  -- Concurrent Statements

END controlgenerico;
```

Componente registroNFSR.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity registroNFSR is
  Generic(n:integer);
        port (clk:in std_logic;--señal de reloj
              reset: in std_logic;--asigna valor al registro
                    iniciar: in std_logic;--realiza los ciclos para ajustar registros
                    cifrar: in std_logic;--para iniciar el cifrado
                    llaveNFSR: in std_logic_vector(0 to 79);--valor de la llave
                  b: out std_logic_vector(0 to 13*n-1);--
b0,b9,b14,b15,b21,b28,b33,b37,b45,b52,b60,b62, b63                    x: out
std_logic_vector(0 to 5);--s3,s25,s46,s64,b63
                  xn: out std_logic_vector(0 to 1*n-1);--b63
                  lf: out std_logic_vector(0 to 7*n-1);--b1,b2,b4,b10,b31,b43,b56
                  hx: in std_logic_vector(0 to n-1);
                  nbit: in std_logic_vector(0 to n-1));
        end registroNFSR;
architecture regNFSR of registroNFSR is
 signal NFSR: std_logic_vector(0 to 79);--registro de 79 bits
begin
```

```
    process(clk, reset, llaveNFSR, iniciar, cifrar, hx)
      begin
       if ((clk'event) and (clk = '1')) then
         if(reset='1' ) then
            NFSR<=llaveNFSR;--se carga la llave en NFSR
            elsif(iniciar='1') then
                NFSR(0 to 79)<=NFSR(n to 79)& (nbit xor hx);
                elsif(cifrar='1')then
                    NFSR(0 to 79)<=NFSR(n to 79)& nbit;
          end if;
       end if;
    end process;
 b<=NFSR(0 to n-1) & NFSR(9 to 9+(n-1)) & NFSR(14 to 14+(n-1)) &
NFSR(15 to 15+(n-1)) & NFSR(21 to 21+(n-1)) & NFSR(28 to 28+(n-1)) &
NFSR(33 to 33+(n-1)) & NFSR(37 to 37+(n-1)) & NFSR(45 to 45+(n-1)) &
NFSR(52 to 52+(n-1)) & NFSR(60 to 60+(n-1)) & NFSR(62 to 62+(n-1)) &
NFSR(63 to 63+(n-1));
 xn<=NFSR(63 to 63+(n-1));
 lf<= NFSR(1 to 1+(n-1)) & NFSR(2 to 2+(n-1)) & NFSR(4 to 4+(n-1)) &
NFSR(10 to 10+(n-1)) & NFSR(31 to 31+(n-1)) & NFSR(43 to 43+(n-1)) &
NFSR(56 to 56+(n-1));

end architecture;
```

Componente registroLFSR

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity registroLFSR is

   Generic(n: integer);
          port (clk:in std_logic;--señal de reloj
                reset: in std_logic;--asigna valor al registro
                       iniciar: in std_logic;--realiza los ciclos para ajustar registros
                       cifrar: in std_logic;--para iniciar el cifrado
                       ivLFSR: in std_logic_vector(0 to 63);--valor de la llave
                     s: out std_logic_vector(0 to 6*n-1);--s0, s13, s23, s38, s51, s62
                     xl: out std_logic_vector(0 to 4*n-1);--s3, s25, s46, s64
                     blnl: out std_logic_vector(0 to n-1);
                     fx: in std_logic_vector(0 to n-1);
```

```
                    hx: in std_logic_vector(0 to n-1));
   end registroLFSR;
architecture regLFSR of registroLFSR is
 signal LFSR: std_logic_vector(0 to 79);--registro de 79 bits
 signal lbit: std_logic_vector(0 to n-1);--nuevo val de LFSR
begin

    process(clk, reset, ivLFSR, iniciar, cifrar, fx, hx)
      begin
       if ((clk'event) and (clk = '1')) then
         if(reset='1' ) then
            LFSR<=ivLFSR & "1111111111111111";--se carga el IV y se compl.
con unos
            elsif(iniciar='1') then
               LFSR(0 to 79)<=LFSR(n to 79)& (lbit xor hx);
               elsif(cifrar='1')then
                  LFSR(0 to 79)<=LFSR(n to 79)& lbit;
         end if;
       end if;
    end process;

    s<=LFSR(0 to n-1)& LFSR(13 to 13+(n-1))& LFSR(23 to 23+(n-1)) &
LFSR(38 to 38+(n-1)) & LFSR(51 to 51+(n-1)) & LFSR(62 to 62+(n-1));
    xl<=LFSR(3 to 3+(n-1)) & LFSR(25 to 25+(n-1)) & LFSR(46 to 46+(n-1)) &
LFSR(64 to 64+(n-1));
    blnl<=LFSR(0 to n-1);
    lbit<=fx;


end architecture;
```

Componente funfx

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity funfx is
   Generic(n: integer);
          port (s: in std_logic_vector(0 to 6*n-1);--valores para f(x)
                    fx: out std_logic_vector(0 to n-1));--resultado de las funciones
          end funfx;
architecture funcionfx of funfx is
begin
     ffx:for i in 0 to (n-1) generate
```

```
            fx(i)<=s(i) xor s(1*n+i) xor s(2*n+i) xor s(3*n+i) xor s(4*n+i) xor
s(5*n+i);
    end generate ffx;
end architecture;
```

Componente fungx

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity fungx is
  Generic(n:integer);
        port (b: in std_logic_vector(0 to 13*n-1);--valores para g(x)
                  gx: out std_logic_vector(0 to n-1));--resultado de las funciones
        end fungx;
architecture funciongx of fungx is
   signal var1: std_logic_vector(0 to n-1);
   signal var2: std_logic_vector(0 to n-1);
   signal var3: std_logic_vector(0 to n-1);
   signal var4: std_logic_vector(0 to n-1);
   signal var5: std_logic_vector(0 to n-1);
   signal var6: std_logic_vector(0 to n-1);
   signal var7: std_logic_vector(0 to n-1);
   signal var8: std_logic_vector(0 to n-1);
   signal var9: std_logic_vector(0 to n-1);
   signal var10: std_logic_vector(0 to n-1);
   signal var11: std_logic_vector(0 to n-1);
begin
   fgx:for i in 0 to (n-1) generate
      var1(i)<=b(12*n+i) and b(10*n+i);
      var2(i)<=b(7*n+i) and b(6*n+i);
      var3(i)<=b(3*n+i) and b(1*n+i);
      var4(i)<=(b(10*n+i) and b(9*n+i)) and b(8*n+i);
      var5(i)<=(b(6*n+i) and b(5*n+i)) and b(4*n+i);
      var6(i)<=(b(12*n+i) and b(8*n+i)) and (b(5*n+i) and b(1*n+i));
      var7(i)<=(b(10*n+i) and b(9*n+i)) and (b(7*n+i) and b(6*n+i));
      var8(i)<=(b(12*n+i) and b(10*n+i)) and (b(4*n+i) and b(3*n+i));
      var9(i)<=((b(12*n+i) and b(10*n+i)) and (b(9*n+i) and b(8*n+i))) and
b(7*n+i);
      var10(i)<=((b(6*n+i) and b(5*n+i)) and (b(4*n+i) and b(3*n+i))) and
b(1*n+i);
      var11(i)<=((b(9*n+i) and b(8*n+i)) and (b(7*n+i) and b(6*n+i))) and
(b(5*n+i) and b(4*n+i));
```

```
      gx(i)<=b(i) xor b(1*n+i) xor b(2*n+i) xor b(4*n+i) xor b(5*n+i) xor b(6*n+i)
xor b(7*n+i) xor b(8*n+i) xor b(9*n+i) xor b(10*n+i) xor b(11*n+i) xor var1(i)
xor var2(i) xor var3(i) xor var4(i) xor var5(i) xor var6(i) xor var7(i) xor var8(i)
xor var9(i) xor var10(i) xor var11(i);
     end generate fgx;
end architecture;
```

Componente funhx

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity funhx is
   Generic(n:integer);
        port (xn: in std_logic_vector(0 to 1*n-1);--b63
            xl: in std_logic_vector(0 to 4*n-1);
              lf: in std_logic_vector(0 to 7*n-1);--valores de la funcion lineal
                  hx: out std_logic_vector(0 to n-1));--resultado de las funciones
        end funhx;
architecture funcionhx of funhx is
begin
    fhx:for i in 0 to (n-1) generate
      hx(i)<=(xl(1*n+i) xor xn(i) xor (xl(i) and xl(3*n+i)) xor (xl(2*n+i) and
xl(3*n+i)) xor (xl(3*n+i) and xn(i)) xor ((xl(i) and xl(1*n+i)) and xl(2*n+i)) xor
((xl(i) and xl(2*n+i)) and xl(3*n+i)) xor ((xl(i) and xl(2*n+i)) and xn(i)) xor
((xl(1*n+i) and xl(2*n+i)) and xn(i)) xor ((xl(2*n+i) and xl(3*n+i)) and xn(i))) xor
(lf(i) xor lf(1*n+i) xor lf(2*n+i) xor lf(3*n+i) xor lf(4*n+i) xor lf(5*n+i) xor
lf(6*n+i));
    end generate fhx;
end architecture;
```

