

**DISEÑO E IMPLEMENTACIÓN EN FPGA DEL ESTÁNDAR AES EN MODOS
DE OPERACIÓN NO REALIMENTADOS.**

IAN CARLO GUZMÁN VELÁSQUEZ

**UNIVERSIDAD DEL VALLE- FACULTAD DE INGENIERÍAS
ESCUELA DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA
PROGRAMA ACADÉMICO DE INGENIERÍA ELECTRÓNICA
AREA DE ARQUITECTURAS DIGITALES
SANTIAGO DE CALI
2013**

**DISEÑO E IMPLEMENTACIÓN EN FPGA DEL ESTÁNDAR AES EN MODOS
DE OPERACIÓN NO REALIMENTADOS.**

**Trabajo de grado presentado como requisito parcial para optar por el
título de Ingeniero Electrónico**

Estudiante:

IAN CARLO GUZMÁN VELÁSQUEZ

Director:

RUBÉN DARÍO NIETO LONDOÑO, Ph.D.

**UNIVERSIDAD DEL VALLE- FACULTAD DE INGENIERÍAS
ESCUELA DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA
PROGRAMA ACADÉMICO DE INGENIERÍA ELECTRÓNICA
AREA DE ARQUITECTURAS DIGITALES
SANTIAGO DE CALI
2013**

Nota de aceptación:

Ing. Vladimir Trujillo, Ms.C

Ing. Alexander vera

Santiago de Cali, agosto de 2013.

AGRADECIMIENTOS

Agradezco en primer lugar a Dios por darme la inteligencia y la sabiduría que me permitieron desarrollar y culminar este trabajo de grado. A mis padres por brindarme en todo momento su apoyo incondicional a lo largo de todo el proceso de mi formación académica. Al profesor Rubén Nieto por su incansable colaboración y disposición a lo largo del desarrollo de este trabajo. A los jurados evaluadores de este trabajo por aportar la experiencia y la objetividad con la cual evalúan este trabajo y finalmente a la Universidad del Valle por brindarme toda la formación académica profesional.

CONTENIDO

Lista de figuras	viii
Lista de tablas	xii
Introducción.....	xiv

CAPÍTULO 1: ESTANDAR DE ENCRIPCIÓN AVANZADO

1.1. Historia del algoritmo de <i>Rijndael</i>	16
1.2. Introducción al estándar <i>AES</i>	17
1.3. Proceso de cifrado (encriptación).....	20
1.3.1. Transformación de <i>SubBytes</i>	22
1.3.2. Transformación de <i>ShiftRows</i>	23
1.3.3. Transformación de <i>MixColumns</i>	24
1.3.4. Transformación <i>AddRoundKey</i>	25
1.3.5. Expansión de clave	25
1.4. Proceso de cifrado inverso (desencriptación).....	26
1.4.1. Transformación de <i>InvShiftRows</i>	28
1.4.2. Transformación de <i>InvSubBytes</i>	28
1.4.3. Transformación de <i>InvMixColumns</i>	29
1.4.4. Inversa de la transformación <i>AddRoundKey</i>	29

CAPÍTULO 2: MODOS DE OPERACIÓN

2.1. Introducción.....	30
2.2. Modo <i>ECB</i> (<i>Electronic Codebook</i>).....	30
2.3. Modo <i>CBC</i> (<i>Cipher Block Chaining</i>)	31
2.4. Modo <i>CFB</i> (<i>Cipher Feedback</i>).....	34
2.5. Modo <i>OFB</i> (<i>Output Feedback</i>)	36
2.6. Modo <i>CTR</i> (<i>Counter</i>)	38
2.6.1. Generación de los bloques contador.....	40
2.6.2. La función incremental estándar	41
2.6.3. Elección del bloque de contador inicial.....	42

CAPÍTULO 3: DISEÑO Y SIMULACIÓN

3.1. Introducción.....	43
3.2. Transformación <i>SubBytes</i>	43
3.2.1. Transformación <i>SubBytes</i> basada en tablas	43
3.2.2. Transformación <i>SubBytes</i> basada en el modelo matemático	44
3.3. Transformación <i>ShiftRows</i>	46
3.4. Transformación <i>MixColumns</i>	47
3.5. Transformación <i>AddRoundKey</i>	49
3.6. Transformación <i>InvSubBytes</i>	50
3.6.1. Transformación <i>InvSubBytes</i> basada en tablas	50
3.6.2. Transformación <i>InvSubBytes</i> basada en el modelo matemático....	51
3.7. Transformación <i>InvShiftRows</i>	52
3.8. Transformación <i>InvMixColumns</i>	52
3.9. Inversa de la transformación <i>AddRoundKey</i>	54
3.10. Unidad de SubClaves.....	54
3.10.1. Unidad de Subclaves iterativa	56
3.10.2. Unidad de Subclaves en cascada	59
3.11. Bloque Ronda.....	60
3.11.1. Bloque Ronda final	61
3.12. Bloque Ronda Inverso	61
3.12.1 Bloque Ronda final Inverso.....	62
3.13. Proceso de encriptación en modo de operación <i>ECB (Electronic Codebook)</i>	62
3.13.1. Sistema de encriptación en modo <i>ECB</i> para múltiples bloques ..	64
3.14. Proceso de desencriptación en modo de operación <i>ECB (Electronic Codebook)</i>	68
3.14.1. Sistema de desencriptación en modo <i>ECB</i> para múltiples bloques ..	69
3.15. Proceso de encriptación y desencriptación en modo de operación <i>CTR (Counter)</i>	72
3.15.1. Módulo de bloque de contador	73
3.15.2. Proceso de encriptación y desencriptación	74
3.15.3. Sistema de encriptación en modo <i>CTR</i> para múltiples bloques ..	76

3.15.4. Sistema de desencriptación en modo <i>CTR</i> para múltiples bloques	79
--	----

CAPÍTULO 4: RESULTADOS Y ANALISIS DE IMPLEMENTACIÓN

4.1. Transformaciones.....	82
4.2. Unidad de subclaves	83
4.3. Proceso de encriptación en modo <i>ECB</i>	84
4.3.1. Sistema de encriptación en modo <i>ECB</i> para múltiples bloques	85
4.4. Proceso de desencriptación en modo de operación <i>ECB</i>	88
4.4.1. Sistema de desencriptación en modo <i>ECB</i> para múltiples bloques.....	90
4.5. Proceso de encriptación en modo de operación <i>CTR (Counter)</i>	92
4.5.1. Sistema de encriptación en modo <i>CTR</i> para múltiples bloques	93
4.6. Proceso de desencriptación en modo de operación <i>CTR (Counter)</i>	96
4.7. Análisis de resultados de los modos de operación.....	98
4.7.1. Sistema de Encriptación y desencriptación en los modos <i>ECB</i> y <i>CTR</i> para múltiples bloques	99
4.8. Estado del arte en la implementación del algoritmo de <i>Rijndael</i> Para sistemas reprogramables.....	100

CAPÍTULO 5: CONCLUSIONES Y TRABAJO FUTURO

5.1. Conclusiones.....	106
5.2. Observaciones	107
5.3. Trabajo futuro	108
REFERENCIAS.....	109

LISTA DE FIGURAS

Figura 1.1: Matriz de Estado, S	18
Figura 1.2: La matriz de estado.....	18
Figura 1.3: Algoritmo para el proceso de encriptación	21
Figura 1.4: La transformación <i>SubBytes</i> aplica la tabla S-box a cada byte del estado	22
Figura 1.5: S-Box: valores de sustitución para el byte xy (en formato hexadecimal).....	23
Figura 1.6: La transformación <i>shiftrows</i> rota cíclicamente las últimas tres filas del estado.....	24
Figura 1.7: La transformación <i>MixColumns</i> opera en el estado columna por columna.....	25
Figura 1.8: La transformación <i>AddRounKey</i> realiza una XOR entre cada columna de la matriz de estado con una columna de la clave de ronda.....	25
Figura 1.9: Algoritmo para el proceso de desencriptación	27
Figura 1.10: La transformación <i>Invshiftrows</i> rota cíclicamente las últimas tres filas de la matriz de estado	28
Figura 1.11: S-Box inversa: valores de sustitución para <i>InvSubBytes</i> del byte xy (en formato hexadecimal).....	29
Figura 2.1: Modo <i>ECB</i> para los procesos de a) Encriptación y b) Desencriptación	31
Figura 2.2: Diagrama de bloques del proceso de encriptación y desencriptación en Modo <i>CBC</i>	33
Figura 2.3: Diagrama de bloques para los procesos de encriptación y desencriptación en Modo <i>CFB</i>	35
Figura 2.4: Diagrama de bloques para los procesos de encriptación y desencriptacion en Modo <i>OFB</i>	38
Figura 2.5: Diagrama de bloques para los procesos de encriptación y desencriptación en Modo <i>CTR</i>	40
Figura 3.1: Organización interna del módulo <i>SubBytes</i>	44
Figura 3.2: Simulación del módulo <i>SubBytes</i> basado en tablas.....	44

Figura 3.3: Diagrama de bloques de la transformación	
<i>SubBytes</i> basada en el modelo matemático para 8 bits.....	45
Figura 3.4: Simulación del módulo <i>SubBytes</i> para 8 bits basada	
en el modelo matemático	45
Figura 3.5: Diagrama de bloques para la transformación	
<i>SubBytes</i> para 16 bytes (128 bits).....	45
Figura 3.6: Simulación de la tranformación <i>SubBytes</i>	
basada en la operación matemática para 128 bits	46
Figura 3.7: Diagrama de circuito para realizar la transformación <i>ShiftRows</i> ..	46
Figura 3.8: Simulación del módulo <i>ShiftRows</i>	47
Figura 3.9: Diagrama de flujo para el cálculo de <i>xtime()</i>	47
Figura 3.10: Circuito que ejecuta la función <i>xtime()</i>	48
Figura 3.11: Diagrama de circuito de la transformación <i>MixColumns</i>	49
Figura 3.12: Simulación del módulo <i>MixColumns</i>	49
Figura 3.13: Transformación <i>AddRoundKey</i>	49
Figura 3.14: Simulación del módulo <i>AddRoundKey</i>	50
Figura 3.15: Organización circuital para el módulo <i>InvSubBytes</i>	50
Figura 3.16: Simulación del módulo <i>InvSubBytes</i>	50
Figura 3.17: Diagrama de bloques de la transformación <i>InvSubBytes</i>	
basada en el modelo matemático para 8 bits	51
Figura 3.18: Diagrama de circuito de la transformación	
<i>InvSubBytes</i> para 16 bytes basada en el modelo matemático	51
Figura 3.19: Diagrama de circuito para realizar la transformación	
<i>InvShiftRows</i>	52
Figura 3.20: Simulación del módulo <i>InvShiftRows</i>	52
Figura 3.21: Multiplicador para la transformación <i>InvMixColumns</i>	53
Figura 3.22: Diagrama de circuito para la transformación <i>InvMixColumns</i>	53
Figura 3.23: Simulación del módulo <i>InvMixColumns</i>	54
Figura 3.24: Circuito de expansión de clave de ronda	54
Figura 3.25: Diagrama de circuito general de la unidad de	
subclaves iterativa	56
Figura 3.26: Ruta de datos de la unidad de subclaves iterativa	56

Figura 3.27: Resultados de simulación funcional de la unidad de subclaves iterativa	58
Figura 3.28: Unidad de subclaves en cascada.....	59
Figura 3.29: Resultados de simulación de la unidad de subclaves en cascada	60
Figura 3.30: Bloque Ronda.....	61
Figura 3.31: Bloque Ronda Final.....	61
Figura 3.32: Bloque Ronda Inverso.....	62
Figura 3.33: Bloque Ronda Final Inverso	62
Figura 3.34: Arquitectura <i>pipeline</i> para el proceso de encriptación.....	63
Figura 3.35: Resultados de simulación para el proceso de encriptación en modo <i>ECB</i>	64
Figura 3.36: Esquema de cifrado para múltiples bloques en modo <i>ECB</i>	65
Figura 3.37: Proceso de encriptación, lectura y escritura entre la unidad de control, las memorias y el módulo de encriptación	66
Figura 3.38: Datos encriptados en formato hexadecimal escritos en <i>RAM</i>	67
Figura 3.39: Datos encriptados en formato <i>ASCII</i> escritos en memoria <i>RAM</i> ..	67
Figura 3.40: Organización <i>pipeline</i> para el proceso de desencriptación	68
Figura 3.41: Resultados de simulación para el proceso de desencriptación en modo <i>ECB</i>	69
Figura 3.42: Esquema de descifrado para múltiples bloques en modo <i>ECB</i>	70
Figura 3.43: Proceso de desencriptación, lectura y escritura entre la unidad de control, las memorias y el módulo de desencriptación en formato <i>ASCII</i>	71
Figura 3.44: Proceso de desencriptación, lectura y escritura entre la unidad de control, las memorias y el módulo de desencriptación en formato hexadecimal	71
Figura 3.45: Lectura de Datos desencriptados escritos en memoria <i>RAM</i>	72
Figura 3.46: Distribución del bloque de contador	73
Figura 3.47: Diagrama de circuito del módulo de bloque contador	74
Figura 3.48: Resultados de simulación del módulo contador	74
Figura 3.49: Diagrama de bloques de circuito para el proceso de encriptación y desencriptación en modo <i>CTR</i>	75

Figura 3.50: Resultados de simulación para el proceso de encriptación en modo contador.....	75
Figura 3.51: Esquema de cifrado y descifrado para múltiples bloques en modo <i>CTR</i>	76
Figura 3.52: Proceso de encriptación, lectura y escritura entre la unidad de control, las memorias y el módulo de encriptación ..	77
Figura 3.53: Datos encriptados en formato hexadecimal escritos en memoria <i>RAM</i>	78
Figura 3.54: Esquema de cifrado/descifrado para múltiples bloques en modo <i>CTR</i>	79
Figura 3.55: Proceso de desencriptación, lectura y escritura entre la unidad de control, las memorias y el módulo de encriptación	80
Figura 3.56: Lectura de Datos desencriptados escritos en memoria <i>RAM</i>	80
Figura 4.1: Resultados de encriptación en modo <i>ECB</i> usando <i>Chipscope pro</i>	85
Figura 4.2: Estimación de consumo de potencia para la implementación del circuito de encriptación en modo <i>ECB</i>	88
Figura 4.3: Datos encriptados en modo <i>ECB</i> leídos en cada posición de la memoria <i>RAM</i> sobre la <i>FPGA</i> usando <i>Chipscope pro</i>	88
Figura 4.4: Estimación de consumo de potencia para la implementación del circuito de encriptación para multiples bloques en modo <i>ECB</i>	88
Figura 4.5: Resultados de desencriptación en modo <i>ECB</i> usando <i>Chipscope pro</i>	89
Figura 4.6: Estimación de consumo de potencia para la implementación del circuito de desencriptación en modo <i>ECB</i>	90
Figura 4.7: Datos desencriptados en modo <i>ECB</i> leídos en cada posición de la memoria <i>RAM</i> sobre la <i>FPGA</i> usando <i>Chipscope pro</i>	92
Figura 4.8: Estimación de consumo de potencia para la implementación del circuito de desencriptación en modo <i>ECB</i> para multiples bloques.....	92

Figura 4.9: Estimación de consumo de potencia para la implementación del circuito de encriptación/desencriptación en modo <i>CTR</i>	93
Figura 4.10: Datos encriptados en modo <i>CTR</i> leídos en cada posición de la memoria <i>RAM</i> sobre la <i>FPGA</i> usando <i>Chipscope pro</i>	95
Figura 4.11: Estimación de consumo de potencia para la implementación del circuito de encriptación/desencriptación para multiples bloques en modo <i>CTR</i>	96
Figura 4.12: Datos desencriptados en modo <i>CTR</i> leídos en cada posición de la memoria <i>RAM</i> sobre la <i>FPGA</i> usando <i>Chipscope pro</i>	98
Figura 4.13: Arquitecturas de <i>Rijndael</i> para 128 bits: a) <i>Rolling</i> ; b) <i>Unrolling100</i>	100
Figura 4.14: Arquitectura <i>pipeline</i> de ronda interna y de ronda externa ó <i>Subpipeline</i>	101
Figura 4.15: Arquitectura <i>pipeline</i> de ronda externa	101
Figura 4.16: Arquitectura <i>pipeline</i> de multironda.....	102
Figura 4.17: Arquitectura <i>pipeline</i> de campo compuesto.....	103
Figura 4.18: <i>pipeline</i> de fase de sustitución de byte de campo compuesto....	103

LISTA DE TABLAS

Tabla 3.1: Tabla que contiene las constantes <i>Rcon</i> de acuerdo con el número de ronda.....	54
Tabla 3.2: Tabla que contiene los datos encriptados en modo <i>ECB</i> , escritos en memoria <i>RAM</i>	70
Tabla 3.3: Tabla que contiene los datos desencriptados en modo <i>ECB</i> , escritos en memoria <i>RAM</i>	72
Tabla 3.4: Resultados de encriptación en modo contador escritos en memoria <i>RAM</i>	78
Tabla 3.5: Resultados de desencriptación en modo contador escritos en memoria <i>RAM</i>	81
Tabla 4.1: Uso de recursos en <i>FPGA</i> y tiempos de retardo reportados para cada módulo de transformación	82