Componente funxor

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity funfxor is
   Generic(n: integer);
        port (gx: in std_logic_vector(0 to n-1);
                  blnl: in std_logic_vector(0 to n-1);
                  nbit: out std_logic_vector(0 to n-1));
   end funfxor;
architecture funcionxor of funfxor is
```

```
begin
    fxor:for i in 0 to (n-1) generate
            nbit(i)<=gx(i) xor blnl(i);
    end generate fxor;
end architecture;
```

Componente regsitro_es

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity registro_es is
   Generic(n: integer);
              port (
                 reset: in std_logic;--asigna valor al registro
                   iniciar: in std_logic;--realiza los ciclos para ajustar registros
                       cifrar: in std_logic;--para iniciar el cifrado
                   finalizo: in std_logic;
                       hx: in std_logic_vector(0 to n-1);
                     ocupado: out std_logic;
                     dato_valido: out std_logic;
                     termino: out std_logic;
                     zi: out std_logic_vector(0 to n-1)
                     );
        end registro_es;
architecture reg_es of registro_es is
begin

    process(reset, iniciar, cifrar, finalizo)
          begin
            if(reset='1' or iniciar='1') then
              ocupado<='1';
              dato_valido<='0';
              termino<='0';
              elsif(cifrar='1')then
                  ocupado<='1';
                  dato_valido<='1';
                  termino<='0';
              elsif(finalizo='1') then
            ocupado<='0';
                  dato_valido<='0';
                  termino<='1';
            end if;
         end process;
```

```
 zi<= hx;
end architecture;
```

Apéndice B. Mickey-128

En este apéndice se muestra la arquitectura del algoritmo, la descripción de los componentes de la arquitectura y el código de cada uno de los componentes, los cuales están escritos en el lenguaje VHDL.

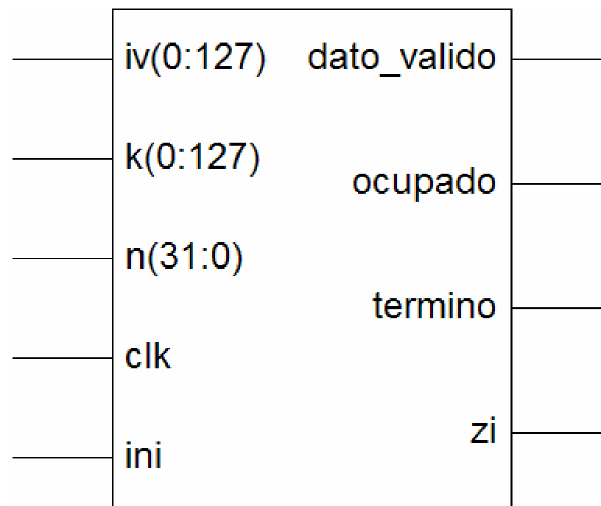

Fig. B.1 Componente Mickey-128 generado por la herramienta Xilinx ISE 8.2i

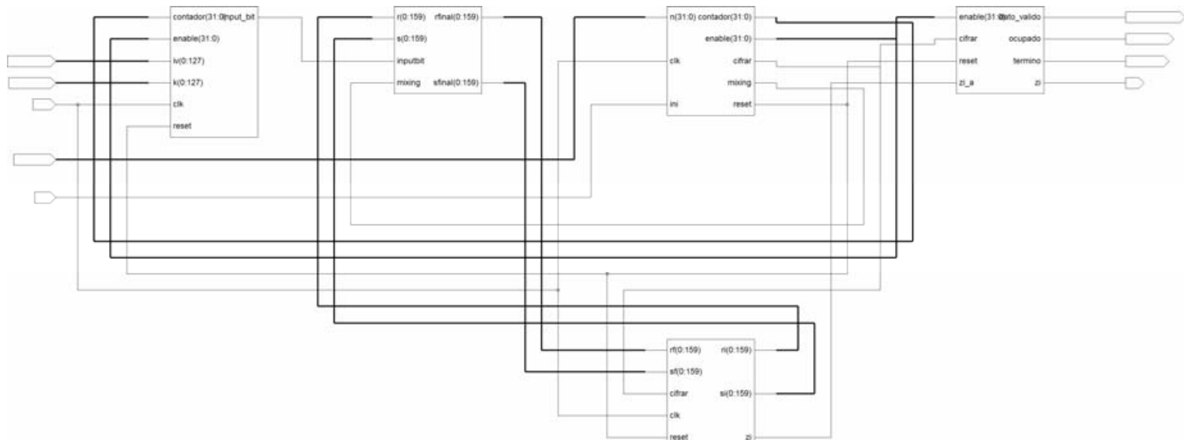

Fig. B.2 Diagrama RTL de Mickey-128 generado por la herramienta Xilinx ISE 8.2i

Nombre de componente	Componentes Internos	Señales de Entrada	Señales de Salida	Descripción del Componente
bloquemickey	compo_sr compo_clockkg bloque_inputbit control	clk ini llave iv m	zi ocupado dato_valido termino	Componente principal, el cual se encuentra compuesto a su vez por los componentes que definen a la arquitectura que describe al algoritmo de Mickey-128.
registro_es		reset enable cifrar zi_a	ocupado dato_valido termino zi	Este componente es para definir el estado en el que se encontraran las señales de monitoreo.
compo_sr		clk reset cifrar rf sf	ri si zi	Componente en el que se definen los registros y se actualizan los mismos con los nuevos valores.
compo_clockkg	compo_inputbit compo_clockr compo_clocks	r s mixing inputbit	rfinal sfinal	En este componente se define tanto el bit que ayuda a definir una línea de control así como los valores de las líneas de control para los componentes compo_clockr y compo_clocks.
bloque_inputbit		clk reset iv k enable contador	input_bit	Aquí se obtiene el bit que servirá como entrada a los otros componentes y que servirá para tomar una decisión con respecto a la operación que se les realizara a los bits de los registros r y s.

control		clk ini n	cifrar contador enable mixing reset	Este componente define al algoritmo de control de la arquitectura de Mickey-128 y que permite inicializar a los registros y pasar a la etapa de generación del keystream.
compo_inputbit		mixing input_bit1 s80	input_bit	Aquí se define la línea de control principal para realizar una acción u otra para el manejo de los bits de los registros.
compo_clockr	compo_rtaps compo_rprim	r input_bit control_bit	rfinal	Este componente se encuentra compuesto por compo_rtaps y compo_rprim. Con los cuales se define el comportamiento de la función de modificación de los elementos del registro r.
compo_clocks	compo_sg compo_sp	s input_bit control_bit	sfinal	Componente que define la modificaión de los elementos del registro s, este componente a su vez se encuentra compuesto por compo_sg y compo_sp.
compo_rtaps		ri feedback	rp	Se define la primer modificación del registro r cuyo resultado de registro pasa al componente compo_rprim.
compo_rprim		rp r control_bit	rfinal	Recibe el resultado del componente compo_rtaps para nuevamente modificarlo y dar como resultado el valor del registro final.
compo_sg		s	sg	Se modifica el registro s en base a los valores de las señales de control, su salida pasa al siguiente componente compo_sp.

compo_sp		sg feedback control_bit	si	Modifica al valor del registro que le proporciono el componente compo_sg para dar como salida el valor final del registro.

Código de los componentes.

Componente principal. bloquemickey

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity mickey is
   Generic(niv: integer:=128);
        port (
             iv: in std_logic_vector(0 to niv-1);
             k: in std_logic_vector(0 to 127);
             clk: in std_logic;
             ini: in std_logic;
             n: integer;
             zi: out std_logic;
                         ocupado: out std_logic;
                         dato_valido: out std_logic;
                         termino: out std_logic
            );end mickey;

architecture bloquemickey of mickey is
  component compo_clockkg is
            port ( r: in std_logic_vector(0 to 159);
              s: in std_logic_vector(0 to 159);
                  mixing: in std_logic;
                  inputbit: in std_logic;
                  rfinal: out std_logic_vector(0 to 159);
                  sfinal: out std_logic_vector(0 to 159)
                 );end component;

  component  compo_sr is
            port ( clk: in std_logic;
                  reset: in std_logic;
                  cifrar: in std_logic;
            rf: in std_logic_vector(0 to 159);
                  sf: in std_logic_vector(0 to 159);
                  ri: out std_logic_vector(0 to 159);
                  si: out std_logic_vector(0 to 159);
                  zi: out std_logic
                 );end component;

        component bloque_inputbit is
            Generic(niv: integer);
```

```
        port ( clk: in std_logic;
            reset: in std_logic;
            iv: in std_logic_vector(0 to niv-1);
            k: in std_logic_vector(0 to 127);
            enable: integer;
            contador: integer;
            input_bit: out std_logic
           );end component;

     component control IS
        Generic(niv: integer);
    PORT( clk      : IN     std_logic;
        ini      : IN     std_logic;
        n        : IN     integer;
        cifrar   : OUT    std_logic;
        contador : OUT    integer;
        enable   : OUT    integer;
        mixing   : OUT    std_logic;
        reset    : OUT    std_logic
       );end component;
  component registro_es is
    port( reset: in std_logic;--asigna valor al registro
        enable: integer;--realiza los ciclos para ajustar registros
                cifrar: in std_logic;--para iniciar el cifrado
        zi_a: in std_logic;
                ocupado: out std_logic;
              dato_valido: out std_logic;
              termino: out std_logic;
              zi: out std_logic
               );end component;
signal si: std_logic_vector(0 to 159);
signal ri: std_logic_vector(0 to 159);
signal input_bit: std_logic;
signal mixing: std_logic;
signal rf: std_logic_vector(0 to 159);
signal sf: std_logic_vector(0 to 159);
signal reset: std_logic;
signal cifrar: std_logic;
signal enable: integer;
signal contador: integer;
signal zi_a: std_logic;

begin
  ireges: registro_es port map
           (  reset => reset,
```

```
          enable => enable,
                      cifrar => cifrar,
          zi_a => zi_a,
                      ocupado => ocupado,
                    dato_valido => dato_valido,
                    termino => termino,
                    zi => zi);
   icompo_sr: compo_sr port map
               ( clk => clk,
                 reset => reset,
                 cifrar => cifrar,
                 rf => rf,
                 sf => sf,
                 ri => ri,
                 si => si,
                 zi => zi_a);
          icompo_clockkg: compo_clockkg port map
               ( r => ri,
             s => si,
                 mixing => mixing,
                 inputbit => input_bit,
                 rfinal => rf,
                 sfinal => sf );
          ibloque: bloque_inputbit Generic map(niv=>niv)
          port map
               ( clk => clk,
                 reset => reset,
                 iv => iv,
                 k => k,
                 enable => enable,
                 contador => contador,
                 input_bit => input_bit);
           icontrol: control Generic map(niv=>niv)
           port map
          ( clk => clk,
            ini => ini,
            n  => n,
            cifrar => cifrar,
            contador => contador,
            enable => enable,
            mixing => mixing,
            reset => reset);
end architecture;
```

Componente registro_es

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity registro_es is
   port (
                  reset: in std_logic;--asigna valor al registro
         enable: integer;--realiza los ciclos para ajustar registros
                        cifrar: in std_logic;--para iniciar el cifrado
         zi_a: in std_logic;
                        ocupado: out std_logic;
                     dato_valido: out std_logic;
                     termino: out std_logic;
                     zi: out std_logic
                      );
        end registro_es;
architecture reg_es of registro_es is
begin

    process(reset, enable, cifrar)
         begin
           if(reset='1' or enable=0 or enable=1 or enable=2) then
              ocupado<='1';
              dato_valido<='0';
              termino<='0';
              elsif(cifrar='1')then
                  ocupado<='1';
                  dato_valido<='1';
                  termino<='0';
              else
            ocupado<='0';
                  dato_valido<='0';
                  termino<='1';
            end if;
         end process;
 zi<= zi_a;
end architecture;
```

Componente compo_sr

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
```

```
use IEEE.std_logic_arith.all;
entity compo_sr is
        port (
      clk: in std_logic;
            reset: in std_logic;
            cifrar: in std_logic;
      rf: in std_logic_vector(0 to 159);
            sf: in std_logic_vector(0 to 159);
            ri: out std_logic_vector(0 to 159);
            si: out std_logic_vector(0 to 159);
            zi: out std_logic
           );end compo_sr;

architecture cifradormickey of compo_sr is
   signal r: std_logic_vector(0 to 159);
   signal s: std_logic_vector(0 to 159);

   begin
      process(clk, reset,cifrar, rf, sf)
          begin
             if((clk'event) and (clk='1')) then
                if(reset='1') then
                   r(0 to 159)<=(others=>'0');
                   s(0 to 159)<=(others=>'0');

                 else
                   r<=rf;
                   s<=sf;
                   if(cifrar='1') then
                      zi<= r(0) xor s(0);
                   end if;
                end if;
             end if;
          end process;
   ri<=r;
   si<=s;
   end architecture;
```

Componente compo_clock_kg

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity compo_clockkg is
```

```
        port (
             r: in std_logic_vector(0 to 159);
             s: in std_logic_vector(0 to 159);
             mixing: in std_logic;
             inputbit: in std_logic;
             rfinal: out std_logic_vector(0 to 159);
             sfinal: out std_logic_vector(0 to 159)
            );end compo_clockkg;

architecture comclock_kg of compo_clockkg is
component compo_inputbit is
          port ( mixing: in std_logic;
                input_bit1: in std_logic;
                s80: in std_logic;
                input_bit: out std_logic
               );end component;
component compo_clockr is
          port ( r: in std_logic_vector(0 to 159);
              input_bit: in std_logic;
                control_bit: in std_logic;
                rfinal: out std_logic_vector(0 to 159)
               );end component;
component compo_clocks is
          port ( s: in std_logic_vector(0 to 159);
                input_bit: in std_logic;
                control_bit: in std_logic;
                sfinal: out std_logic_vector(0 to 159)
               );end component;

signal control_bit_r: std_logic;
signal control_bit_s: std_logic;
signal input: std_logic;

begin
   control_bit_r<=s(54)xor r(106);
   control_bit_s<=s(106)xor r(53);
   icompoinputbit: compo_inputbit port map
                     ( mixing => mixing,
                       input_bit1 => inputbit,
                       s80 => s(80),
                       input_bit => input);
   icompoclockr: compo_clockr port map
                   ( r => r,
                   input_bit => input,
              control_bit => control_bit_r,
```

```
                        rfinal => rfinal);
   icompoclocks: compo_clocks port map
                      ( s => s,
                        input_bit => inputbit,
                        control_bit => control_bit_s,
                        sfinal => sfinal);
end architecture;
```

Componente compo_inputbit

```
library IEEE;
use IEEE.std_logic_1164.all;
entity compo_inputbit is
        port (
             mixing: in std_logic;
             input_bit1: in std_logic;
             s80: in std_logic;
             input_bit: out std_logic
            );end compo_inputbit;

architecture compinputbit of compo_inputbit is
   begin
      input_bit<=input_bit1 xor s80 when mixing='1' else
               input_bit1;
   end architecture;
```

Componente compo_clockr

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity compo_clockr is
        port (
             r: in std_logic_vector(0 to 159);
             input_bit: in std_logic;
             control_bit: in std_logic;
             rfinal: out std_logic_vector(0 to 159)
            );end compo_clockr;

architecture comclock_r of compo_clockr is

   component compo_rprim is
             port (
                   rp: in std_logic_vector(0 to 159);
```