Tabla 4.2: Uso de recursos en <i>FPGA</i> y tiempos de retardo reportados para la unidad de subclaves	83
Tabla 4.3: Uso de recursos en <i>FPGA</i> , <i>throughput</i> y tiempo de procesamiento para el circuito de encriptación <i>pipeline</i> en modo <i>ECB</i>	84
Tabla 4.4: Uso de recursos en <i>FPGA</i> , <i>throughput</i> y tiempo de procesamiento para el circuito de encriptación <i>pipeline</i> en modo <i>ECB</i> para múltiples bloques.....	86
Tabla 4.5: Uso de recursos en <i>FPGA</i> y tiempo de procesamiento para el circuito de desencriptación <i>pipeline</i> en modo <i>ECB</i>	88
Tabla 4.6: Uso de recursos en <i>FPGA</i> , <i>throughput</i> y tiempo de procesamiento para el circuito de desencriptación <i>pipeline</i> en modo <i>ECB</i> para múltiples bloques.....	70
Tabla 4.7: Uso de recursos en <i>FPGA</i> , <i>throughput</i> y tiempo de procesamiento para el circuito de encriptación/desencriptación <i>pipeline</i> en modo <i>Counter (CTR)</i>	93
Tabla 4.8: Uso de recursos en <i>FPGA</i> , <i>throughput</i> y tiempo de procesamiento para el circuito de encriptación/desencriptación <i>pipeline</i> en modo <i>Counter (CTR)</i> para múltiples bloques	94
Tabla 4.9: Área, frecuencia máxima, <i>throughput</i> y consumo de potencia para la encriptación y desencriptación en los modos <i>ECB</i> y <i>CTR</i>	98
Tabla 4.10: Área, frecuencia máxima, <i>throughput</i> y consumo de potencia para la encriptación y desencriptación en los modos <i>ECB</i> y <i>CTR</i> para múltiples bloques	99
Tabla 4.11: Implementaciones en hardware reprogramable de arquitectura <i>pipeline</i> para el algoritmo de <i>Rijndael</i>	104

INTRODUCCIÓN

Las tecnologías de la información y las comunicaciones han avanzado hasta el punto de revolucionar considerablemente la forma en que los seres humanos interactúan y realizan las actividades cotidianas. Mediante la red de Internet se realizan actividades como pagos y compras electrónicas que involucran números de identificación personal y códigos de tarjetas de crédito, envío de información confidencial en forma de texto, voz ó video, información de datos bancarios, votos electrónicos, contraseñas, entre muchos otros. Parte de la información (y a veces toda) enviada a través de un canal de comunicación puede ser interceptado por entidades o personas no autorizadas, si dicha información no está codificada o la codificación es vulnerable a ser decodificada. Por lo anterior, la seguridad en la transferencia de la información se ha convertido en un factor importante y deben utilizarse técnicas de transmisión segura cuando se usan canales de comunicación inseguros de acceso público como Internet.

Actualmente uno de los estándares más utilizados para brindar seguridad a las comunicaciones, la información y las entidades que se comunican es el estándar *AES* (*Advanced Encryption Standard*), el cual es un esquema de cifrado por bloques adoptado como un estándar de cifrado por el gobierno de los Estados Unidos. El *AES* fue anunciado por el Instituto Nacional de Estándares y Tecnología (*National Institute of Standards and Technology: NIST*) como publicación 197 del *Federal Information and Publication Standards FIPS (FIPS-197)* de los Estados Unidos el 26 de noviembre de 2001 después de un proceso de estandarización. El estándar *AES* se puede implementar tanto en hardware como en software, sin embargo las implementaciones en software son más populares debido a que se pueden encontrar en aplicaciones como librerías de lenguajes de programación como Java, *Javascript*, *C++*, *C#/.NET*, *LabView*, *PHP*; herramientas de compresión de archivos como *RAR*, *WinZip*; encriptación de discos como *DiskCryptor*, *FileVault*; encriptación de archivos como *AES crypt*, *gKrypt* (*AES* en *GPUs*); protocolos de comunicación para redes de área local como *IEEE 802.11i*, *IPsec*, entre otros.

El presente trabajo de grado consiste en la concepción y diseño de un hardware para la implementación del algoritmo de *Rijndael* (estándar *AES*: *Advanced Encryption Standard*) para 128 bits tanto para los procesos de encriptación como desencriptación en modos de configuración de entrada no realimentados *ECB* (*Electronic Codebook*) y *CTR* (*Counter*). Para el diseño se utilizó el lenguaje de descripción de hardware *VHDL* [4], haciendo uso de la plataforma de síntesis e implementación de sistemas digitales *Xilinx ISE 13.1* [5]. La implementación se realizó utilizando la *FPGA XC5VLX110T* [13] de la familia *Virtex 5* de *Xilinx*.

El objetivo principal de este trabajo ha sido diseñar e implementar en hardware reconfigurable (*FPGA*) el estándar *FIPS 197* [1] usando los modos de operación no realimentados *ECB* y *CTR* expuestos en el estándar *SP800-38A* [2] del *NIST*. La particularidad de estos modos de configuración es que

permiten implementación segmentada (*pipeline*) logrando obtener un nivel de paralelismo a nivel de circuitos funcionales y acelerando la ejecución de los bloques de datos a encriptar o desencriptar. En los diseños obtenidos se evalúan las principales características como utilización de recursos hardware, velocidad de procesamiento de datos y consumo de potencia.

El presente trabajo se encuentra distribuido de la siguiente manera:

El Capítulo 1 hace una presentación del Estándar de Encriptación Avanzado (AES: *Advanced Encryption Standard*). Aquí se exponen conceptos básicos como las operaciones aritméticas en el campo finito y las funciones de transformación asociadas a los algoritmos de encriptación y desencriptación.

El Capítulo 2 presenta los Modos de operación del Algoritmo de Rijndael de manera general. Se expone la estructura básica de los modos de operación realimentados y no realimentados (ECB, CBC, CFB, OFB, CTR), en los cuales puede funcionar este algoritmo de cifrado por bloques de clave simétrica.

El Capítulo 3 presenta el diseño de las arquitecturas hardware utilizadas para la implementación del estándar AES-128 en los modos de operación no realimentados (ECB: *Electronic CodeBook* y CTR: *Counter*). También se muestran los resultados de simulación para cada una de las organizaciones circuitales, haciendo uso de los vectores de prueba dispuestos en los estándares [1] y [2].

El Capítulo 4 presenta los resultados y análisis de implementación del estándar AES-128. Estos resultados se obtuvieron usando la herramienta de verificación de hardware *Chipscope pro* 13.1 sobre una *FPGA* de Xilinx, además se muestran los datos reportados por el software *Xilinx* 13.1 tales como utilización de área, retardos, tiempos de procesamiento, frecuencia máxima, consumo de potencia y *throughput*. Finalmente, el Capítulo 5 presenta las conclusiones y las observaciones obtenidas en el desarrollo del trabajo de grado; también se presentan posibles trabajos futuros derivados de este proyecto.

CAPÍTULO 1: STANDAR DE ENCRYPTION AVANZADO

(AES:*Adavanced Encryption Standard*)

1.1 Historia del algoritmo de *Rijndael* [3]

Conforme el algoritmo *DES* (*Data Encryption Standard*) comenzó a evidenciar algunas debilidades potenciales, aún con triple *DES*, el *NIST* (*National institute of standards and technology*), que es la agencia perteneciente al Departamento de Comercio de Estados Unidos encargada de aprobar estándares del Gobierno de Estados Unidos, decidió que se necesitaba un nuevo estándar criptográfico para uso no confidencial. El *NIST* estaba conciente de la controversia alrededor de *DES* y sabía que si anunciaba un nuevo estándar, todos los que tuvieran conocimiento sobre criptografía supondrían de manera automática que la *NSA* (*National Security Agency*) habría conspirado con el fin de leer todo lo que se encryptara con ese nuevo estándar. Bajo estas condiciones, probablemente nadie utilizaría el estándar.

Por esto, el *NIST* adoptó una estrategia sorprendentemente diferente para una burocracia gubernamental: promovió un concurso. En enero de 1997, los investigadores de todo el mundo fueron invitados a emitir propuestas para un nuevo estándar, que se llamaría Estándar de Encriptación Avanzado (*AES*: *Advanced Encryption standard*). Las reglas fueron:

- El algoritmo debía ser un cifrado de bloques simétricos.
- Todo el diseño debía ser público.
- Debían soportarse longitudes de clave de 128, 192 y 256 bits.
- Debían ser posibles implementaciones en software y en hardware.

Se realizaron quince propuestas:

- **CAST-256** (*Entrust Technologies, Inc.*)
- **CRYPTON** (*FutureSystems, Inc.*)
- **DEAL** (*Richard Outerbridge, Lars Knudsen*)
- **DFC** (*CNRS – Centre Nationalpour la RechercheScientifique – EcoleNormaleSuperieure*)
- **E2** (*NTT – Nippon Telegraph and Telephone Corporation*)
- **FROG** (*TecApro International, S.A.*)
- **HPC** (*RichSchroeppel*)
- **LOKI97** (*Lawrie Brown, Josef Pieprzyk, Jennifer Seberry*)
- **MAGENTA** (*Deutsche Telekom AG*)
- **MARS** (*IBM*)
- **RC6** (*RSA Laboratories*)
- **RIJNDAEL** (*John Daemen, Vincent Rijmen*)
- **SAFER+** (*CylinkCorporation*)
- **SERPENT** (*Ross Anderson, Eli Biham, Lars Knudsen*)
- **TWOFISH** (*Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson*)

Que fueron presentadas en conferencias, en las que se alentó a los asistentes a encontrar errores en ellas. En agosto de 1998, el *NIST* seleccionó cinco finalistas con base en seguridad, eficiencia, simplicidad, flexibilidad y requerimientos de memoria. En la última conferencia los finalistas obtuvieron los siguientes puntajes:

- **RINJDAEL** (de Joan Daemen y Vincent Rijmen, 86 votos).
- **SERPENT** (de Ross Anderson, Eli Biham y Lars Knudsen, 59 votos).
- **TWOFISH** (de un equipo encabezado por Bruce Schneier, 31 votos).
- **RC6** (de los Laboratorios RSA, 23 votos).
- **MARS** (de IBM, 13 votos).

En octubre de 2000, el *NIST* hizo público su voto por *Rijndael*, y en noviembre de 2001 *Rijndael* se convirtió en el estándar del gobierno de Estados Unidos. Este se publicó como *FIPS 197 (Federal Information Processing Standards)*. El nombre *Rijndael* se deriva de los apellidos de los autores belgas: Rijmen y Daemen.

Rijndael soporta longitudes de clave y tamaños de bloque de 128 a 256 bits. Las longitudes de clave y de bloque se pueden elegir de manera independiente. Sin embargo, el *AES* especifica que el tamaño de bloque debe ser de 128 bits y la longitud de clave puede ser de 128, 192 o 256 bits.

1.2. Introducción al estándar [1]

El estándar *AES* especifica el algoritmo de *Rijndael* como un bloque de cifrado simétrico que puede procesar bloques de datos de 128 bits, usando claves de cifrado con longitudes de 128, 192 ó 256 bits.

Rijndael se diseñó para manejar otros tamaños de bloque y longitudes de clave, sin embargo estos no son adoptados por el estándar *AES*.

Las entradas y las salidas para el algoritmo *AES* consisten cada una de 128 bits. La clave de cifrado para el algoritmo *AES* es una secuencia de 128, 192 ó 256 bits. Otras longitudes de entradas, salidas y claves de cifrado no son permitidas por el estándar.

La unidad básica para procesamiento en el algoritmo *AES* es el byte. Las secuencias de entrada, salida y clave de cifrado son procesadas como arreglos de bytes.

Todos los valores de byte en el algoritmo *AES* serán presentados como la concatenación de sus bits individuales $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$. Cada byte es interpretado como un elemento de campo finito usando representación polinomial así:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^7 b_i x^i \quad (1.1)$$

Por ejemplo $\{01100011\}$ identifica el elemento de campo finito específico $x^6 + x^5 + x + 1$.

Es conveniente denotar los valores de byte usando notación hexadecimal donde dos cadenas de 4 bits serán consideradas como una unidad.

Por ejemplo: $\{01100011\}$ en notación hexadecimal es $\{63\}$.

Internamente, las operaciones del algoritmo AES se organizan como un arreglo de dos dimensiones de bytes llamado estado. El estado consiste de cuatro filas de bytes, cada fila contiene N_b bytes, donde N_b es la longitud del bloque dividido en 32 ($N_b = 4$). En el estado denotado por el símbolo S , cada byte individual tiene dos índices, el número de fila r en el rango $0 \leq r < 4$ y su número de columna c en el rango $0 \leq c < 4$. Esto permite a un byte individual del estado ser referenciado como $S_{r,c}$ ó $S[r, c]$. Para este estándar, $N_b = 4$, $0 \leq c < 4$.

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

Figura 1.1. Matriz de Estado, S .

Al comienzo del cifrado ó descifrado la entrada a se copia a la matriz de estado. Las operaciones de cifrado o descifrado se llevan a cabo en la matriz de estado, después su resultado final se copia a la salida b :

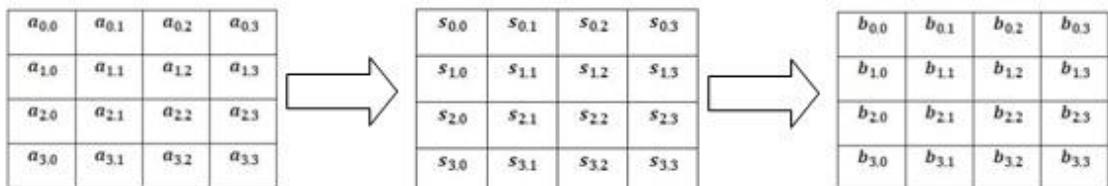


Figura 1.2. La matriz de estado.

Los cuatro bytes de cada columna de la matriz de estado forman palabras de 32 bits, y el número de fila r provee un índice para los cuatro bytes dentro de cada palabra. Por lo tanto, el estado se puede interpretar como un arreglo unidimensional de palabras de 32 bits (Columnas), $w_0 \dots w_3$ donde la columna c provee un índice al arreglo. Por lo tanto el estado se puede considerar como un arreglo de cuatro palabras, como sigue:

$$\begin{aligned}
 w_0 &= s_{0,0}s_{1,0}s_{2,0}s_{3,0} & w_2 &= s_{0,2}s_{1,2}s_{2,2}s_{3,2} \\
 w_1 &= s_{0,1}s_{1,1}s_{2,1}s_{3,1} & w_3 &= s_{0,3}s_{1,3}s_{2,3}s_{3,3}
 \end{aligned} \tag{1.2}$$

Todos los bytes en el algoritmo AES se interpretan como elementos de campo finito. Los elementos de campo finito se pueden sumar y multiplicar.

La suma de dos elementos en un campo finito se logra al sumar los coeficientes de las potencias correspondientes en los polinomios para los dos elementos. La suma se obtiene con la operación XOR. En consecuencia, la resta de polinomios es idéntica a la suma de los mismos.

Alternativamente, la suma de elementos de campo finito se puede describir como la suma módulo 2 de los bits correspondientes en el byte. Para dos bytes $(a_7a_6a_5a_4a_3a_2a_1a_0)$ y $(b_7b_6b_5b_4b_3b_2b_1b_0)$ la suma es $(c_7c_6c_5c_4c_3c_2c_1c_0)$, donde:

$$c_i = a_i \oplus b_i \quad (c_7 = a_7 \oplus b_7, c_6 = a_6 \oplus b_6, \dots, c_0 = a_0 \oplus b_0) \quad (1.3)$$

Por ejemplo las siguientes expresiones son equivalentes las unas a las otras:

Notación polinomial: $(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$

Notación binaria: $\{01010111\} \oplus \{10000011\} = \{11010100\}$

Notación hexadecimal: $\{57\} \oplus \{83\} = \{d4\}$

En la representación polinomial, la multiplicación en $GF(2^8)$ corresponde a la multiplicación de polinomios modulo un **polinomio irreducible** de grado 8. Un polinomio es irreducible si solo sus divisores son el uno y el mismo. Para el algoritmo AES, este polinomio es:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \quad (1.4)$$

O $\{01\}\{1b\}$ en notación hexadecimal.

Por ejemplo, $\{57\} * \{83\} = \{c1\}$ ya que:

$$(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

Entonces:

$$\begin{aligned} & (x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1) \bmod (x^8 + x^4 + x^3 + x + 1) \\ &= x^7 + x^6 + 1 \end{aligned}$$

La reducción modular por $m(x)$ asegura que el resultado será un polinomio binario de grado menor que 8, y por lo tanto este se puede representar por un byte.

Al multiplicar por x el polinomio binario de la forma:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^7 b_i x^i$$

Se obtiene:

$$b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \quad (1.5)$$

El resultado $x * b(x)$ se obtiene al reducir la ecuación anterior mediante la operación modular con $m(x)$. Si $b_7 = 0$ el resultado ya esta en la forma reducida. Si $b_7 = 1$, la reducción se logra al sustraer (con XOR) el polinomio $m(x)$. Luego la multiplicación por x (por ejemplo por {00000010} ó {02}) puede ser implementada a nivel de byte como un desplazamiento a la izquierda y luego una operación condicional XOR con el valor {1b}. Esta operación de bytes se denota como la operación *xtime()*. La multiplicación por potencias más altas de x se pueden implementar por aplicaciones repetidas de *xtime()*. La multiplicación por cualquier constante se puede implementar al sumar resultados intermedios.

Por ejemplo {57} * {13} = {fe} ya que:

$$\begin{aligned} \{57\} * \{02\} &= \text{xtime}(\{57\}) = \{ae\} \\ \{57\} * \{04\} &= \text{xtime}(\{ae\}) = \{47\} \\ \{57\} * \{08\} &= \text{xtime}(\{47\}) = \{8e\} \\ \{57\} * \{10\} &= \text{xtime}(\{8e\}) = \{07\} \end{aligned}$$

Por lo tanto:

$$\begin{aligned} \{57\} * \{13\} &= \{57\} * (\{01\} \oplus \{02\} \oplus \{10\}) \\ \{57\} * \{13\} &= \{57\} \oplus \{ae\} \oplus \{07\} = \{fe\} \end{aligned}$$

1.3. Operación de cifrado (Encriptación)

Al comienzo del cifrado, la entrada se copia a la matriz de estado y después de una transformación *AddRoundKey* el estado se transforma al ser procesado en una función ronda diez veces, Con la ronda final (ronda diez) diferenciándose ligeramente de las restantes nueve rondas.

Cada ronda se compone de cuatro transformaciones: *SubBytes*, *ShiftRows*, *MixColumns* y *AddRoundKey* excepto la última ronda la cual excluye la transformación de *MixColumns*.

Cada ronda i necesita la salida de la ronda $(i - 1)$ y la respectiva subclave proveniente de la rutina de expansión de clave.

En la figura 1.3(a) y 1.3(b) se ilustra el algoritmo para el proceso de encriptación en forma de pseudocódigo y diagrama de flujo respectivamente, Como se puede apreciar, inicialmente la transformación *AddRoundKey* es aplicada a la matriz de estado, posteriormente nueve rondas. La ronda final procesa el estado sin incluir la transformación *MixColumns*. Finalmente se puede notar que para cada transformación *AddRoundKey* hay una subclave que se compone de 4 palabras cada una de 32 bits.

```

Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]

    state = in

    AddRoundKey(state, w[0, Nb-1])

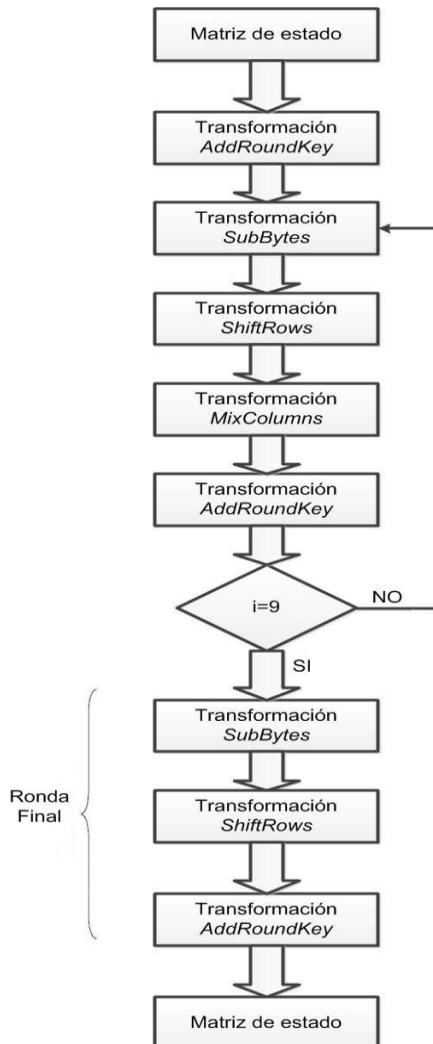
    for round = 1 step 1 to Nr-1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    end for

    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

    out = state
end

```

(a). Pseudocódigo. [1]



(b). Diagrama de flujo

Figura 1.3 Algoritmo para el proceso de encriptación

1.3.1. Transformación *SubBytes*

La transformación *SubBytes* es una sustitución de byte no lineal que opera independientemente en cada byte del estado usando una tabla de sustitución llamada *S-Box*. Dicha tabla es invertible y se construye al realizar dos transformaciones:

- Se calcula el inverso multiplicativo en el campo finito $GF(2^8)$; el elemento $\{00\}$ es mapeado así mismo .
- Se aplica la siguiente transformación afín sobre $GF(2)$:

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i \quad (1.6)$$

Para $0 \leq i < 8$, donde b_i es el i ésimo bit del byte, y c_i es el i ésimo bit de un byte c con el valor $\{63\}$ o $\{01100011\}$.

En forma matricial, el elemento de transformación afín del *S-Box* se puede expresar como:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \quad (1.7)$$

La figura 1.4 ilustra el efecto de la transformación de *SubBytes* en el estado:

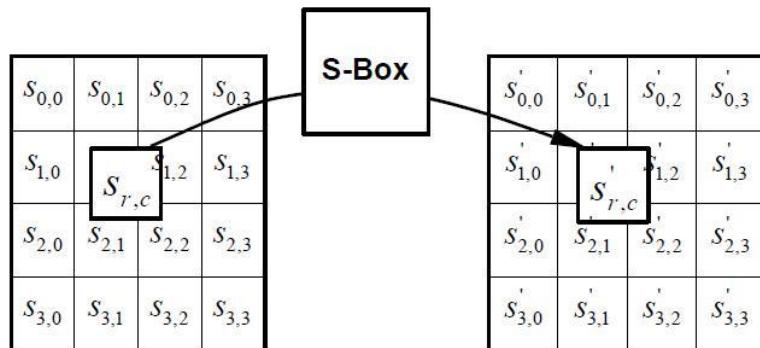


Figura 1.4. La transformación *SubBytes* aplica la tabla *S-BOX* a cada byte del estado. [1]

La figura 1.5, muestra la S-Box usada en la transformación *SubBytes*. Por ejemplo, si $S_{1,1} = \{53\}$, entonces el valor en la sustitución estaría determinado por la intersección de la fila con índice '5' y la columna con índice '3', esto daría como resultado el valor {ed}.

		y																
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
		0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
		1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
		2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
		3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
		4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
		5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
		6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
		7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
		8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
		9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
		a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
		b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
		c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
		d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
		e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
		f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figura 1.5. S-Box: valores de sustitución para el byte xy (en formato hexadecimal). [1]

1.3.2. Transformación *ShiftRows*

En la transformación *ShiftRows*, los bytes en las últimas tres filas del estado se desplazan cíclicamente sobre diferentes números de bytes (*offsets*). La primera fila, $r = 0$, no se desplaza.

Específicamente, la transformación *ShiftRows* funciona de la siguiente manera:

$$S'_{r,c} = S_{r,(c+shift(r,Nb))modNb} \text{ para } 0 < r < 4 \text{ y } 0 \leq c < Nb \quad (1.8)$$

Donde el valor de desplazamiento $Shift(r, Nb)$ depende del número de la fila r , por ejemplo para $Nb = 4$ se tiene:

$$Shift(1,4) = 1; \quad Shift(2,4) = 2; \quad Shift(3,4) = 3$$

Esto tiene el efecto de mover los bytes de posiciones altas a posiciones más bajas en la fila (valores más bajos de la columna c en una fila dada), mientras que los bytes más bajos circulan hacia el tope de la fila.

La figura 1.6 ilustra la transformación *ShiftRows*:

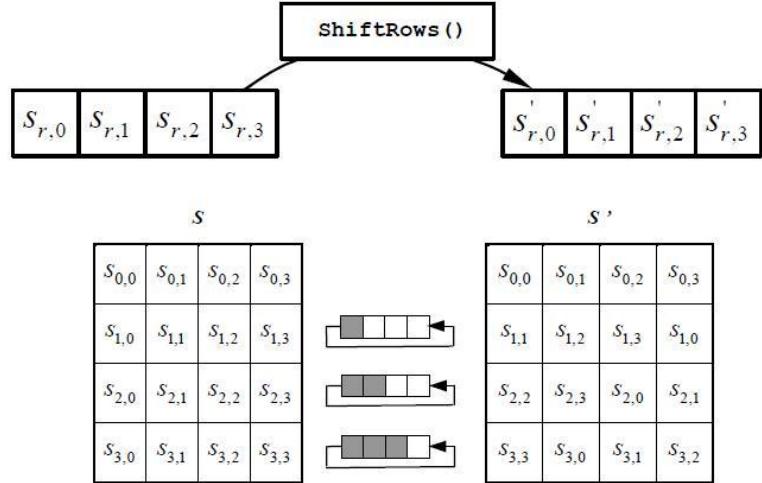


Figura 1.6. La transformación *shiftrows* rota cíclicamente las últimas tres filas del estado. [1]

1.3.3. Transformación *MixColumns*

La transformación *MixColumns* opera en el estado columna por columna, tratando a cada columna como un polinomio de cuatro términos. Las columnas se consideran como polinomios sobre $GF(2^8)$ y se multiplican modulo $x^4 + 1$ con un polinomio fijo $a(x)$ dado por:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \quad (1.9)$$

Esto puede ser escrito como una multiplicación matricial. Sea

$$s'(x) = a(x) \otimes s(x)$$

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad 0 \leq c < Nb. \quad (1.10)$$

Como resultado de esta multiplicación, los cuatro bytes en una columna son reemplazados de la siguiente manera:

$$\begin{aligned} s'_{0,c} &= (\{02\} * s_{0,c}) \oplus (\{03\} * s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} * s_{1,c}) \oplus (\{03\} * s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} * s_{2,c}) \oplus (\{03\} * s_{3,c}) \\ s'_{3,c} &= (\{03\} * s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} * s_{3,c}) \end{aligned} \quad (1.11)$$

La figura 1.7 ilustra la transformación de *MixColumns*:

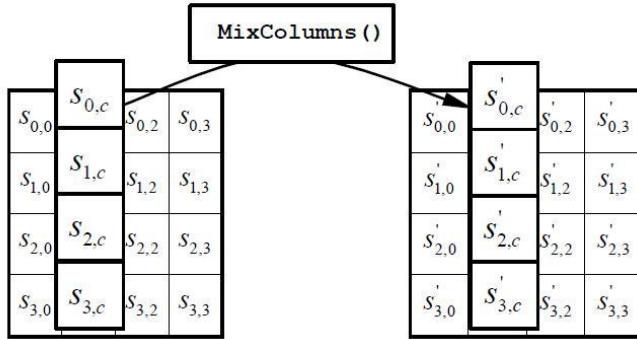


Figura 1.7. La transformación *MixColumns* opera en el estado columna por columna. [1]

1.3.4. Transformacion *AddRoundKey*

En la transformación *AddRoundKey*, una clave de ronda o subclave es añadida a la matriz de estado con una simple operación XOR. Cada clave de ronda o subclave consiste de N_b palabras provenientes de la unidad de subclaves. Esas N_b palabras son cada una añadidas a las columnas del estado como sigue:

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round * Nb + c}] \quad 0 \leq c < Nb \quad (1.12)$$

Donde $[w_i]$ son las palabras de la generación de subclaves, y *round* es un valor en el rango $0 \leq round \leq N_r$. En el cifrado, la adición de la clave de ronda inicial ocurre cuando $round = 0$, antes de la primera aplicación de la función ronda. La aplicación de la transformación *AddRoundKey* a las N_r rondas del cifrado ocurre cuando $1 \leq round \leq N_r$.

La acción de esta transformación se ilustra en la figura 1.8, donde $l = round * Nb$.

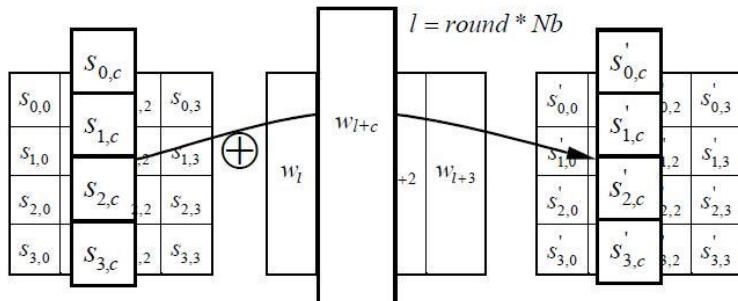


Figura 1.8. La transformación *AddRounKey* realiza una XOR entre cada columna de la matriz de estado con una columna de la clave de ronda. [1]

1.3.5. Expansión de clave

El algoritmo AES toma la clave de cifrado K , y realiza una rutina de expansión de clave para generar una clave expandida. La expansión de clave genera un total de

$N_b(N_r + 1)$ palabras, donde N_b es el número de columnas de la matriz de estado y N_r es el número de rondas que ejecuta el algoritmo. El algoritmo requiere un conjunto inicial de N_b palabras, y cada una de las N_r rondas requiere N_b palabras de datos de clave. La resultante clave expandida consiste de un arreglo lineal de palabras de cuatro bytes, denotadas por $[w_i]$, con i en el rango $0 \leq i < N_b(N_r + 1)$.

Para el estándar AES-128 la clave inicial de 128 bits necesita ser expandida a $4*(10+1)$, es decir 44 palabras de 32 bits. Esto equivale a once claves de 128 bits. La clave inicial es la clave de encriptación y se utiliza en la ronda inicial del algoritmo. Las claves siguientes se derivan de la clave que les precede según la función f , de la siguiente manera:

$$SubClaveRonda_i = f(SubClaveRonda_{i-1}) \quad (1.13)$$

La clave inicial se representa como un arreglo lineal W en el cual las subclaves de ronda se obtienen de la clave inicial K_0 :

$$K_0 = (w_3, w_2, w_1, w_0) \quad (1.14)$$

Las primeras N_k ($N_k = 4$) palabras de la clave expandida se llenan con la clave de encriptación. Cada palabra siguiente, $w[i]$, es resultado de la operación XOR entre la palabra previa $w[i - 1]$ y la palabra que está N_k posiciones más cercana $w[i - N_k]$. Para las palabras que están en posiciones múltiplo de N_k , se aplica una transformación a $w[i - 1]$ antes de la XOR, seguido por una operación XOR con una constante de ronda $Rcon[i]$. La transformación consiste en un desplazamiento cíclico de los bytes en una palabra (*RotWord*), seguida por la aplicación de una transformación mediante S-Box (*SubWord*) a cada byte.

SubWord() es una función que toma una entrada de cuatro bytes y aplica la función S-Box a cada uno de los cuatro bytes para producir una palabra de salida.

RotWord() toma una palabra $[a_0, a_1, a_2, a_3]$ como entrada, realiza una permutación cíclica, y retorna la palabra $[a_1, a_2, a_3, a_0]$.

El arreglo de palabras constante de ronda, **Rcon[i]**, contiene los valores dados por $[x^{i-1}, \{00\}, \{00\}, \{00\}]$, con x^{i-1} siendo potencias de x (x es denotado como $\{02\}$) en el campo $GF(2^8)$.

1.4. Operación de cifrado inverso (Desencriptación)

Las transformaciones de cifrado pueden ser invertidas y luego implementadas en orden inverso para producir un cifrado inverso directo para el algoritmo AES. Las transformaciones individuales usadas en el cifrado inverso: *InvShiftRows*, *InvSubBytes*, *InvMixColumns*, y *AddRoundKey* procesan el estado para recuperar el texto plano. En la figura 1.9(a) y 1.9(b) se ilustra el algoritmo para el proceso de desencriptación mediante el uso de pseudocódigo y un diagrama de flujo respectivamente.

```

InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]

    state = in

    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

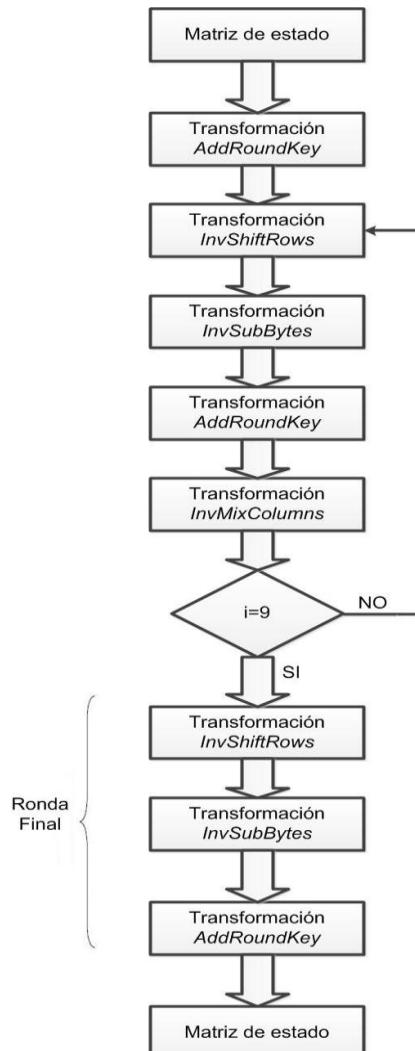
    for round = Nr-1 step -1 downto 1
        InvShiftRows(state)
        InvSubBytes(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
        InvMixColumns(state)
    end for

    InvShiftRows(state)
    InvSubBytes(state)
    AddRoundKey(state, w[0, Nb-1])

    out = state
end

```

(a) Pseudocódigo. [1]



(b). Diagrama de flujo.

Figura 1.9. Algoritmo para el proceso de desencriptación.

1.4.1. Transformación *InvShiftRows*

Es el inverso de la transformación *ShiftRows*. Los bytes en las últimas tres filas del estado son cíclicamente desplazadas sobre diferentes números de bytes (*offsets*). La primera fila, $r = 0$, no es desplazada. Las últimas tres filas son cíclicamente desplazadas por $N_b - shift(r, N_b)$ bytes, donde el valor de $shift(r, N_b)$ depende del número de la fila, como se explicó en la sección 1.3.2.