```
                        r: in std_logic_vector(0 to 159);
                        control_bit: in std_logic;
                        rfinal: out std_logic_vector(0 to 159)
                       );end component;
            component compo_rtaps is
                 port (
                        ri: in std_logic_vector(0 to 159);
                        feedback: in std_logic;
                        rp: out std_logic_vector(0 to 159)
                       );end component;

    signal feedback: std_logic;
    signal ri: std_logic_vector(0 to 159);
    signal rp: std_logic_vector(0 to 159);
begin
    ri(0 to 159)<='0' & r(0 to 158);
    feedback<=r(159)xor input_bit;

    irtaps: compo_rtaps port map
                   (ri => ri,
                    feedback => feedback,
                    rp => rp);
            irprim: compo_rprim port map
                   (rp => rp,
                    r => r,
                    control_bit => control_bit,
                    rfinal => rfinal);
end architecture;
```

Componente compo_rtaps

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity compo_rtaps is
           port (
                 ri: in std_logic_vector(0 to 159);
                 feedback: in std_logic;
                 rp: out std_logic_vector(0 to 159)
                );end compo_rtaps;

architecture compo1 of compo_rtaps is
begin
    rp(0)<= ri(0)xor feedback; rp(1)<=ri(1);
```

```
rp(2)<= ri(2); rp(3)<=ri(3);
rp(4)<= ri(4)xor feedback; rp(5)<=ri(5)xor feedback;
rp(6)<= ri(6); rp(7)<=ri(7);
rp(8)<= ri(8)xor feedback; rp(9)<=ri(9);
rp(10)<= ri(10)xor feedback; rp(11)<=ri(11)xor feedback;
rp(12)<= ri(12); rp(13)<=ri(13);
rp(14)<= ri(14)xor feedback; rp(15)<=ri(15);
rp(16)<= ri(16)xor feedback; rp(17)<=ri(17);
rp(18)<= ri(18); rp(19)<=ri(19);
rp(20)<= ri(20)xor feedback; rp(21)<=ri(21);
rp(22)<= ri(22); rp(23)<=ri(23);
rp(24)<= ri(24); rp(25)<=ri(25)xor feedback;
rp(26)<= ri(26); rp(27)<=ri(27);
rp(28)<= ri(28); rp(29)<=ri(29);
rp(30)<= ri(30)xor feedback; rp(31)<=ri(31);
rp(32)<= ri(32)xor feedback; rp(33)<=ri(33);
rp(34)<= ri(34); rp(35)<=ri(35)xor feedback;
rp(36)<= ri(36)xor feedback; rp(37)<=ri(37);
rp(38)<= ri(38)xor feedback; rp(39)<=ri(39);
rp(40)<= ri(40); rp(41)<=ri(41);
rp(42)<= ri(42)xor feedback; rp(43)<=ri(43)xor feedback;
rp(44)<= ri(44); rp(45)<=ri(45);
rp(46)<= ri(46)xor feedback; rp(47)<=ri(47);
rp(48)<= ri(48); rp(49)<=ri(49);
rp(50)<= ri(50)xor feedback; rp(51)<=ri(51)xor feedback;
rp(52)<= ri(52); rp(53)<=ri(53)xor feedback;
rp(54)<= ri(54)xor feedback; rp(55)<=ri(55)xor feedback;
rp(56)<= ri(56)xor feedback; rp(57)<=ri(57)xor feedback;
rp(58)<= ri(58); rp(59)<=ri(59);
rp(60)<= ri(60)xor feedback; rp(61)<=ri(61)xor feedback;
rp(62)<= ri(62)xor feedback; rp(63)<=ri(63)xor feedback;
rp(64)<= ri(64); rp(65)<=ri(65)xor feedback;
rp(66)<= ri(66)xor feedback; rp(67)<=ri(67);
rp(68)<= ri(68); rp(69)<=ri(69)xor feedback;
rp(70)<= ri(70); rp(71)<=ri(71);
rp(72)<= ri(72); rp(73)<=ri(73)xor feedback;
rp(74)<= ri(74)xor feedback; rp(75)<=ri(75);
rp(76)<= ri(76)xor feedback; rp(77)<=ri(77);
rp(78)<= ri(78); rp(79)<=ri(79)xor feedback;
rp(80)<= ri(80)xor feedback; rp(81)<=ri(81)xor feedback;
rp(82)<= ri(82)xor feedback; rp(83)<=ri(83);
rp(84)<= ri(84); rp(85)<=ri(85)xor feedback;
rp(86)<= ri(86)xor feedback; rp(87)<=ri(87);
rp(88)<= ri(88); rp(89)<=ri(89);
rp(90)<= ri(90)xor feedback; rp(91)<=ri(91)xor feedback;
```

```
    rp(92)<= ri(92)xor feedback; rp(93)<=ri(93);
    rp(94)<= ri(94); rp(95)<=ri(95)xor feedback;
    rp(96)<= ri(96); rp(97)<=ri(97)xor feedback;
    rp(98)<= ri(98); rp(99)<=ri(99);
    rp(100)<= ri(100)xor feedback; rp(101)<=ri(101)xor feedback;
    rp(102)<= ri(102); rp(103)<=ri(103);
    rp(104)<= ri(104); rp(105)<=ri(105)xor feedback;
    rp(106)<= ri(106)xor feedback; rp(107)<=ri(107)xor feedback;
    rp(108)<= ri(108)xor feedback; rp(109)<=ri(109)xor feedback;
    rp(110)<= ri(110); rp(111)<=ri(111)xor feedback;
    rp(112)<= ri(112)xor feedback; rp(113)<=ri(113)xor feedback;
    rp(114)<= ri(114); rp(115)<=ri(115)xor feedback;
    rp(116)<= ri(116)xor feedback; rp(117)<=ri(117)xor feedback;
    rp(118)<= ri(118); rp(119)<=ri(119);
    rp(120)<= ri(120); rp(121)<=ri(121);
    rp(122)<= ri(122); rp(123)<=ri(123);
    rp(124)<= ri(124); rp(125)<=ri(125);
    rp(126)<= ri(126); rp(127)<=ri(127)xor feedback;
    rp(128)<= ri(128)xor feedback; rp(129)<= ri(129)xor feedback;
    rp(130)<= ri(130)xor feedback; rp(131)<= ri(131)xor feedback;
    rp(132)<= ri(132); rp(133)<= ri(133)xor feedback;
    rp(134)<= ri(134); rp(135)<= ri(135)xor feedback;
    rp(136)<= ri(136)xor feedback; rp(137)<= ri(137)xor feedback;
    rp(138)<= ri(138); rp(139)<= ri(139);
    rp(140)<= ri(140)xor feedback; rp(141)<= ri(141);
    rp(142)<= ri(142)xor feedback; rp(143)<= ri(143);
    rp(144)<= ri(144); rp(145)<= ri(145)xor feedback;
    rp(146)<= ri(146); rp(147)<= ri(147);
    rp(148)<= ri(148)xor feedback; rp(149)<= ri(149);
    rp(150)<= ri(150)xor feedback; rp(151)<= ri(151);
    rp(152)<= ri(152)xor feedback; rp(153)<= ri(153)xor feedback;
    rp(154)<= ri(154)xor feedback; rp(155)<= ri(155);
    rp(156)<= ri(156)xor feedback; rp(157)<= ri(157)xor feedback;
    rp(158)<= ri(158); rp(159)<= ri(159);

end architecture;
```

Componente compo_rprim

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity compo_rprim is
        port (
```

```
                rp: in std_logic_vector(0 to 159);
                r: in std_logic_vector(0 to 159);
                control_bit: in std_logic;
                rfinal: out std_logic_vector(0 to 159)
               );end compo_rprim;

architecture comp2 of compo_rprim is
    begin
        rfinal<=rp when control_bit='0' else
                rp xor r;
    end architecture;
```

Componente compo_clocks

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity compo_clocks is
          port (
                s: in std_logic_vector(0 to 159);
                input_bit: in std_logic;
                control_bit: in std_logic;
                sfinal: out std_logic_vector(0 to 159)
               );end compo_clocks;

architecture comclock_s of compo_clocks is
   component compo_sp is
               port ( sg: in std_logic_vector(0 to 159);
                      feedback: in std_logic;
                      control_bit: in std_logic;
                      si: out std_logic_vector(0 to 159)
                     );end component;

   component compo_sg is
               port ( s: in std_logic_vector(0 to 159);
                      sg: out std_logic_vector(0 to 159)
                     );end component;

          signal feedback: std_logic;
          signal sg: std_logic_vector(0 to 159);
begin
    feedback<= s(159)xor input_bit;
    icompo_sg: compo_sg port map
                      ( s => s,
```

```
                    sg =>sg);

   icompo_sp: compo_sp port map
                  ( sg=>sg,
                    feedback=>feedback,
                    control_bit=> control_bit,
                    si=>sfinal);
end architecture;
```

Componente compo_sg

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity compo_sg is
        port (
             s: in std_logic_vector(0 to 159);
             sg: out std_logic_vector(0 to 159)
            );end compo_sg;

architecture comsg of compo_sg is

   constant comp1: std_logic_vector(0 to
158):="10001100111110001001100010111110000110010011110001101101
01111111000001111100001100000000000111110101000101100011100000
1100110011010101101110110100010111111111111";
         constant comp0: std_logic_vector(0 to 158):=
"11111010010011110110101110111010101010101001000001100100100 11
11001000110000011100000000010011110100011001001101111111010111
1011000111110101100000011111011111000";
   begin
     sg(1 to 158)<=s(0 to 157) xor ((s(1 to 158)xor comp0(1 to 158))and(s(2 to
159)xor comp1(1 to 158)));
     sg(0)<='0'; sg(159)<=s(158);
   end architecture;
```

Componente compo_sp

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity compo_sp is
        port (
```

```
            sg: in std_logic_vector(0 to 159);
            feedback: in std_logic;
            control_bit: in std_logic;
            si: out std_logic_vector(0 to 159)
           );end compo_sp;

architecture comsp of compo_sp is
   signal fb0: std_logic_vector(0 to 159);
   signal fb1: std_logic_vector(0 to 159);
   signal feedback1: std_logic_vector(0 to 159);
   begin
     fb0(0 to 159)<=
"1111010111111000001111000010001101000100110001011111010001110
0001000000110110010101001110110011010001001110100100010101000
1010111000001111010000110001101100000 1";
     fb1(0 to 159)<=
"1101010111101110001011111101100100001001001100011001111000001
1100110110100011000010110011111011011100111011111101101001000
1101101111011100000000111100101100010 00";
     feedback1(0 to 159)<=(others=>feedback);
     si(0 to 159)<= (sg(0 to 159)xor (fb0(0 to 159) and feedback1(0 to
159)))when control_bit='0' else
                   (sg(0 to 159)xor (fb1(0 to 159) and feedback1(0 to 159)));
   end architecture;
```

Componente bloque_inputbit

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity bloque_inputbit is
  Generic(niv:integer:=128);
        port (
       clk: in std_logic;
            reset: in std_logic;
            iv: in std_logic_vector(0 to niv-1);
            k: in std_logic_vector(0 to 127);
            enable: integer;
            contador: integer;
            input_bit: out std_logic
           );end bloque_inputbit;

architecture bloqueinputbit of bloque_inputbit is
   begin
```

```
    process(clk, reset, enable, contador, iv, k)
      begin
        if((clk'event) and (clk='1')) then
          if(reset='1') then
            input_bit<=iv(0);
           elsif(enable=0)then
            input_bit<= iv(contador);
            elsif(enable=1)then
             input_bit<=k(contador);
             elsif(enable=2)then
              input_bit<='0';
          end if;
        end if;
      end process;
  end architecture;
```

Componente control

```
-- VHDL Entity mickey_lib.control.interface
--
-- Created:
--          by - HP_Propietario.UNKNOWN (YAEL)
--          at - 23:42:08 23/11/2006
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2004.1 (Build 41)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY control IS
  Generic(niv:integer:=128);
  PORT(
    clk      : IN     std_logic;
    ini      : IN     std_logic;
    n        : IN     integer;
    cifrar   : OUT    std_logic;
    contador : OUT    integer;
    enable   : OUT    integer;
    mixing   : OUT    std_logic;
    reset    : OUT    std_logic
  );

-- Declarations
```

```
END control ;

--
-- VHDL Architecture mickey_lib.control.fsm
--
-- Created:
--          by - HP_Propietario.UNKNOWN (YAEL)
--          at - 23:42:08 23/11/2006
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2004.1 (Build 41)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ARCHITECTURE fsm OF control IS

  -- Architecture Declarations
  signal aux: integer;

  TYPE STATE_TYPE IS (
    inicio,
    S_reset,
    S_loadiv,
    S_loadk,
    S_preclock,
    S_cifrar
  );

  -- State vector declaration
  ATTRIBUTE state_vector : string;
  ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state" ;


  -- Declare current and next state signals
  SIGNAL current_state : STATE_TYPE ;
  SIGNAL next_state : STATE_TYPE ;

BEGIN

  ----------------------------------------------------------------------------
  clocked : PROCESS(
    clk
  )
  ----------------------------------------------------------------------------
```

```
BEGIN
  IF (clk'EVENT AND clk = '1') THEN
    current_state <= next_state;
    -- Default Assignment To Internals
    aux <= 0;

    -- Combined Actions for internal signals only
    CASE current_state IS
    WHEN inicio =>
      aux<=1;
    WHEN S_loadiv =>
      aux<= aux+1;
      IF (aux=niv) THEN
        aux<=0;
      END IF;
    WHEN S_loadk =>
      aux<= aux+1;
      IF (aux=128) THEN
        aux<=0;
      END IF;
    WHEN S_preclock =>
      aux<= aux+1;
      IF (aux=160) THEN
        aux<=0;
      END IF;
    WHEN S_cifrar =>
      aux<= aux+1;
    WHEN OTHERS =>
      NULL;
    END CASE;


  END IF;

END PROCESS clocked;

-----------------------------------------------------------------
nextstate : PROCESS (
  aux,
  current_state,
  ini
)
-----------------------------------------------------------------
BEGIN
  CASE current_state IS
```

```
    WHEN inicio =>
      IF (ini='1') THEN
        next_state <= S_reset;
      ELSE
        next_state <= inicio;
      END IF;
    WHEN S_reset =>
        next_state <= S_loadiv;
    WHEN S_loadiv =>
      IF (aux=niv) THEN
        next_state <= S_loadk;
      ELSE
        next_state <= S_loadiv;
      END IF;
    WHEN S_loadk =>
      IF (aux=128) THEN
        next_state <= S_preclock;
      ELSE
        next_state <= S_loadk;
      END IF;
    WHEN S_preclock =>
      IF (aux=160) THEN
        next_state <= S_cifrar;
      ELSE
        next_state <= S_preclock;
      END IF;
    WHEN S_cifrar =>
      IF (aux=128) THEN
        next_state <= inicio;
      ELSE
        next_state <= S_cifrar;
      END IF;
    WHEN OTHERS =>
      next_state <= inicio;
    END CASE;

  END PROCESS nextstate;

  ------------------------------------------------------------------
  output : PROCESS (
    aux,
    current_state
  )
  ------------------------------------------------------------------
  BEGIN
```

```
-- Default Assignment
cifrar <= '0';
contador <= 0;
enable <= 0;
mixing <= '0';
reset <= '0';
-- Default Assignment To Internals

-- Combined Actions
CASE current_state IS
WHEN inicio =>
   reset <= '0' ;
  cifrar <= '0' ;
  enable <= 0;
  contador <= 1;
  mixing <= '0' ;
WHEN S_reset =>
  reset<='1';
WHEN S_loadiv =>
  reset<='0';
  mixing<='1';
  contador<= aux;
  IF (aux=niv) THEN
    contador<=0;
  END IF;
WHEN S_loadk =>
  enable<=1;
  contador<= aux;
  IF (aux=128) THEN
    contador<=0;
  END IF;
WHEN S_preclock =>
  enable<=2;
  contador<= aux;
  IF (aux=160) THEN
    contador<=0;
  END IF;
WHEN S_cifrar =>
  mixing<='0';
  contador<= aux;
  cifrar<='1';
WHEN OTHERS =>
  NULL;
END CASE;
```

```
  END PROCESS output;