Específicamente, la transformación *InvShiftRows* funciona de la siguiente manera:

$$S'_{r,(c+shift(r,Nb)) \bmod Nb} = S_{r,c} \quad \text{para } 0 < r < 4 \quad \text{y} \quad 0 \leq c < Nb \quad (1.15)$$

La siguiente figura ilustra la transformación *InvShiftRows*:

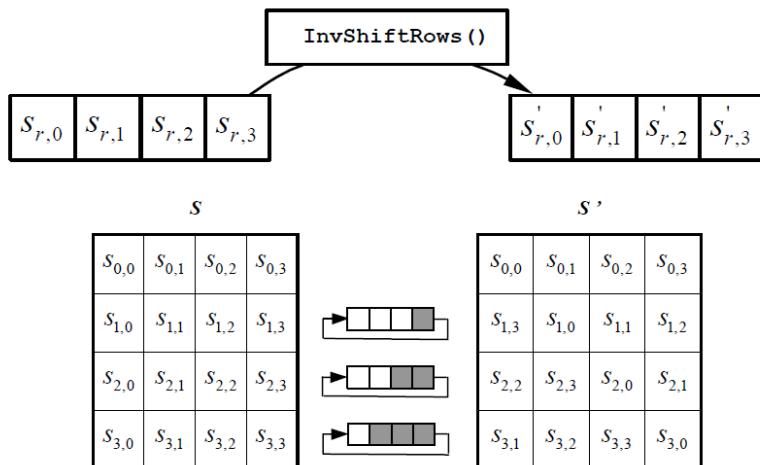


Figura 1.10. La transformación *InvShiftRows* rota cíclicamente las últimas tres filas de la matriz de estado. [1]

1.4.2. Transformación *InvSubBytes*

Esta transformación es la inversa de la transformación *SubBytes* en la cual la *S-Box* inversa se aplica a cada byte del estado. Esto se obtiene al aplicar la inversa de la transformación afín (ecuación 1.7) seguido de tomar el inverso multiplicativo en $GF(2^8)$. La *S-box* inversa usada en la transformación *InvSubBytes* se presenta a continuación:

		y																
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
x		0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb		
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e		
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25		
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92		
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84		
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06		
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b		
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73		
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e		
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b		
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4		
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f		
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef		
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61		
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d		

Figura 1.11. S-Box inversa: valores de sustitución para *InvSubBytes* del byte xy (en formato hexadecimal). [1]

1.4.3. Transformación *InvMixColumns*

Esta transformación es la inversa de la transformación *MixColumns*. La transformación *InvMixColumns* opera en el estado columna por columna, tratando cada columna como un polinomio de cuatro términos. Las columnas son consideradas como polinomios sobre $GF(2^8)$ y multiplicadas modulo $x^4 + 1$ con un polinomio fijo $a^{-1}(x)$, dado por:

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\} \quad (1.16)$$

Esto puede ser escrito como la multiplicación matricial:

$$s'(x) = a^{-1}(x) \otimes s(x)$$

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad 0 \leq c < Nb. \quad (1.17)$$

1.4.4. Inversa de la transformación *AddRoundKey*

La transformación inversa de *AddRoundKey* es su propia inversa, ya que solamente involucra una aplicación de la operación XOR.

CAPÍTULO 2: MODOS DE OPERACIÓN

2.1. Introducción

Para utilizar los modos de operación se asume que un algoritmo de cifrado por bloques de clave simétrica aprobado como *FIPS (Federal Information processing standard)* se ha elegido como algoritmo subyacente y que una clave secreta aleatoria denotada como K se ha establecido entre las partes de la comunicación. La clave criptográfica regula el funcionamiento del algoritmo de cifrado por bloques y por lo tanto regula el funcionamiento del modo. Las especificaciones del cifrado por bloques, algoritmos y modos son públicos así que la seguridad del modo depende como mínimo de la clave.

En este capítulo al proceso de encriptación también se le llama la función de cifrado denotada como $CIPH_K$ y al proceso de desencriptación también se le llama la función inversa de cifrado denotada como $CIPH_K^{-1}$. Las entradas y salidas de ambas funciones son llamadas sus bloques de entrada o bloques de salida. Los bloques de entrada y de salida del algoritmo de cifrado por bloques tienen la misma longitud llamado el tamaño del bloque, denotado como b .

2.2. Modo *Electronic CodeBook (ECB)*

El modo *Electronic codebook (ECB)* es un modo de confidencialidad que permite cifrar un bloque de texto plano dada una clave secreta. En el proceso de encriptación *ECB*, la función de cifrado se aplica de manera independiente a cada bloque de texto plano. La secuencia resultante de bloques de salida componen el texto cifrado.

El proceso de encriptación en modo ECB se representa por:

$$C_j = CIPH_K(P_j) \quad \text{para} \quad j = 1 \dots n. \quad (2.1)$$

Dónde:

C_j : j -esimo bloque de texto cifrado.

$CIPH_K$: Función de cifrado.

P_j : j -esimo bloque de texto plano.

En el proceso de desencriptación *ECB*, la función de descifrado se aplica de manera independiente a cada bloque de texto cifrado. La secuencia resultante de bloques de salida conforman el texto plano.

El proceso de desencriptación en modo ECB se representa por:

$$P_j = CIPH_j^{-1}C_j \quad \text{para } j = 1 \dots n. \quad (2.2)$$

Dónde:

- P_j : j -esímo bloque de texto plano.
- $CIPH_j^{-1}$: Función inversa de cifrado.
- C_j : j -esímo bloque texto cifrado.

La figura 2.1 ilustra los procesos descritos:

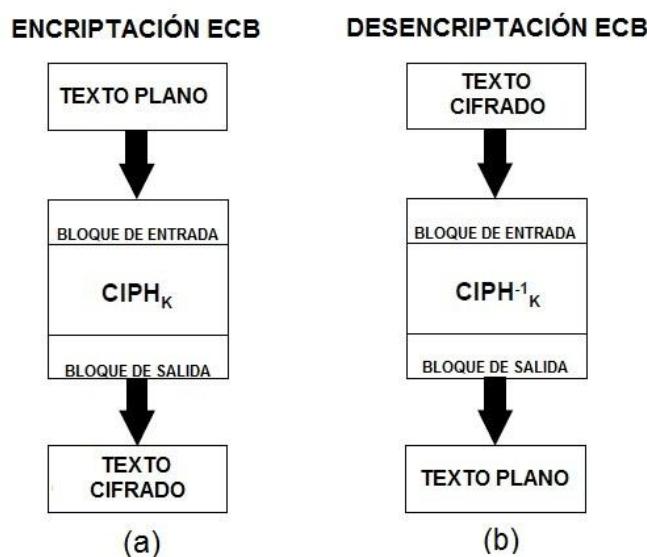


Figura 2.1. Modo *ECB* para los procesos de a) Encriptación y b) Desencriptación. [2]

Tanto en la encriptación como en la desencriptación en modo *ECB* es posible procesar en paralelo varias funciones de cifrado ó de descifrado.

Dada una clave secreta en el modo *ECB*, cualquier texto plano y su texto cifrado equivalente no cambian, incluso en el caso de que un mismo texto plano se encripte más de una vez. Si esta propiedad es indeseable en alguna aplicación particular, el modo *ECB* no debería usarse.

2.3. Modo *CBC* (*Cipher Block Chaining*)

El modo *CBC* es un modo de confidencialidad cuyo proceso de encriptación realiza la combinación de los bloques de texto plano con los bloques de texto cifrado previos. Este modo requiere un vector de inicialización (*IV: Initialization Vector*) que se combina con el primer bloque de texto plano. El vector de inicialización no

necesita ser secreto, pero debe ser impredecible. El modo *CBC* se define de la siguiente manera:

Para el proceso de Encriptación:

$$\begin{aligned} C_1 &= \text{CIPH}_K(P_1 \oplus IV) ; \\ C_j &= \text{CIPH}_K(P_j \oplus C_{j-1}) \quad \text{para } j = 2 \dots n \end{aligned} \quad (2.3)$$

Dónde:

C_j : j -esímo bloque de texto cifrado.

CIPH_K : Función de cifrado.

P_j : j -esímo bloque de texto plano.

IV : Vector de inicialización

Para el proceso de Desencriptación:

$$\begin{aligned} P_1 &= \text{CIPH}_k^{-1}C_1 \oplus IV ; \\ P_j &= \text{CIPH}_k^{-1}C_j \oplus C_{j-1} \quad \text{para } j = 2 \dots n \end{aligned} \quad (2.4)$$

Dónde:

P_j : j -esímo bloque de texto plano.

CIPH_j^{-1} : Función de cifrado inversa.

C_j : j -esímo bloque texto cifrado.

IV : Vector de inicialización

La figura 2.2 ilustra el diagrama de bloques correspondiente a los procesos de encriptación y desencriptación:

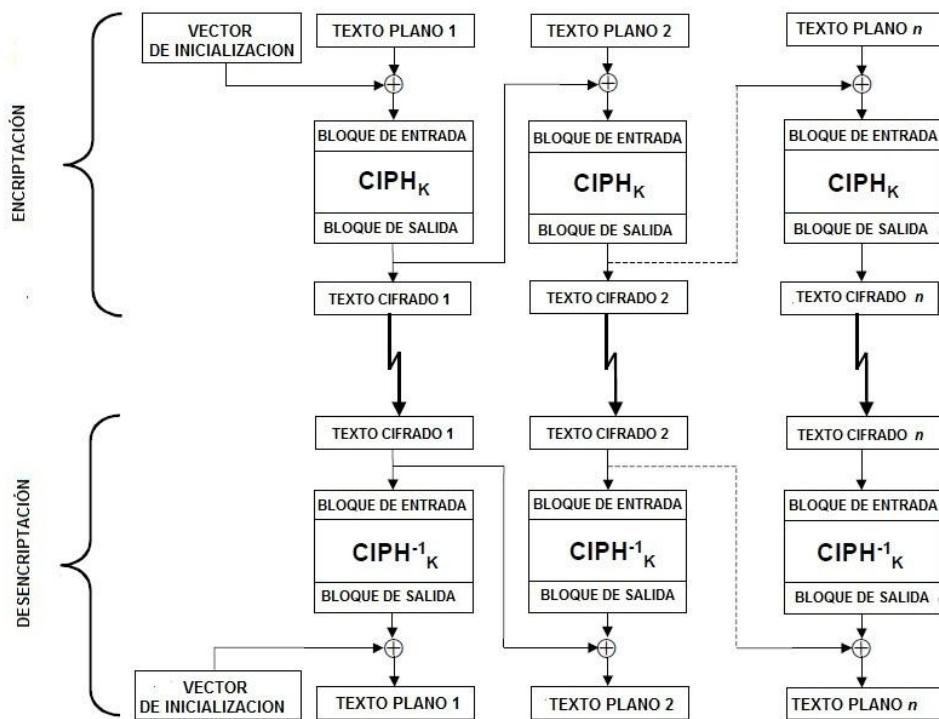


Figura 2.2. Diagrama de bloques del proceso de encriptación y desencriptación en Modo CBC. [2]

En la encriptación en modo CBC, el primer bloque de entrada se forma luego de realizar una operación XOR entre el primer bloque de texto plano y el vector de inicialización ($/V$). La función de cifrado es aplicada al primer bloque de entrada y el bloque resultante de salida es el primer bloque de texto cifrado. Este bloque de salida es también operado mediante una XOR con el segundo bloque de texto plano para producir el segundo bloque de salida. Este bloque es a su vez el segundo bloque de texto cifrado y se opera mediante una XOR con el siguiente bloque de texto plano para formar el siguiente bloque de entrada. Cada bloque sucesivo de texto plano es operado mediante una XOR con el bloque previo de salida/texto cifrado para producir el nuevo bloque de entrada. La función de cifrado es aplicada a cada bloque de entrada para producir el bloque de texto cifrado.

En la desencriptación CBC el proceso es similar pero con las funciones inversas, la función inversa de cifrado es aplicada al primer bloque de texto cifrado y el bloque resultante de salida es operado mediante una XOR con el vector de inicialización para recuperar el primer bloque de texto plano. La función inversa de cifrado es también aplicada al segundo bloque de texto cifrado y el resultante bloque de salida es operado mediante una XOR con el primer bloque de texto cifrado para recuperar el segundo bloque de texto plano. En general, para recuperar cualquier bloque de texto plano (excepto el primero), la función inversa de cifrado es aplicada al correspondiente bloque de texto cifrado, y el bloque resultante es operado mediante una XOR con el bloque de texto cifrado previo.

En la encriptación *CBC*, el bloque de entrada a cada operación de cifrado (excepto el primero) depende del resultado de la operación de cifrado previa, así que las operaciones de cifrado no pueden ser realizadas en paralelo. En la desencriptación *CBC*, sin embargo, los bloques de entrada para la función inversa de cifrado, es decir los bloques de texto cifrado están inmediatamente disponibles, así que múltiples operaciones de cifrado inverso pueden ser realizadas en paralelo.

2.4. Modo *CFB* (*Cipher feedback*)

El modo *CFB* (*Cipher feedback*) es un modo de confidencialidad que realiza la realimentación de segmentos de texto cifrado sucesivos a las entradas de la función de cifrado para generar los bloques de salida que son operados mediante una función XOR con el texto plano para producir el texto cifrado y viceversa. El modo *CFB* requiere un vector de inicialización (*/IV*) como bloque inicial de entrada. El vector de inicialización no necesita ser secreto, pero debe ser impredecible.

El modo *CFB* además requiere un parámetro entero, denotado como s , tal que $1 \leq s \leq b$. En la especificación del modo *CFB*, cada segmento de texto plano $P_j^{\#}$ y segmento de texto cifrado $C_j^{\#}$ consisten de s bits. El valor de s a menudo se incorpora en el nombre del modo, por ejemplo *CFB-1* bit, *CFB-8* bits, *CFB-64* bits, *CFB-128* bits. El modo *CFB* está definido de la siguiente manera:

Para el Proceso de encriptación:

$$\begin{aligned} I_1 &= IV; \\ I_j &= LSB_{b-s}(I_{j-1})|C_{j-1}^{\#} && \text{para } j = 2 \dots n \\ O_j &= CIPH_k(I_j) && \text{para } j = 1,2 \dots n \\ C_j^{\#} &= P_j^{\#} \oplus MSB_s(O_j) && \text{para } j = 1,2 \dots n \end{aligned} \quad (2.5)$$

Para el Proceso de desencriptación:

$$\begin{aligned} I_1 &= IV; \\ I_j &= LSB_{b-s}(I_{j-1})|C_{j-1}^{\#} && \text{para } j = 2 \dots n \\ O_j &= CIPH_k(I_j) && \text{para } j = 1,2 \dots n \\ P_j^{\#} &= C_j^{\#} \oplus MSB_s(O_j) && \text{para } j = 1,2 \dots n \end{aligned} \quad (2.6)$$

Dónde:

- IV : Vector de inicialización
- S : Parámetro entero
- O_j : j -esímo bloque de salida de la función de cifrado.
- C_j : j -esímo bloque de texto de cifrado.
- $CIPH_K$: Función de cifrado.
- P_j : j -esímo bloque de texto plano.
- b : Tamaño del bloque.

En la encriptación CFB, el primer bloque de entrada es el vector de inicialización ($/V$), y la función de cifrado es aplicada a ese vector para producir el primer bloque de salida. El primer segmento de texto cifrado se genera al hacer una operación XOR entre el primer segmento de texto plano y los s bits más significativos del primer bloque de salida. (los restantes $b - s$ bits del primer bloque de salida se descartan). Los $b - s$ bits menos significativos del vector de inicialización son luego concatenados con los s bits del primer segmento de texto cifrado para formar el segundo bloque de entrada. Una descripción alternativa para la formación del segundo bloque de entrada es que los bits del primer bloque de entrada se desplazan circularmente s posiciones a la izquierda, y luego el segmento de texto cifrado reemplaza los s bits menos significativos del resultado.

El proceso se repite con los bloques de entrada sucesivos hasta que se produzca un segmento de texto cifrado para cada segmento de texto plano. En general, cada bloque de entrada sucesivo es cifrado para producir un bloque de salida. Los s bits más significativos de cada bloque de salida se operan en una XOR con el correspondiente segmento de texto plano para formar un segmento de texto cifrado. Cada segmento de texto cifrado (excepto el último) es “realimentado” hacia el bloque de entrada previo, para formar un nuevo bloque de entrada.