  -- Concurrent Statements

END fsm;
```

Apéndice C. Trivium

En este apéndice se muestra la arquitectura del algoritmo, la explicación componente a componente de la arquitectura y el código VHDL de cada uno de los componentes.

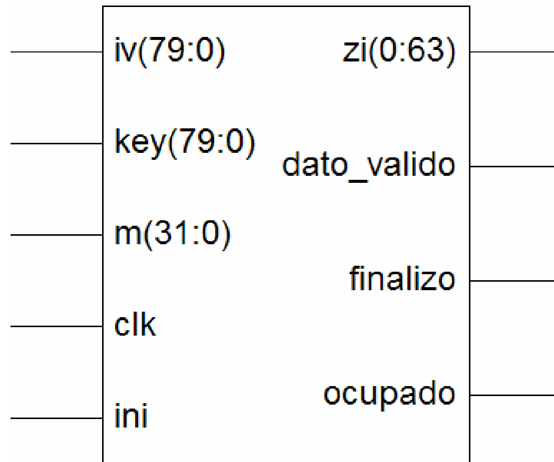

Fig. C.1 Componente Trivium generado por la herramienta Xilinx ISE 8.2i

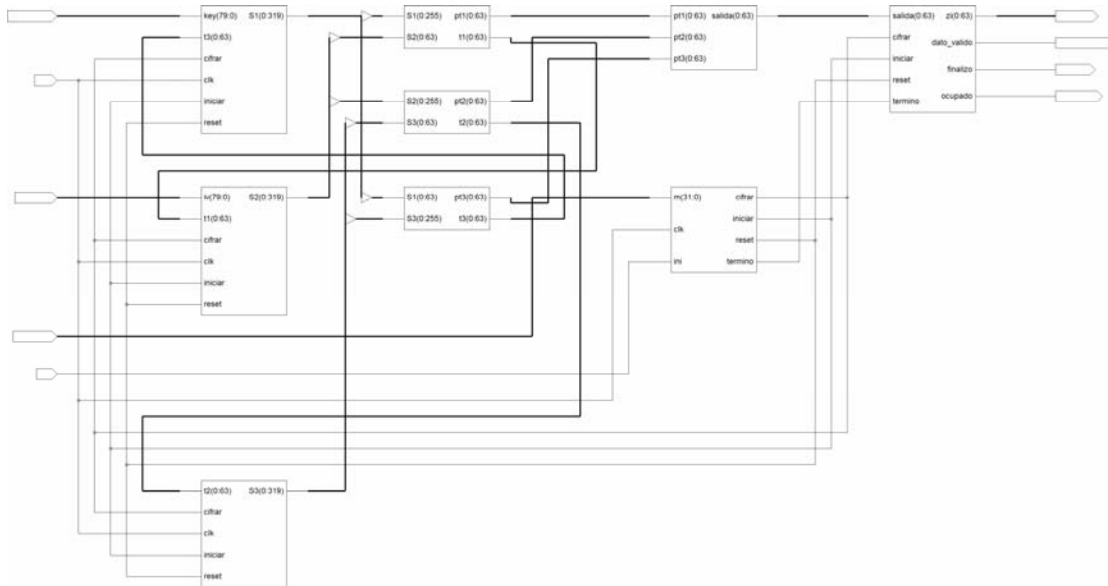

Fig. C.2 Diagrama RTL de Trivium generado por la herramienta Xilinx ISE 8.2i

Nombre de componente	Componentes Internos	Señales de Entrada	Señales de Salida	Descripción del componente
Cif_Triviumgenerico	▪ genericontrol_Trivium ▪ regs1_s93 ▪ regs94_s177 ▪ regs178_s288 ▪ t1_1 ▪ t2_2 ▪ t3_3 ▪ funzi	▪ clk ▪ key ▪ iv ▪ m ▪ ini	▪ zi ▪ ocupado ▪ dato_valido ▪ finalizo	Es el componente principal que contiene a los demás componentes, es aquí donde van a entrar las señales principales y de la cual van a salir las señales que requiera el usuario para implementar y saber cuando el bit del keystream es factible de usar.
genericontrol_trivium		▪ clk ▪ ini ▪ m	▪ cifrar ▪ iniciar ▪ reset ▪ termino	Componente en el que se encuentra el algoritmo de control de la arquitectura para poder pasar de la etapa de inicialización a la etapa de generación del keystream.
regs1_s93		▪ clk ▪ reset ▪ iniciar ▪ cifrar ▪ key ▪ t3	▪ S1	Componente que define las operaciones a realizar en la primera parte en la que fue dividido el registro principal, además se va actualizando esa parte del vector y envia los bits de las posiciones requeridas para obtener el nuevo bit que ocupará la posición más significativa de esa parte del registro.

regs94_s177		▪ clk ▪ reset ▪ iniciar ▪ cifrar ▪ iv ▪ t1	▪ S2	Componente que contiene la segunda parte del registro en la cual se actualiza dicha parte y se envían los bits requerido para calcular el nuevo valor del bit que se colocara en la parte más significativa de esta parte del registro.
regs178_s288		▪ clk ▪ reset ▪ iniciar ▪ cifrar ▪ t2	▪ S3	En este componente se encuentra la tercera parte en la que se dividió el registro principal, en el que se actualiza el registro y son enviado solo s bits necesarios para obtener el valor del nuevo bit que se ubicara en la posición más significativa.
t1_1		▪ S1 ▪ S2	▪ pt1 ▪ t1	Componente en se efectúa la obtención del bit que se usara para calcular junto con los otros valores de las t2_2 y de t3_3 para obtener el bit del keystream. Y que además se usara para actualizar el valor del bit de la posición más significativa de regs94_s177.
t2_2		▪ S2 ▪ S3	▪ pt2 ▪ t2	Componente en el se obtiene el valor del bit necesario para calcular keystream, y que además se usa para actualizar la posición más significativa de regs178_s288.

t3_3		▪ S3 ▪ S1	▪ pt3 ▪ t3	Componente en el que se calcula el ultimo valor requerido para calcular el bit del keystream, y que funcionara para actualizar la parte más significativa de regs1_s93.
funzi		▪ pt1 ▪ pt2 ▪ pt3	▪ salida	En este componente se calcula mediante la función xor de entre los bits resultantes de t1_1, t2_2 y t3_3 que formara parte del keystream.

Código de los componentes.

Componente principal. Cifradorgenerico

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity Cif_Triviumgenerico is
    Generic(n:integer:=64);
 port(
     clk: in std_logic;--señal de reloj
           key: in std_logic_vector(79 downto 0);--la llave a utilizar en el cifrado
                 iv: in std_logic_vector(79 downto 0);--valor del vector de
inicialización
                 m: in integer;-- longitud de keystream
                 ini: in std_logic;
                 zi: out std_logic_vector(0 to n-1);--bit del keystream
                 ocupado: out std_logic;
                 dato_valido: out std_logic;
                 finalizo: out std_logic
                 );
end Cif_Triviumgenerico;

architecture Cifradorgenerico of Cif_Triviumgenerico is

component genericontrol_trivium IS
   Generic(n:integer:=1);
   PORT(
      clk     : IN     std_logic;
      ini     : IN     std_logic;
      m       : IN     integer;
      cifrar  : OUT    std_logic;
      iniciar : OUT    std_logic;
      reset   : OUT    std_logic;
      termino : OUT    std_logic
   );
end component ;

component registro_es is
   Generic(n: integer);
              port (
                   reset: in std_logic;--asigna valor al registro
                  iniciar: in std_logic;--realiza los ciclos para ajustar registros
                         cifrar: in std_logic;--para iniciar el cifrado
```

```
                    termino: in std_logic;
                                        salida: in std_logic_vector(0 to n-1);
                                        ocupado: out std_logic;
                                        dato_valido: out std_logic;
                                        finalizo: out std_logic;
                        zi: out std_logic_vector(0 to n-1)
                        );
        end component;
component regs1_s93 is--0_92
  Generic(n: integer);
        port(
          clk: in std_logic;--señal de reloj
          reset: in std_logic;--señal de reinicio
                iniciar: in std_logic;--señal de inicializacion del cifrador
                cifrar: in std_logic;--señal de generacion de cifrado
                key: in std_logic_vector(79 downto 0);--la llave a utilizar en el
cifrado
                t3: in std_logic_vector(0 to n-1);-- t3 primas para recalcular bits
                S1: out std_logic_vector(0 to 5*n-1)--bits s66, s93, s171, s91,
s92(65,92,170,90,91)
        );
end component;

component regs94_s177 is
  Generic(n: integer);
        port(
          clk: in std_logic;--señal de reloj
          reset: in std_logic;--señal de reinicio
                iniciar: in std_logic;--señal de inicializacion del cifrador
                cifrar: in std_logic;--señal de generacion de cifrado
                iv: in std_logic_vector(79 downto 0);--valor del vector de
inicialización
                t1: in std_logic_vector(0 to n-1);-- t`s primas para recalcular bits
                S2: out std_logic_vector(0 to 5*n-1)--bit del keystream
        );
end component;

component regs178_s288 is
  Generic(n: integer);
        port(
          clk: in std_logic;--señal de reloj
          reset: in std_logic;--señal de reinicio
                iniciar: in std_logic;--señal de inicializacion del cifrador
                cifrar: in std_logic;--señal de generacion de cifrado
                t2: in std_logic_vector(0 to n-1);-- t`s primas para recalcular bits
```

```
              S3: out std_logic_vector(0 to 5*n-1)--bit del keystream
       );
end component;

component t1_1 is
  Generic(n: integer);
       port(
         S1: in std_logic_vector(0 to 4*n-1);
         S2: in std_logic_vector(0 to n-1);
         pt1: out std_logic_vector(0 to n-1);
         t1: out std_logic_vector(0 to n-1)
       );
end component;

component t2_2 is
  Generic(n: integer:=1);
       port(
         S2: in std_logic_vector(0 to 4*n-1);
         S3: in std_logic_vector(0 to n-1);
         pt2: out std_logic_vector(0 to n-1);
         t2: out std_logic_vector(0 to n-1)
       );
end component;

component t3_3 is
  Generic(n: integer:=1);
       port(
         S3: in std_logic_vector(0 to 4*n-1);
         S1: in std_logic_vector(0 to n-1);
         pt3: out std_logic_vector(0 to n-1);
         t3: out std_logic_vector(0 to n-1)
       );
end component;

component funzi is
   Generic(n: integer:=1);
        port(
            pt1: in std_logic_vector(0 to n-1);
                        pt2: in std_logic_vector(0 to n-1);
                        pt3: in std_logic_vector(0 to n-1);
                        salida: out std_logic_vector(0 to n-1)
            );
end component;
--señales para conectar componentes
signal cifrar: std_logic;
```

```
signal iniciar: std_logic;
signal reset: std_logic;
signal termino: std_logic;
signal t3: std_logic_vector(0 to n-1);
signal S1: std_logic_vector(0 to 5*n-1);
signal t1: std_logic_vector(0 to n-1);
signal S2: std_logic_vector(0 to 5*n-1);
signal t2: std_logic_vector(0 to n-1);
signal S3: std_logic_vector(0 to 5*n-1);
signal pt1: std_logic_vector(0 to n-1);
signal pt2: std_logic_vector(0 to n-1);
signal pt3: std_logic_vector(0 to n-1);
signal salida: std_logic_vector(0 to n-1);

begin
icontrol: genericontrol_trivium Generic map(n=>n)
   port map
   (clk => clk,
    ini => ini,
    m => m,
    cifrar => cifrar,
    iniciar => iniciar,
    reset => reset,
    termino => termino);
ireg_es: registro_es Generic map(n=>n)
   port map
   (reset => reset,
    iniciar => iniciar,
           cifrar => cifrar,
    termino => termino,
           salida => salida,
           ocupado => ocupado,
           dato_valido => dato_valido,
           finalizo => finalizo,
           zi => zi);
i1_93: regs1_s93 Generic map(n=>n)
          port map
          ( clk => clk,
            reset => reset,
            iniciar => iniciar,
            cifrar => cifrar,
            key => key,
            t3 => t3,
            S1 => S1 );
i94_177: regs94_s177 Generic map(n=>n)
```

```
          port map
          (  clk => clk,
             reset => reset,
                    iniciar => iniciar,
                    cifrar => cifrar,
                    iv => iv,
                    t1 => t1,
                    S2 => S2);
i178_288: regs178_s288 Generic map(n=>n)
          port map
          (  clk => clk,
             reset => reset,
                    iniciar => iniciar,
                    cifrar => cifrar,
                    t2 => t2,
                    S3 => S3);
it1_1: t1_1 Generic map(n =>n)
          port map
          (  S1 => S1(0 to 4*n-1),
             S2 => S2(4*n to 5*n-1),
             pt1 => pt1,
             t1 => t1);
it2_2: t2_2 Generic map(n=>n)
          port map
          (  S2 => S2(0 to 4*n-1),
             S3 => S3(4*n to 5*n-1),
             pt2 => pt2,
             t2 => t2);
it3_3: t3_3 Generic map(n=>n)
          port map
          (  S3 => S3(0 to 4*n-1),
             S1 => S1(4*n to 5*n-1),
             pt3 => pt3,
             t3 => t3);
izi: funzi Generic map(n=>n)
           port map
           ( pt1 => pt1,
                    pt2 => pt2,
                    pt3 => pt3,
                    salida => salida);

end;
```