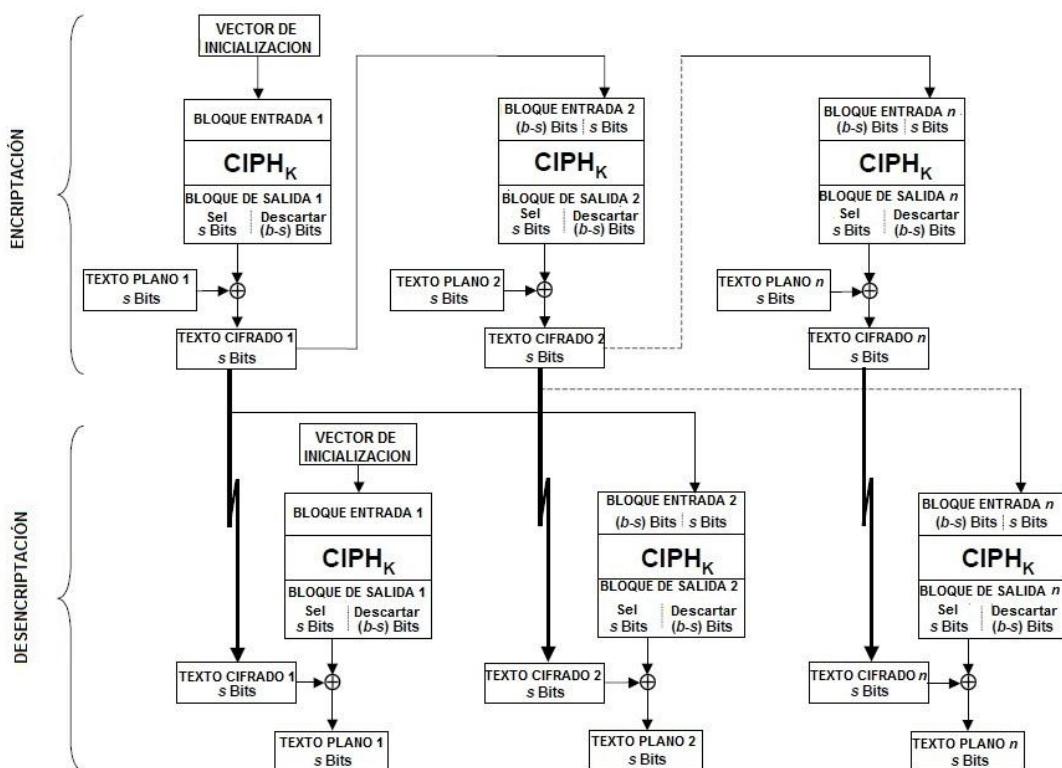


Figura 2.3. Diagrama de bloques para los procesos de encriptación y desencriptación en Modo CFB. [2]

En la desencriptación en modo *CFB*, el vector de inicialización es el primer bloque de entrada y cada bloque de entrada sucesivo es formado como en la encriptación *CFB*, al concatenar los $b - s$ bits menos significativos del bloque de entrada previo con los s bits más significativos del texto cifrado previo. La función de cifrado es aplicada a cada bloque de entrada para producir los bloques de salida. Los s bits más significativos de los bloques de salida son operados en una XOR con los segmentos de texto cifrado correspondiente para recuperar los segmentos de texto plano.

En la encriptación en modo *CFB*, como en la encriptación *CBC*, el bloque de entrada para cada función de cifrado (excepto la primera) depende del resultado de la función de cifrado previa; Por lo tanto, múltiples operaciones de cifrado no se pueden realizar en paralelo.

2.5. Modo *OFB* (*Output Feedback*)

El modo *OFB* es un modo de confidencialidad que realiza la iteración de la función de cifrado con un vector de inicialización para generar una secuencia de bloques de salida que junto con el texto plano son operadas en una XOR para producir el texto cifrado, y viceversa. El vector de inicialización debe ser único para cada ejecución de este modo bajo una clave dada. El modo *OFB* se define de la siguiente manera:

Encriptación en modo *OFB*:

$$\begin{aligned}
 I_1 &= IV; \\
 I_j &= O_{j-1} && \text{para } j = 2 \dots n; \\
 O_j &= CIPH_k(I_j) && \text{para } j = 1,2 \dots n; \\
 C_j &= P_j \oplus O_j && \text{para } j = 1,2 \dots n-1; \\
 C_n^* &= P_n^* \oplus MSB_u(O_n)
 \end{aligned} \tag{2.7}$$

Desencriptación en modo *OFB*:

$$\begin{aligned}
 I_1 &= IV; \\
 I_j &= O_{j-1} && \text{para } j = 2 \dots n; \\
 O_j &= CIPH_k(I_j) && \text{para } j = 1,2 \dots n; \\
 P_j &= C_j \oplus O_j && \text{para } j = 1,2 \dots n-1; \\
 P_n^* &= C_n^* \oplus MSB_u(O_n)
 \end{aligned} \tag{2.8}$$

Dónde:

IV : Vector de inicialización

O_j : j -esimo bloque de salida de la función de cifrado.

$CIPH_K$: Función de cifrado.

P_j : j -esimo bloque de texto plano.

- C_j : j -esimo bloque texto de cifrado.
- C^* : Último bloque de texto cifrado.
- P^* : Último bloque de texto plano.

En la encriptación *OFB*, el vector de inicialización es transformado por la función de cifrado para producir el primer bloque de salida. El primer bloque de salida es operado con el primer bloque de texto plano mediante una XOR para producir el primer bloque de texto cifrado. La función de cifrado es luego invocada en el primer bloque de salida para producir el segundo bloque de salida. El segundo bloque de salida es operado mediante una XOR con el segundo bloque de texto plano para producir el segundo bloque de texto cifrado y la función de cifrado es invocada en el segundo bloque de salida para producir el tercer bloque de salida. Por lo tanto, los bloques de salida sucesivos son producidos al aplicar la función de cifrado a los bloques de salida previos, y los bloques de salida son operados mediante una XOR con los correspondientes bloques de texto plano para producir los bloques de texto cifrado. Para el último bloque, el cual puede ser un bloque parcial de u bits, los bits más significativos de u del último bloque son usados para la operación XOR; los restantes $b-u$ bits del último bloque de salida son descartados.

En la desencriptación en modo *OFB*, el vector de inicialización es transformado por la función de cifrado para producir el primer bloque de salida. El primer bloque de salida es luego transformado por la función de cifrado para producir el segundo bloque de salida. El segundo bloque de salida es operado mediante una XOR con el segundo bloque de texto cifrado para producir el segundo bloque de texto plano, y el segundo bloque de salida es transformado por la función de cifrado para producir el tercer bloque de salida. Por lo tanto los bloques de salida sucesivos son producidos al aplicar la función de cifrado a los bloques previos de salida, y los bloques de salida son operados mediante una XOR con los correspondientes bloques de texto cifrado para recuperar los bloques de texto plano. Para el último bloque, el cual puede ser un bloque parcial de u bits, los bits más significativos de u del último bloque de salida son usados para la operación XOR; los restantes $b-u$ bits del último bloque de salida son descartados.

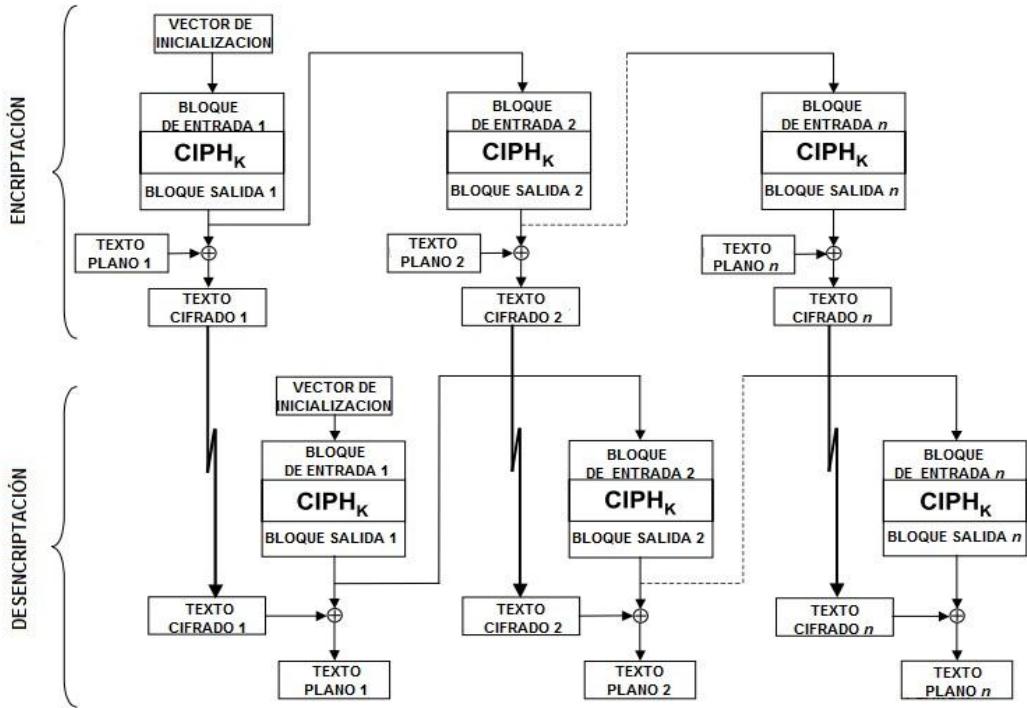


Figura 2.4. Diagrama de bloques para los procesos de encriptación y desencriptación en Modo OFB. [2]

En la encriptación y desencriptación *OFB*, cada función de cifrado (excepto la primera) dependen de los resultados de la función de cifrado previa; por lo tanto, múltiples funciones de cifrado no pueden ser realizadas en paralelo. Sin embargo, si el vector de inicialización es conocido, los bloques de salida pueden ser generados antes de la disponibilidad del texto plano o texto cifrado

2.6. Modo Contador (CTR)

El modo contador es un modo de confidencialidad que realiza la función de cifrado a un conjunto de bloques de entrada llamados contadores, para producir una secuencia de bloques de salida que junto con el texto plano son operados en una XOR para producir el texto cifrado y viceversa. La secuencia de contadores debe tener la propiedad de que cada bloque en la secuencia es diferente de cualquier otro bloque. En todos los mensajes que son encriptados bajo una clave dada, los contadores deben ser distintos. Los contadores para un mensaje dado son denotados como T_1, T_2, \dots, T_n y dada una secuencia de contadores T_1, T_2, \dots, T_n el modo contador se define de la siguiente manera:

Encriptación en modo CTR:

$$O_j = CIPH_k(T_j) \quad \text{para } j = 1, 2, \dots, n;$$

$$\begin{aligned} C_j &= P_j \oplus O_j && \text{para } j = 2 \dots n - 1; \\ C_n^* &= P_n^* \oplus MSB_u(O_n) \end{aligned} \quad (2.9)$$

Desencriptación en modo CTR:

$$\begin{aligned} O_j &= CIPH_k(T_j) && \text{para } j = 1, 2 \dots n; \\ P_j &= C_j \oplus O_j && \text{para } j = 2 \dots n - 1; \\ P_n^* &= C_n^* \oplus MSB_u(O_n) \end{aligned} \quad (2.10)$$

Dónde:

- O_j : j -esímo bloque de salida de la función de cifrado.
- $CIPH_K$: Función de cifrado.
- T_j : j -esímo bloque de contador.
- P_j : j -esímo bloque de texto plano.
- C_j : j -esímo bloque texto de cifrado.
- C^* : Último bloque de texto cifrado.
- P^* : Último bloque de texto plano.

En la encriptación en modo *CTR* la función de cifrado es invocada para cada bloque contador y los bloques resultantes de salida son operados con una XOR con los respectivos bloques de texto plano para producir los bloques de texto cifrado. Para el último bloque, el cual puede ser un bloque parcial de u bits, los u bits más significativos del último bloque de salida son utilizados para la operación XOR; los restantes $b - u$ bits del último bloque de salida son descartados.

En la desencriptación en modo *CTR*, la función de cifrado es invocada para cada bloque contador y los bloques resultantes de salida son operados con una XOR con los respectivos bloques de texto cifrado para recuperar los bloques de texto plano. Para el último bloque, el cual puede ser un bloque parcial de u bits, los u bits más significativos del último bloque de salida son utilizados para la operación XOR; los restantes $b - u$ bits del último bloque de salida son descartados.

En ambos procesos: encriptación *CTR* y desencriptación *CTR*, las funciones de cifrado pueden ser realizadas en paralelo; similarmente, el bloque de texto plano que corresponda a algún bloque de texto cifrado en particular puede ser recuperado independientemente de los otros bloques de texto plano si el correspondiente bloque contador puede ser determinado. Además, las funciones de cifrado pueden ser aplicadas a los contadores antes de la disponibilidad de los datos de texto plano ó texto cifrado.

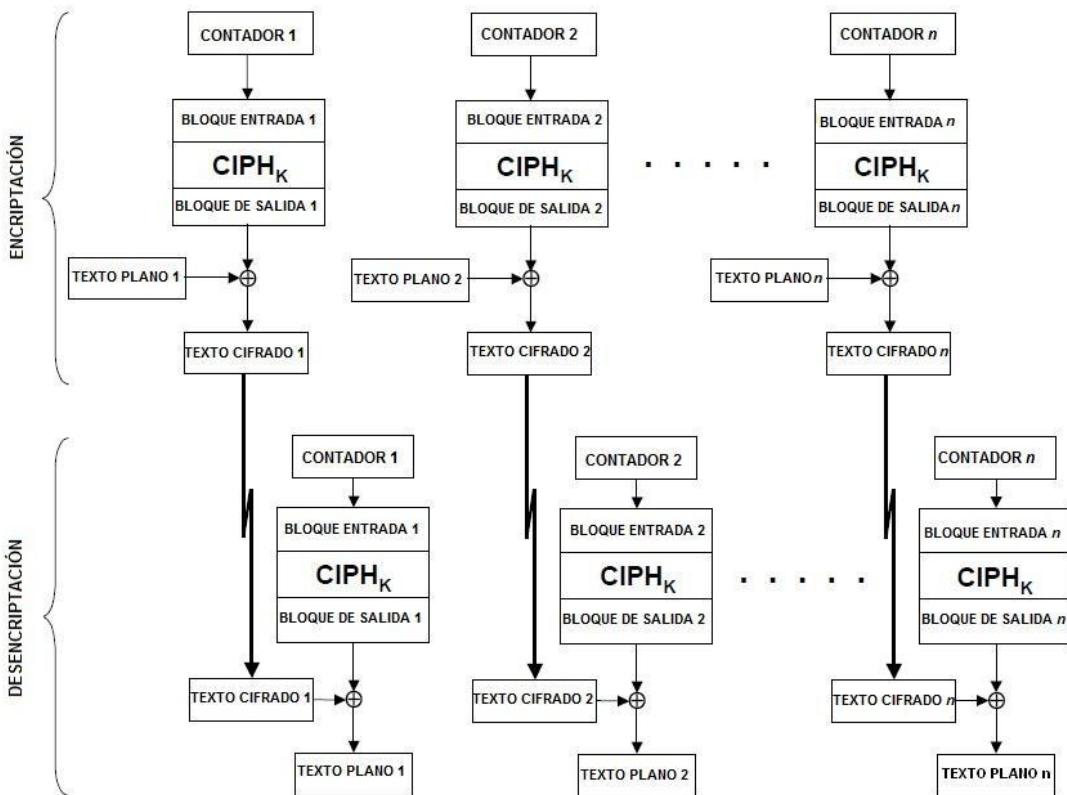


Figura 2.5. Diagrama de bloques para los procesos de encriptación y desencriptación en Modo CTR. [2]

2.6.1. Generación de los bloques contador

La especificación del modo *CTR* requiere un bloque de contador único para cada bloque de texto plano que se encripta bajo una clave dada a través de todos los mensajes. De ser contrario este requerimiento, un bloque contador sería usado repetidamente, entonces la confidencialidad de todos los bloques de texto plano correspondientes a ese bloque de contador puede perderse.

En particular, si algún bloque de texto plano conocido se encripta usando cierto bloque de contador, entonces la salida de la función de cifrado puede ser determinada fácilmente a partir del bloque de texto cifrado asociado. Esta salida permite que bloques que han sido encriptados con el mismo contador sean fácilmente recuperados a partir de sus bloques de texto cifrado asociados.

Otro caso similar es cuando por ejemplo se encriptan dos textos planos bajo el mismo bloque de contador. En este caso se podrían adquirir los dos textos cifrados para realizar una operación XOR entre ellos y así obtener la XOR entre los dos textos planos.

$$C_1 \oplus C_2 = P_1 \oplus P_2 \quad (2.11)$$

Dónde:

C_1 y C_2 : Textos cifrados.
 P_1 y P_2 : Textos Planos.

Hay dos aspectos a satisfacer para el requerimiento de unicidad: Primero, una función incremental para generar los bloques contador a partir de algún bloque contador inicial debe asegurar que los bloques contador no se repitan dentro de un mensaje dado. Segundo, el bloque contador inicial, T_1 , debe ser elegido para asegurar que los contadores sean únicos a través de todos los mensajes que son encriptados bajo una clave dada.

2.6.2. La función incremental estándar

En general, dado el bloque de contador inicial para un mensaje, los bloques de contador sucesivos se derivan al aplicar una función incremental.

El número de bloques en un mensaje de texto plano dado será denotado como n y el numero de bits en el bloque será denotado como b .

La función incremental estándar puede aplicarse a un bloque entero o a una parte de un bloque. Sea m el numero de bits en la parte específica del bloque a ser incrementado; es decir que m es un entero positivo tal que $m \leq b$. Cualquier palabra de m bits puede ser considerada como la representación binaria de un entero no negativo x que es estrictamente menor que 2^m .

Por ejemplo, La función incremental estándar se aplica a los cinco bits menos significativos de bloques de ocho bits, así que $b = 8$ y $m = 5$; sea que * represente cada bit desconocido en este ejemplo, y sea que *** 11110 represente un bloque a ser incrementado. La siguiente secuencia de bloques resultan de cuatro aplicaciones de la función incremental estándar:

```

*** 1 1 1 1 0
*** 1 1 1 1 1
*** 0 0 0 0 0
*** 0 0 0 0 1
*** 0 0 0 1 0

```

Los bloques contador en los cuales un conjunto dado de m bits son incrementados por la función incremental estándar satisfacen el requerimiento de unicidad dentro de un mensaje dado con la condición de que $n \leq 2^m$.

2.6.3. Elección del bloque de contador inicial

El bloque de contador inicial, T_1 , para cada mensaje que se encripta bajo una clave se debe elegir de tal manera que asegure la unicidad de todos los bloques de contador a través de todos los mensajes. Dos ejemplos de aproximaciones para elegir el bloque contador inicial son dados a continuación:

En la primera aproximación, para una clave dada, todos los mensajes de texto plano se encriptan secuencialmente. Para todos los mensajes, el mismo conjunto fijo de m bits del bloque de contador se incrementa por la función incremental estándar. El bloque de contador inicial para el mensaje de texto plano inicial puede ser cualquier palabra de b bits.

El bloque de contador inicial para cualquier mensaje posterior se puede obtener al aplicar la función incremental estándar al conjunto fijo de m bits del bloque de contador final del mensaje previo. En efecto, todos los mensajes de texto plano que se encriptan bajo una clave, se concatenan en un solo mensaje; consecuentemente, el número total de bloques de texto plano no deben exceder 2^m .

Una segunda aproximación para satisfacer la propiedad de unicidad a través de los mensajes es asignar a cada mensaje un *string* único de $b/2$ (aproximando si b es impar) e incorporar el *string* único a cada bloque de contador para el mensaje. Los $b/2$ bits más significativos (aproximando si b es impar) de cada bloque de contador sería el *string* único y la función incremental estándar sería aplicada a los restantes m bits para proporcionar un índice a los bloques de contador para el mensaje. Por lo tanto, si N es el *string* único para un mensaje dado, entonces el j -ésimo bloque de contador está dado por $T_j = N | [j]_m$, para $j = 1 \dots n$. El numero de bloques, n , en cualquier mensaje debe satisfacer $n < 2^m$.

CAPÍTULO 3: DISEÑO Y SIMULACIÓN

Este capítulo presenta las organizaciones circuitales que componen el algoritmo de *Rijndael* utilizando bloques y claves de 128 bits (*AES-128*). Los diseños hardware se han realizado para los procesos de encriptación y desencriptación en los modos de operación no realimentados (*ECB* y *CTR*), los cuales admiten configuraciones segmentadas (*pipeline*).

Todos los módulos han sido simulados, probados e implementados de acuerdo a los estándares *FIPS-197*[1] y *SP800-38A* [2]. En el diseño se ha utilizado el lenguaje de descripción de hardware *VHDL* [4] y el software de síntesis e implementación de sistemas digitales *Xilinx ISE 13.1*. [5].

3.1. Introducción

El algoritmo *AES* se compone de cuatro transformaciones que tienen lugar en cada una de las rondas del algoritmo. Para el caso del cifrado (encriptación), las transformaciones son: *SubBytes*, *ShiftRows*, *MixColumns*, *AddRoundKey* y para el descifrado (desencriptación) las transformaciones son: *InvSubBytes*, *InvShiftRows*, *InvMixColumns*, *AddRoundKey*. El bloque de expansión de claves incluye tres transformaciones: *Rcon*, *RotWord* y *SubWord*.

En general, el algoritmo *AES* se puede utilizar en todos los modos de operación expuestos por [2], sin embargo los modos *ECB* Y *CTR* presentan la ventaja de que pueden ser totalmente paralelizables tanto en el proceso de encriptación como en el de desencriptación.

3.2. Transformación *SubBytes*

Se han usado dos aproximaciones para diseñar esta función en hardware: la primera consiste en construir un único circuito cuya relación entrada/salida sea equivalente a la *S-box*. Con este método se logran implementaciones rápidas y se requieren dispositivos de memoria para almacenar los valores de sustitución del *S-box*.

La segunda aproximación usada consiste en diseñar un circuito que realice la operación matemática en la cual se debe calcular el inverso multiplicativo en el campo finito $GF(2^8)$ para posteriormente calcular la transformación afín. En este caso se debe diseñar un circuito para cada operación y conectarlos entre sí. [6]

3.2.1. Transformación *SubBytes* basada en tablas

Esta transformación se realiza utilizando 16 memorias *ROM* de $256 * 8$, es decir, en cada memoria se encuentran los 256 valores de sustitución del *S-box*, para que al

llegar el dato de 16 bytes a la entrada de este módulo sean reemplazados todos los valores al mismo tiempo. Diseños similares se pueden encontrar en [7], [8] y [9]. La Figura 3.1 ilustra el funcionamiento de este módulo:

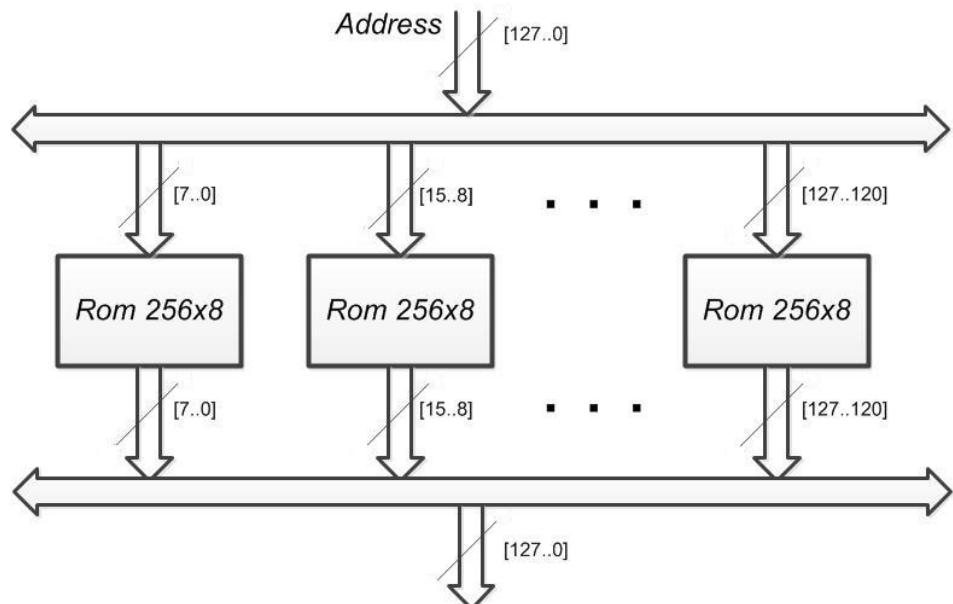


Figura 3.1. Organización interna del módulo *SubBytes*.

Se puede observar que el módulo *SubBytes* no posee señal de reloj; esto se hace con el fin de obtener mayor velocidad sin que sea necesario un ciclo de reloj entero para procesar la matriz de estado.

Usando los vectores ejemplo dados por [1] se realiza la simulación del módulo. Se puede comprobar que los resultados de salida son iguales a los obtenidos en el estándar.

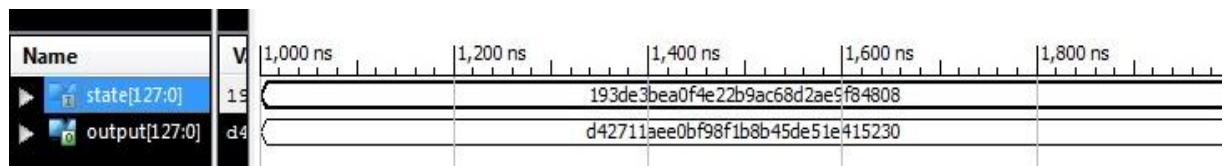


Figura 3.2. Simulación del módulo *SubBytes* basado en tablas.

3.2.2. Transformación *SubBytes* basada en el modelo matemático

Esta transformación utiliza dos módulos para realizar las operaciones matemáticas, el primer módulo se encarga de calcular el inverso multiplicativo y el segundo módulo se encarga de realizar la transformación afín, por lo tanto al conectar los dos módulos en cascada se puede transformar un byte [10]. El diagrama de bloques de la figura 3.3 ilustra la idea general para realizar esta transformación.

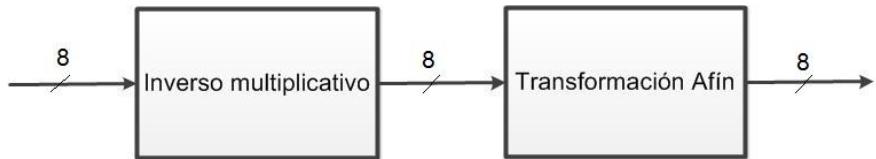


Figura 3.3. Diagrama de bloques de la transformación *SubBytes* basada en el modelo matemático para 8 bits. [6]

Una vez realizada la descripción hardware, el circuito se simula utilizando un valor de prueba dado por la tabla S-Box (figura 1.5). La figura 3.4 muestra el resultado de simulación cuando se toma como valor de prueba el byte “01” a la entrada del circuito. Como se puede observar el resultado concuerda con la tabla S-Box.

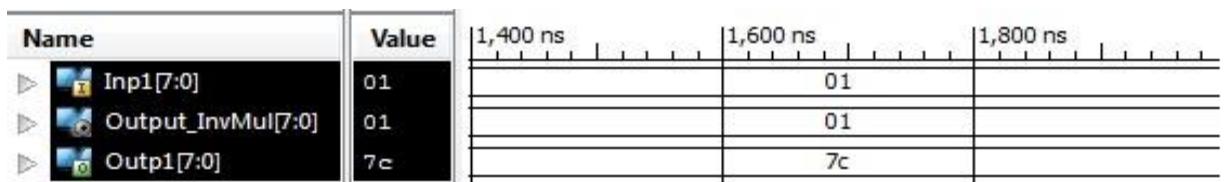


Figura 3.4. Simulación del módulo *SubBytes* para 8 bits basada en el modelo matemático.

Para aplicar la transformación *SubBytes* a un bloque de 128 bits de forma paralela se necesitan 16 bloques conectados en paralelo tal como se muestra en el diagrama de la figura 3.5. Así un bloque de 128 bits se procesa de forma rápida.

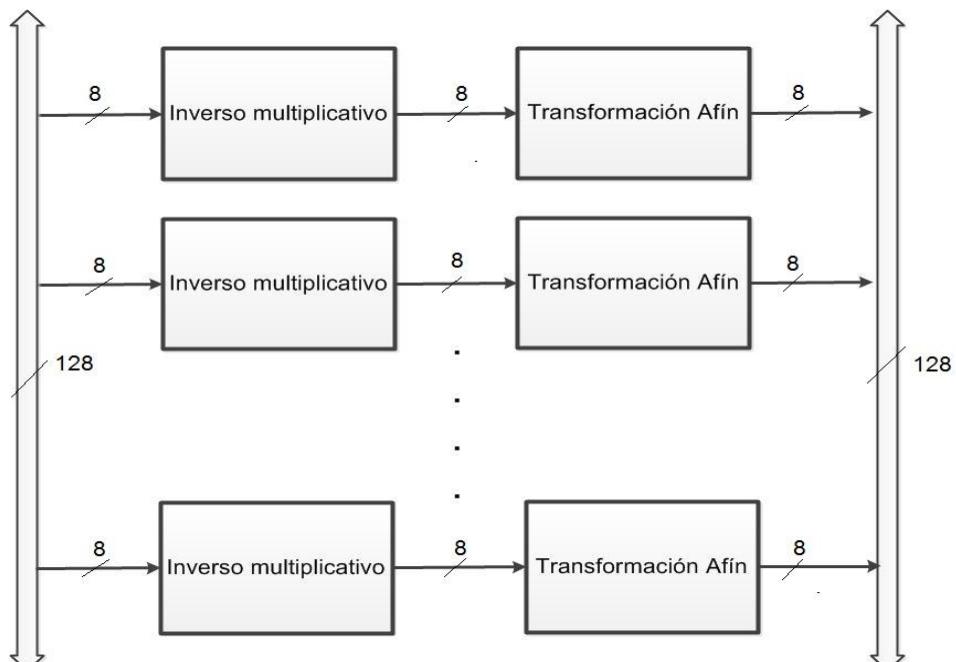


Figura 3.5. Diagrama de bloques para la transformación *SubBytes* para 16 bytes (128 bits). [6]

La figura 3.6 muestra los resultados obtenidos para el vector de entrada sugerido por el estándar:

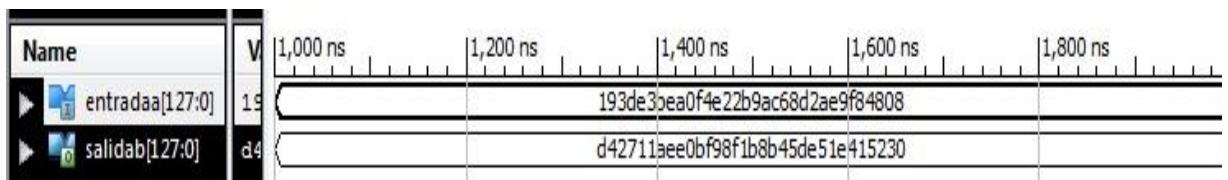


Figura 3.6. Simulación de la transformación *SubBytes* basada en la operación matemática para 128 bits.

3.3. Transformación *ShiftRows*

La transformación *ShiftRows* se puede implementar de manera muy sencilla en hardware mediante líneas de interconexión sin necesidad de componentes adicionales, de esta manera se puede cambiar la posición de cada byte del estado. La figura 3.7 ilustra este procedimiento; mientras que la figura 3.8 muestra los resultados de simulación.

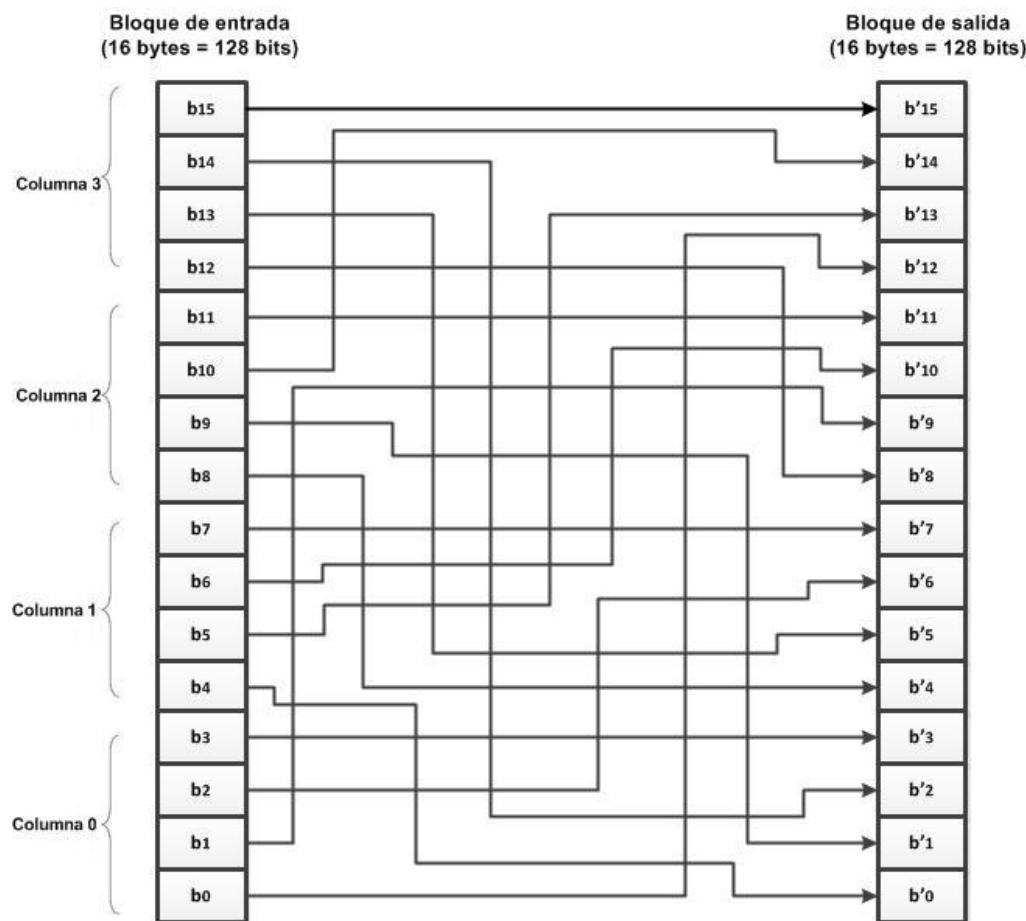


Figura 3.7. Diagrama de circuito para realizar la transformación *ShiftRows*.

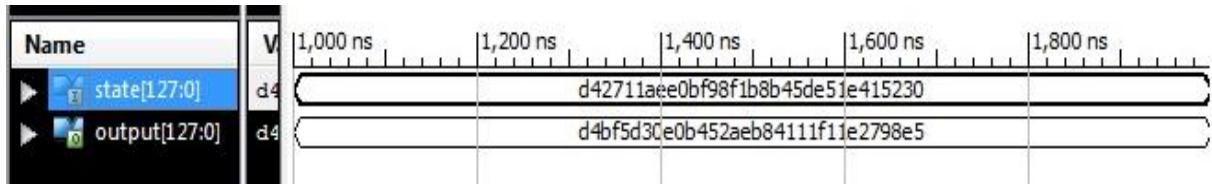


Figura 3.8. Simulación del módulo *ShiftRows*.