Componente genericontrol_Trivium

```
-- VHDL Entity controlgenerico_trivium_lib.genericontrol_trivium.interface
--
-- Created:
--          by - HP_Propietario.UNKNOWN (YAEL)
--          at - 11:43:37 19/03/2007
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2004.1 (Build 41)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY genericontrol_trivium IS
    Generic(n: integer:=1);
   PORT(
      clk     : IN     std_logic;
      ini     : IN     std_logic;
      m       : integer;
      cifrar  : OUT    std_logic;
      iniciar : OUT    std_logic;
      reset   : OUT    std_logic;
      termino : OUT    std_logic
   );

-- Declarations

END genericontrol_trivium ;

--
-- VHDL Architecture
controlgenerico_trivium_lib.genericontrol_trivium.controltrivium_genrico
--
-- Created:
--          by - HP_Propietario.UNKNOWN (YAEL)
--          at - 11:43:37 19/03/2007
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2004.1 (Build 41)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ARCHITECTURE controltrivium_genrico OF genericontrol_trivium IS

   -- Architecture Declarations
```

```
   signal contador: integer;

   TYPE STATE_TYPE IS (
      S_inicio,
      S_reset,
      S_iniciar,
      S_cifrar
   );

   -- State vector declaration
   ATTRIBUTE state_vector : string;
   ATTRIBUTE state_vector OF controltrivium_genrico : ARCHITECTURE IS
"current_state" ;


   -- Declare current and next state signals
   SIGNAL current_state : STATE_TYPE ;
   SIGNAL next_state : STATE_TYPE ;

BEGIN

   -----------------------------------------------------------------
   clocked : PROCESS(
      clk
   )
   -----------------------------------------------------------------
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         current_state <= next_state;
         -- Default Assignment To Internals
         contador <= 0;

         -- Combined Actions for internal signals only
         CASE current_state IS
         WHEN S_inicio =>
            contador<=1;
         WHEN S_iniciar =>
            contador<= contador +1;
            IF (contador = (1152-n)/n) THEN
               contador<=0;
            END IF;
         WHEN S_cifrar =>
            contador <= contador +1;
         WHEN OTHERS =>
            NULL;
```

```
      END CASE;


    END IF;

  END PROCESS clocked;

  ---------------------------------------------------------------
  nextstate : PROCESS (
    contador,
    current_state,
    ini,
    m
  )
  ---------------------------------------------------------------
  BEGIN
    CASE current_state IS
    WHEN S_inicio =>
      IF (ini = '1' ) THEN
        next_state <= S_reset;
      ELSE
        next_state <= S_inicio;
      END IF;
    WHEN S_reset =>
        next_state <= S_iniciar;
    WHEN S_iniciar =>
      IF (contador = (1152-n)/n) THEN
        next_state <= S_cifrar;
      ELSE
        next_state <= S_iniciar;
      END IF;
    WHEN S_cifrar =>
      IF ( contador = m/n) THEN
        next_state <= S_inicio;
      ELSE
        next_state <= S_cifrar;
      END IF;
    WHEN OTHERS =>
      next_state <= S_inicio;
    END CASE;

  END PROCESS nextstate;

  ---------------------------------------------------------------
  output : PROCESS (
```

```
   contador,
   current_state,
   m
)
----------------------------------------------------------------
BEGIN
   -- Default Assignment
   cifrar <= '0';
   iniciar <= '0';
   reset <= '0';
   termino <= '0';
   -- Default Assignment To Internals

   -- Combined Actions
   CASE current_state IS
   WHEN S_inicio =>
      reset<='0';
      iniciar<='0';
      cifrar<='0';
      termino<='0';
   WHEN S_reset =>
       reset <= '1' ;
      iniciar <= '0' ;
      cifrar <= '0' ;
      termino<='0';
   WHEN S_iniciar =>
       reset <= '0' ;
      iniciar <= '1' ;
      cifrar <= '0' ;
      termino<='0';
   WHEN S_cifrar =>
       reset <= '0' ;
      iniciar <= '0';
       cifrar <= '1' ;
      termino<='0';
      IF ( contador = m/n) THEN
         cifrar<='0';
         termino<='1';
      END IF;
   WHEN OTHERS =>
      NULL;
   END CASE;


END PROCESS output;
```

```
  -- Concurrent Statements

END controltrivium_genrico;
```

Componente registro_es

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity registro_es is
  Generic(n: integer);
          port (
              reset: in std_logic;--asigna valor al registro
             iniciar: in std_logic;--realiza los ciclos para ajustar registros
                   cifrar: in std_logic;--para iniciar el cifrado
             termino: in std_logic;
                                 salida: in std_logic_vector(0 to n-1);
                                 ocupado: out std_logic;
                                 dato_valido: out std_logic;
                                 finalizo: out std_logic;
                 zi: out std_logic_vector(0 to n-1)
                 );
       end registro_es;
architecture reg_es of registro_es is
begin

   process(reset, iniciar, cifrar, termino, salida)
       begin
          if(reset='1' or iniciar='1') then
            ocupado<='1';
            dato_valido<='0';
            finalizo<='0';
            elsif(cifrar='1')then
               ocupado<='1';
               dato_valido<='1';
               finalizo<='0';
      elsif(termino='1') then
          ocupado<='0';
               dato_valido<='0';
               finalizo<='1';
          end if;
       end process;
       zi<= salida;
```

```
end architecture;
```

Componente regs1_s93

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity regs1_s93 is--0_92
   Generic(n: integer:=4);
        port(
           clk: in std_logic;--señal de reloj
           reset: in std_logic;--señal de reinicio
                 iniciar: in std_logic;--señal de inicializacion del cifrador
                 cifrar: in std_logic;--señal de generacion de cifrado
                 key: in std_logic_vector(79 downto 0);--la llave a utilizar en el
cifrado
                 t3: in std_logic_vector(0 to n-1);-- t3 primas para recalcular bits
                 S1: out std_logic_vector(0 to 5*n-1)--bits s66, s93, s171, s91,
s92(65,92,170,90,91)
        );
end regs1_s93;


architecture registro1_93 of regs1_s93 is
    signal r: std_logic_vector(0 to 92);
begin
   process(clk, reset, iniciar, cifrar, key, t3)
        begin
            if(clk'event)and(clk='1')then
             if(reset='1')then
                r<=key& "0000000000000";--inicializacion con key
               elsif (iniciar='1')or (cifrar='1') then
                --actualizacion de registros
                  r<=t3& r(0 to 92-n);
             end if;
           end if;
    end process;

    --datos a enviar
   fs1: for i in 0 to n-1 generate
      S1(i)<=r(65-i);
      S1(i+n)<=r(92-i);
      S1(i+2*n)<=r(90-i);
      S1(i+3*n)<=r(91-i);
```

```
      S1(i+4*n)<=r(68-i);
   end generate fs1;
    -- S1<=r(65-n+1 to 65)& r(92-n+1 to 92)& r(90-n+1 to 90)& r(91-n+1 to
91)& r(68-n+1 to 68);
end architecture;
```

Componente regs94_s177

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity regs94_s177 is
   Generic(n: integer:=1);
          port(
             clk: in std_logic;--señal de reloj
             reset: in std_logic;--señal de reinicio
                    iniciar: in std_logic;--señal de inicializacion del cifrador
                    cifrar: in std_logic;--señal de generacion de cifrado
                    iv: in std_logic_vector(79 downto 0);--valor del vector de
inicialización
                    t1: in std_logic_vector(0 to n-1);-- t`s primas para recalcular bits
                    S2: out std_logic_vector(0 to 5*n-1)--bit del keystream
          );
end regs94_s177;


architecture registro94_177 of regs94_s177 is
    signal r: std_logic_vector(0 to 83);
begin
   inicializar: process(clk, reset, iniciar, cifrar, iv, t1)
        begin
            if(clk'event)and(clk='1')then
             if(reset='1')then
                 r<=iv& "0000";--inicializacion con vector de inicializacion
               elsif (iniciar='1')or (cifrar='1') then
                --actualizacion de registros
                  r<=t1& r(0 to 83-n);
             end if;
           end if;
    end process;

    --datos a enviar
    fs2: for i in 0 to n-1 generate
      S2(i)<=r(68-i);
```

```
      S2(i+n)<=r(83-i);
      S2(i+2*n)<=r(81-i);
      S2(i+3*n)<=r(82-i);
      S2(i+4*n)<=r(77-i);
   end generate fs2;
     --S2<=r(68 to 68-n+1)& r(83 to 83-n+1) & r(81 to 81-n+1)& r(82 to 82-n+1)
& r(77 to 77-n+1);
end architecture;
```

Componente regs178_s288

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity regs178_s288 is
   Generic(n: integer:=1);
          port(
             clk: in std_logic;--señal de reloj
             reset: in std_logic;--señal de reinicio
                    iniciar: in std_logic;--señal de inicializacion del cifrador
                    cifrar: in std_logic;--señal de generacion de cifrado
                    t2: in std_logic_vector(0 to n-1);-- t`s primas para recalcular bits
                    S3: out std_logic_vector(0 to 5*n-1)--bit del keystream
          );
end regs178_s288;


architecture registro178_288 of regs178_s288 is
    signal r: std_logic_vector(0 to 110);
begin
   inicializar: process(clk, reset, iniciar, cifrar, t2)
         begin
            if(clk'event)and(clk='1')then
             if(reset='1')then

r<="0000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000111";--
inicializacion con 0`s y "111"

                elsif (iniciar='1')or (cifrar='1') then
                 --actualizacion de registros
                   r<=t2& r(0 to 110-n);
             end if;
           end if;
```

```
    end process;

    --datos a enviar
    fs3: for i in 0 to n-1 generate
       S3(i)<=r(65-i);
       S3(i+n)<=r(110-i);
       S3(i+2*n)<=r(108-i);
       S3(i+3*n)<=r(109-i);
       S3(i+4*n)<=r(86-i);
   end generate fs3;
     --S3<=r(65 to 65-n+1)& r(110 to 110-n+1)& r(108 to 108-n+1)& r(109 to
109-n+1) & r(86 to 86-n+1);
end architecture;
```

Componente t1_1

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity t1_1 is
   Generic(n: integer:=1);
          port(
             S1: in std_logic_vector(0 to 4*n-1);
             S2: in std_logic_vector(0 to n-1);
             pt1: out std_logic_vector(0 to n-1);
             t1: out std_logic_vector(0 to n-1)
          );
end t1_1;

architecture primerat1 of t1_1 is
          signal t1interna: std_logic_vector(0 to n-1);
begin
      --- Se cálcula el valor de los t1
      t1interna<=S1(0 to n-1)xor S1(n to 2*n-1);--es la señal t1 q se usa para el
calculo de zi
      t1<=(t1interna xor S2)xor(S1(2*n to 3*n-1) and S1(3*n to 4*n-1));--se usa
para el calculo de estados en el reg.int
      pt1<=t1interna;
end architecture;
```

Componente t2_2

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity t2_2 is
  Generic(n: integer:=1);
        port(
          S2: in std_logic_vector(0 to 4*n-1);
          S3: in std_logic_vector(0 to n-1);
          pt2: out std_logic_vector(0 to n-1);
          t2: out std_logic_vector(0 to n-1)
        );
end t2_2;

architecture primerat2 of t2_2 is
        signal t2interna: std_logic_vector(0 to n-1);
begin
    --- Se cálcula el valor de los t2
    t2interna<=S2(0 to n-1)xor S2(n to 2*n-1);--es la señal t2 q se usa para el
calculo de zi
    t2<=(t2interna xor S3)xor(S2(2*n to 3*n-1) and S2(3*n to 4*n-1));--se usa
para el calculo de estados en el reg.int
    pt2<=t2interna;
end architecture;
```

Componente t3_3

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity t3_3 is
  Generic(n: integer:=1);
        port(
          S3: in std_logic_vector(0 to 4*n-1);
          S1: in std_logic_vector(0 to n-1);
          pt3: out std_logic_vector(0 to n-1);
          t3: out std_logic_vector(0 to n-1)
        );
end t3_3;

architecture primerat3 of t3_3 is
        signal t3interna: std_logic_vector(0 to n-1);
begin
    --- Se cálcula el valor de los t3
    t3interna<=S3(0 to n-1)xor S3(n to 2*n-1);--es la señal t3 q se usa para el
calculo de zi
```

```
     t3<=(t3interna xor S1)xor(S3(2*n to 3*n-1) and S3(3*n to 4*n-1));--se usa
para el calculo de estados en el reg.int
                  pt3<=t3interna;
end architecture;
```

Componente funzi

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity funzi is
   Generic(n: integer:=1);
         port(
              pt1: in std_logic_vector(0 to n-1);
                           pt2: in std_logic_vector(0 to n-1);
                           pt3: in std_logic_vector(0 to n-1);
                           salida: out std_logic_vector(0 to n-1)
             );
end funzi;

architecture funcionzi of funzi is

begin
   fzi: for i in 0 to n-1 generate
            salida(i)<=pt1(i) xor pt2(i) xor pt3(i);
     end generate fzi;
end funcionzi;
```