3.4. Transformación *MixColumns*

Los autores del algoritmo de *Rijndael* han propuesto la función *xtime()* para la multiplicación de polinomios en $GF(2^8)$ que simplifica la multiplicación de un polinomio por x , Gracias a ello las multiplicaciones de potencias más altas de x se pueden realizar descomponiendo uno de los operandos y aplicando la función *xtime()* de forma sucesiva.

La función *xtime()* consiste en aplicar un desplazamiento a la izquierda al valor que representa el polinomio además de una operación XOR con $(1B_{16})$, según lo expuesto en la sección 1.2. [6], [10]

El siguiente diagrama de flujo representa esta operación:

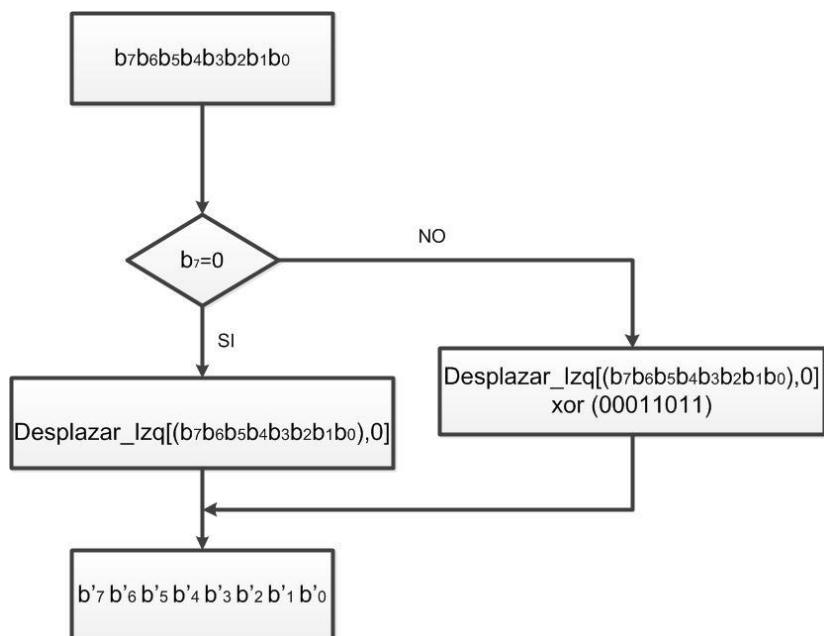


Figura 3.9. Diagrama de flujo para el cálculo de *xtime()*.

Una manera de determinar el circuito que realiza la operación *xtime()* es teniendo en cuenta la multiplicación de un polinomio $b(x)$ por x . Sea $b'(x)$ el resultado de aplicar la función *xtime()* a $b(x)$, se tienen las siguientes expresiones:

$$b' = \text{xtime}(b) = (b(x) * x) \bmod m(x). \quad (3.1)$$

$$\begin{aligned} \text{xtime}(b) &= (b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \bmod \\ &(x^8 + x^4 + x^3 + x + 1) \end{aligned} \quad (3.2)$$

$$b'(x) = b_6x^7 + b_5x^6 + b_4x^5 + (b_7 + b_3)x^4 + (b_7 + b_2)x^3 + b_1x^2 + (b_7 + b_0)x + b_7 \quad (3.3)$$

De (3.3) se deduce que:

$$b'_7 = b_6, \quad b'_6 = b_5, \quad b'_5 = b_4, \quad b'_4 = (b_7 + b_3),$$

$$b'_3 = (b_7 + b_2)b'_2 = b_1, b'_1 = (b_7 + b_0), b'_0 = b_7$$

$$b'(x) = b'_7x^7 + b'_6x^6 + b'_5x^5 + b'_4x^4 + b'_3x^3 + b'_2x^2 + b'_1x + b'_0 \quad (3.4)$$

Con estos términos deducidos es posible determinar el circuito que ejecuta la función `xtime()` a nivel de bits. Como se puede apreciar en la figura 3.10, solo se requieren tres compuertas XOR y algunas líneas de interconexión.

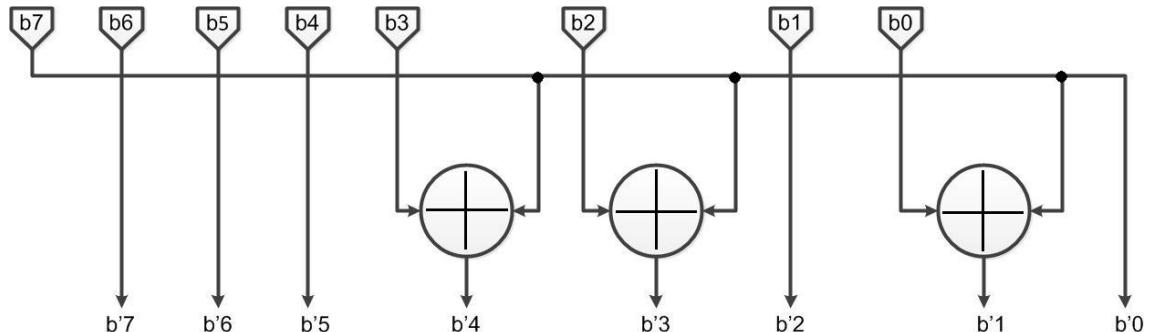


Figura 3.10. Circuito que ejecuta la función `xtime()` .[6]

Usando el circuito que ejecuta la operación `xtime()` es posible implementar fácilmente el circuito que realiza la transformación de *MixColumns* para 128 bits utilizando cuatro veces el módulo que se muestra en la figura 3.11. Cada módulo recibe como entrada una columna de la matriz de estado y genera las salidas correspondientes, se puede observar que esta transformación se implementa mediante la interconexión de puertas XOR. La figura 3.12 ilustra el diagrama de tiempos obtenido en la simulación de la función *MixColumns*.

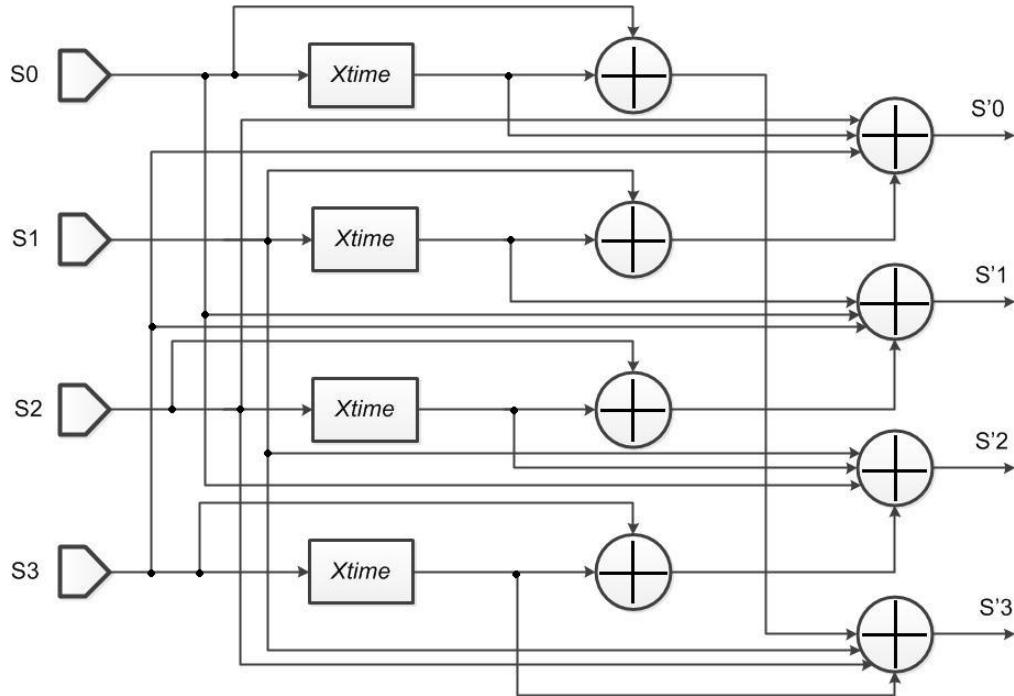


Figura 3.11. Diagrama de circuito de la transformación *MixColumns*. [6]

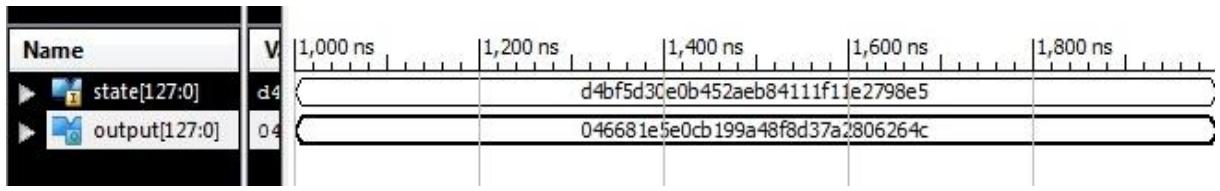


Figura 3.12. Simulación del módulo *MixColumns*.

3.5. Transformación *AddRoundKey*

Esta transformación es muy sencilla y consiste en realizar una operación XOR entre una clave de ronda y la matriz de estado, tal como se muestra en la figura 3.13. La figura 3.14 ilustra un resultado de esta transformación.

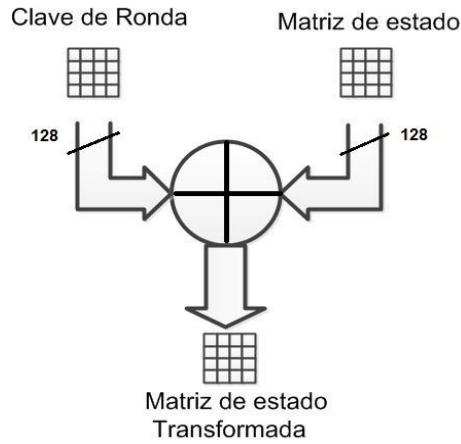


Figura 3.13. Transformación *AddRoundKey*.

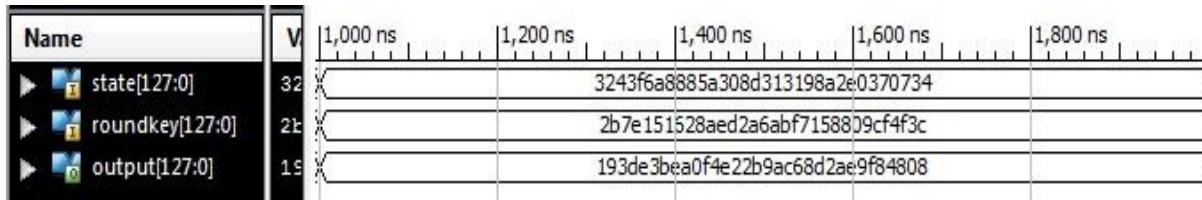


Figura 3.14. Simulación del módulo *AddRoundKey*.

3.6 Transformación *InvSubBytes*

3.6.1 *InvSubBytes* basada en tablas

De igual forma que en la transformación *SubBytes* basada en tablas, la transformación *InvSubBytes* se implementa utilizando 16 memorias de 256×8 , es decir, en cada memoria se encuentran los 256 valores de sustitución de la *S-Box* inversa, para que al llegar el dato de 16 bytes a la entrada de este módulo sean reemplazados todos los valores al mismo tiempo. La Figura 3.15 ilustra el circuito para implementar esta función.

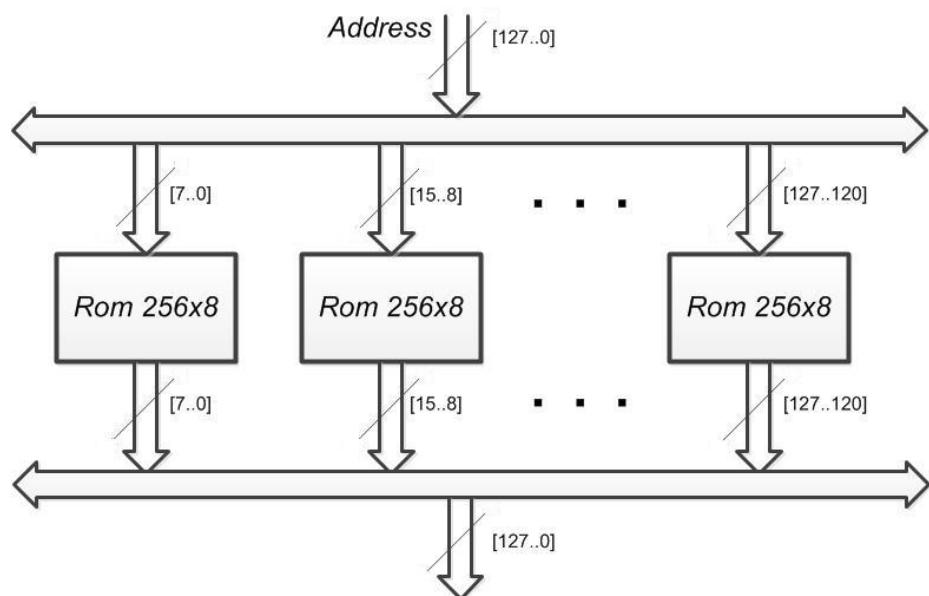


Figura 3.15. Organización circuital para el módulo *InvSubBytes*.

En la figura 3.16 se observa el resultado de la simulación usando el vector de entrada usado en el estándar.

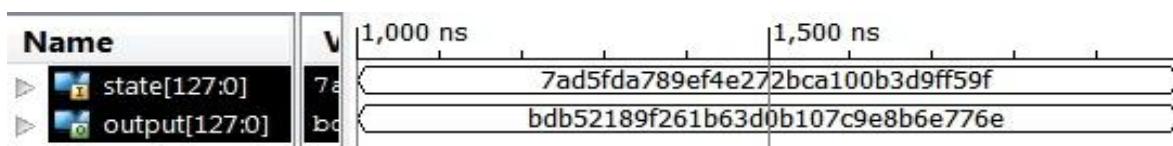


Figura 3.16. Simulación del módulo *InvSubBytes*.

3.6.2. InvSubBytes basada en el modelo matemático

Esta transformación consta de dos operaciones matemáticas, la primera se encarga de calcular la inversa de la transformación Afín y la segunda se encarga de calcular el inverso multiplicativo. Cada operación se implementa en hardware separado y luego al conectar los dos módulos en serie se obtiene el resultado. La figura 3.17 ilustra el diagrama de bloques para realizar esta transformación:

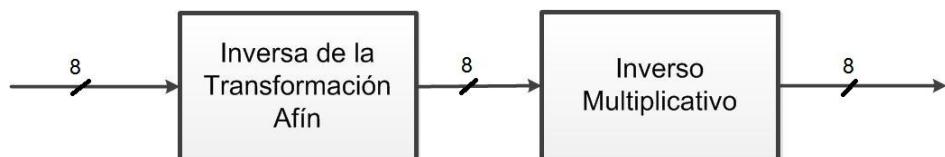


Figura 3.17. Diagrama de bloques de la transformación *InvSubBytes* basada en el modelo matemático para 8 bits. [6]

Para aplicar la transformación *InvSubBytes* a un bloque de 128 bits se necesitan 16 bloques en paralelo como los mostrados en la figura 3.17, por lo tanto puede obtenerse un circuito como el mostrado en la figura 3.18, de esta manera un bloque de 128 bits puede ser transformado en paralelo.

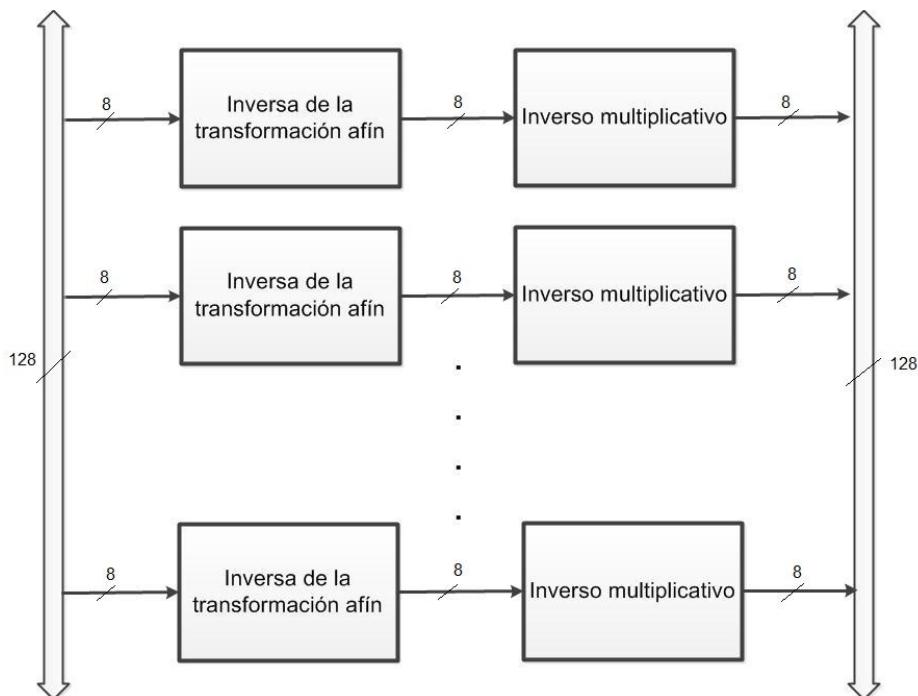


Figura 3.18. Diagrama de circuito de la transformación *InvSubBytes* para 16 bytes basada en el modelo matemático. [6]

3.7. Transformación *InvShiftRows*

Similar a la transformación *ShiftRows*, esta transformación se puede implementar en hardware comutando líneas de interconexión sin necesidad de utilizar otros componentes adicionales de hardware; de esta manera, se puede cambiar la posición de cada byte del estado según lo expuesto en la sección 1.4.1. La figura 3.19 ilustra este procedimiento.

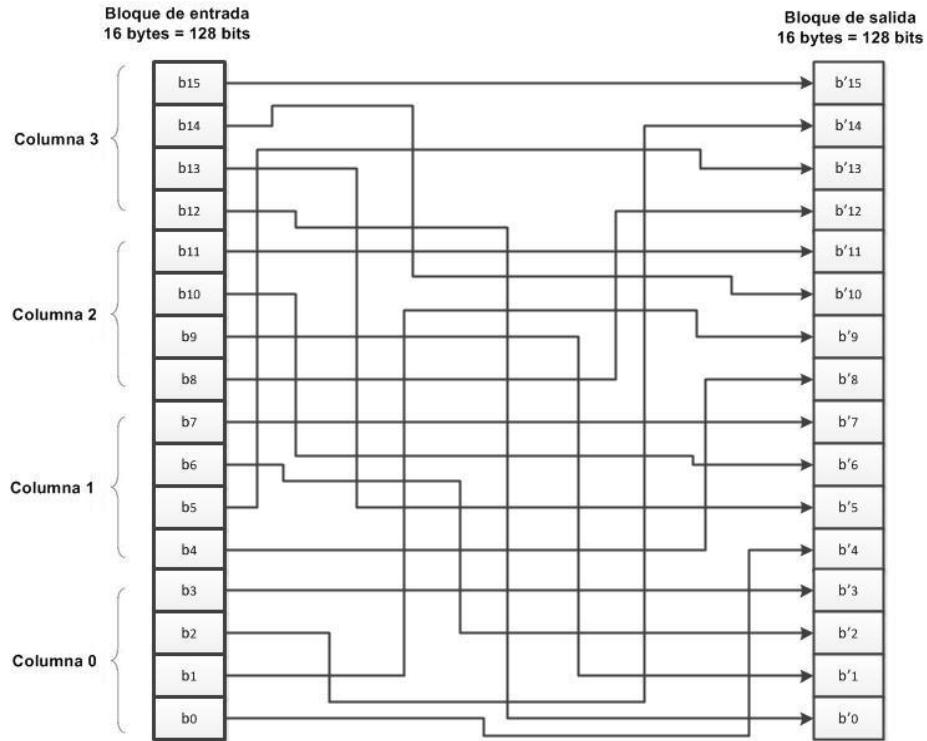


Figura 3.19. Diagrama de circuito para realizar la transformación *InvShiftRows*.

La figura 3.20 ilustra los resultados de simulación para esta función.

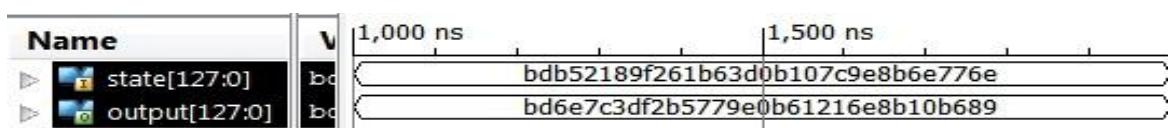


Figura 3.20. Simulación del módulo *InvShiftRows*.

3.8. Transformación *InvMixColumns*

La transformación *InvMixColumns* consiste en la multiplicación de cada una de las columnas de la matriz de estado por una matriz cuyos componentes son los valores {0E}, {09}, {0D}, {0B}. Según lo expuesto en la sección 1.2 se puede reescribir la multiplicación de dos elementos como una combinación lineal de productos, por ejemplo: se puede expresar $x * \{0E\} = x * \{08\} \oplus x * \{04\} \oplus x * \{02\}$ para cualquier $x \in GF(2^8)$. Teniendo en cuenta esta propiedad y utilizando la función *xtime()* se

puede implementar un circuito como el mostrado en la figura 3.21, el cual multiplica un byte B_0 por las constantes: {0E}, {09}, {0D}, {0B}.

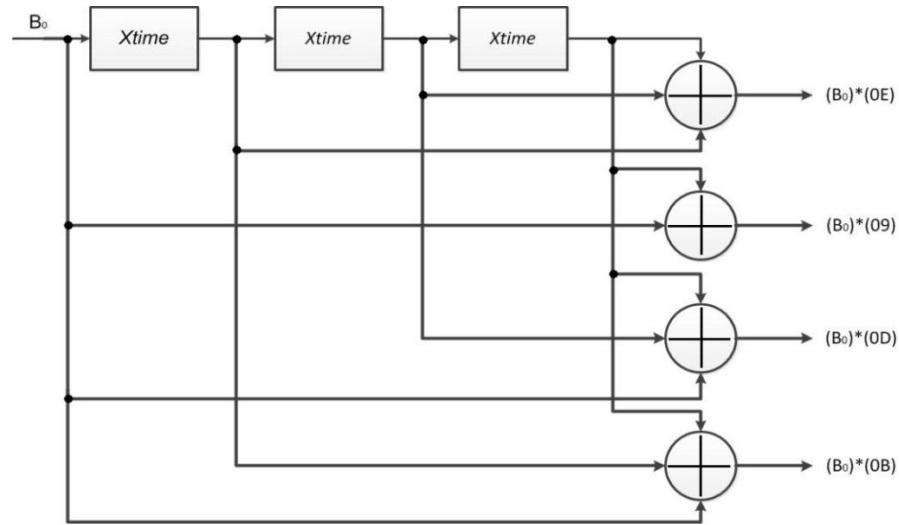


Figura 3.21. Multiplicador para la transformación *InvMixColumns*. [6]

Para transformar una columna de la matriz de estado se requieren cuatro circuitos multiplicadores como el mostrado en la figura 3.21, de tal manera que se multiplique cada byte de la columna en paralelo por los valores {0E}, {09}, {0D}, {0B} y luego sumar los valores correspondientes según lo expuesto en la sección 1.4.3. La figura 3.22 ilustra el circuito que transforma una columna de la matriz de estado:

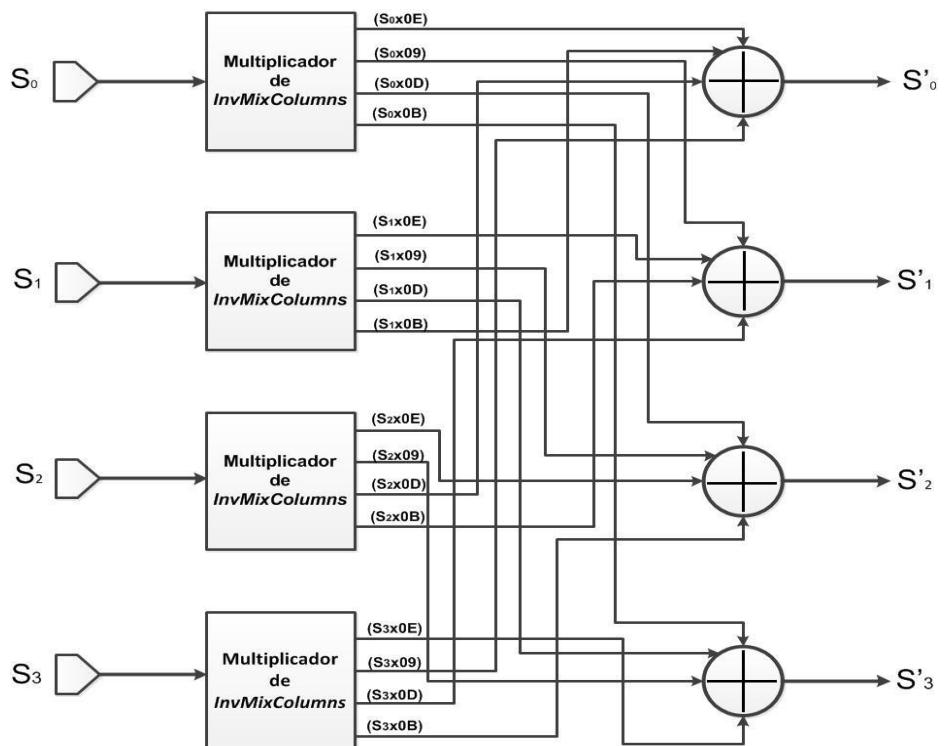


Figura 3.22. Diagrama de circuito para la transformación *InvMixColumns*. [6]

Para transformar toda la matriz de estado (de cuatro columnas), se requiere usar cuatro circuitos en paralelo. Utilizando los vectores de prueba del estándar [1] se obtiene el resultado de simulación mostrado en la figura 3.23:

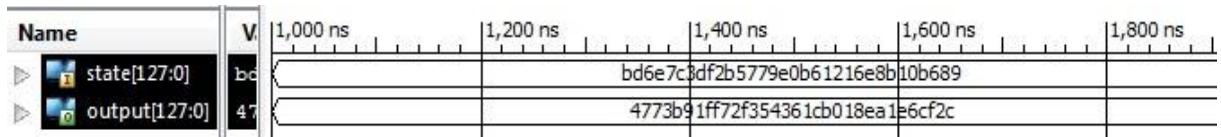


Figura 3.23. Simulación del módulo *InvMixColumns*.

3.9. Inversa de la transformación *AddRoundKey*

La transformación *AddRoundKey* descrita en la sección 1.3.4 es su propia inversa ya que solo involucra una operación XOR.

3.10. Unidad de subclaves

En esta sección se presentan dos diseños hardware para la implementación de la unidad de Subclaves: una arquitectura iterativa y una arquitectura en cascada.

La unidad básica de subclaves ha sido diseñada de acuerdo con la especificación del estándar [1] y se muestra en la figura 3.24.

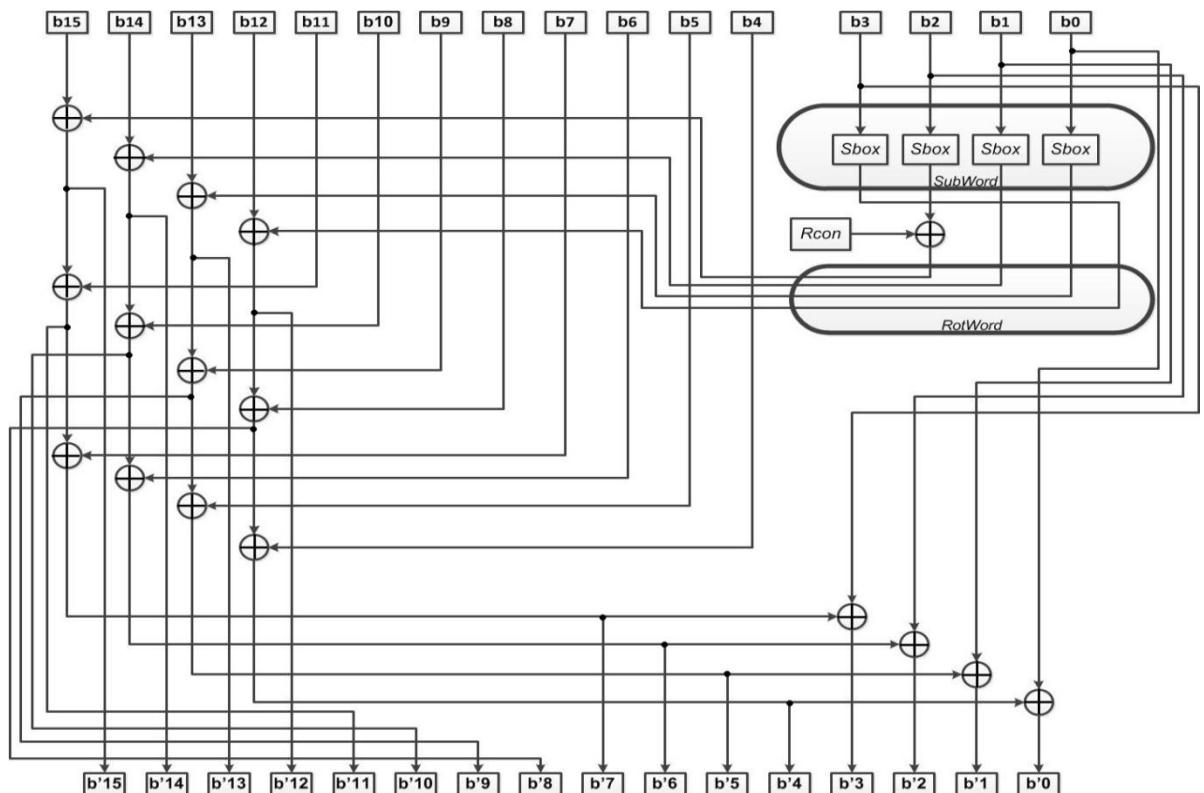


Figura 3.24. Circuito de expansión de clave de ronda. [6]

En la figura 3.24 se observa que los bytes menos significativos de la matriz de estado (b_3, b_2, b_1, b_0) por ser múltiplo de N_k , se transforman para calcular la palabra $w[i]$. Se empieza con la transformación *SubWord()* (equivalente a la transformación *SubBytes*), la cual se implementa mediante cuatro memorias *ROM* de $256*8$ en las cuales se almacenan los valores de la tabla *S-Box*. Posteriormente se realiza la transformación *RotWord()* (equivalente de la transformación *ShiftRows*) la cual se implementa cambiando el bus que lleva el byte más significativo de la salida de la transformación *Subword()* a la posición de byte menos significativa. Una vez hechas estas transformaciones se aplica la transformación *Rcon* consistente en una operación XOR con un valor constante *Rcon* según un número de ronda correspondiente. La tabla 3.1 indica los valores de *Rcon* constantes asociados con cada ronda.

Tabla 3.1. Tabla que contiene las constantes *Rcon* de acuerdo con el número de ronda.

Número de ronda	Valor Rcon (Hex)
1	01
2	02
3	04
4	08
5	10
6	20
7	40
8	80
9	1B
10	36

En el diseño se puede utilizar una memoria ROM para almacenar las constantes de ronda *Rcon* de tal manera que se sincronize el sistema para la lectura de cada constante en la ronda respectiva. Una vez realizadas las transformaciones {*SubWord*, *RotWord*, *Rcon*}, el byte resultante se suma a la palabra que está a N_k posiciones más cercanas, o sea con la palabra ($b_{15}, b_{14}, b_{13}, b_{12}$) para obtener la primera columna resultante.

De acuerdo a la figura 3.24, las columnas restantes se transforman teniendo en cuenta que cada palabra siguiente $w[i]$ es resultado de la operación XOR entre la palabra previa $w[i - 1]$ y la palabra que está N_k posiciones más cercana.

Este proceso se debe repetir diez veces para generar las subclaves para cada una de las diez rondas. Al incluir la clave inicial se tiene un total de once subclaves que equivalen a 44 palabras de 32 bits.

3.10.1. Unidad de subclaves iterativa

La unidad de subclaves iterativa utiliza una unidad de control que se encarga de asignar las señales necesarias por cada ciclo de reloj al módulo de procesamiento de datos ó ruta de datos de la unidad de subclaves, para que la clave inicial de entrada sea expandida de tal manera que se genere una subclave por cada ciclo de reloj. La figura 3.25 ilustra el diagrama de circuito general de la unidad de subclaves iterativa:

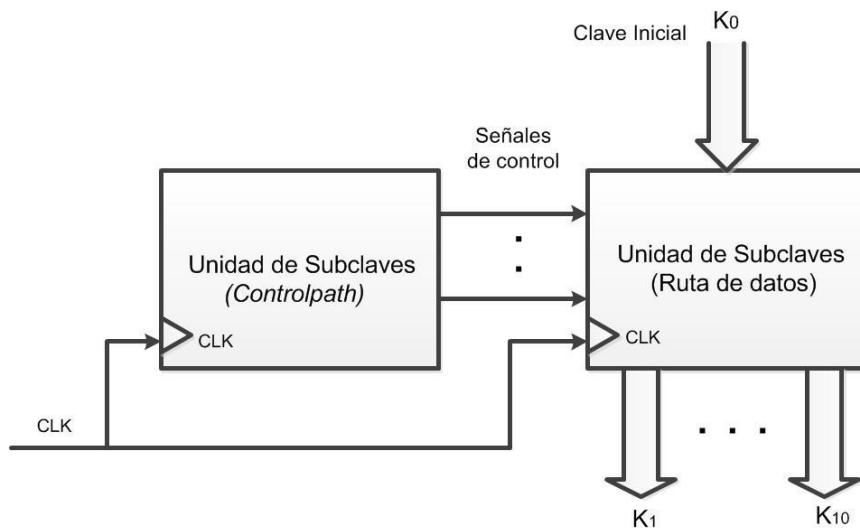


Figura 3.25. Diagrama de circuito general de la unidad de subclaves iterativa.

La ruta de datos está conformada por un circuito de expansión de clave de ronda (Figura 3.24), una memoria *Rom* de 10×8 que almacena los diez valores de las constantes *Rcon*, varios registros de 128 bits y un multiplexor dos a uno de 128 bits que selecciona la clave inicial ó un dato a iterar, la figura 3.26 ilustra este esquema:

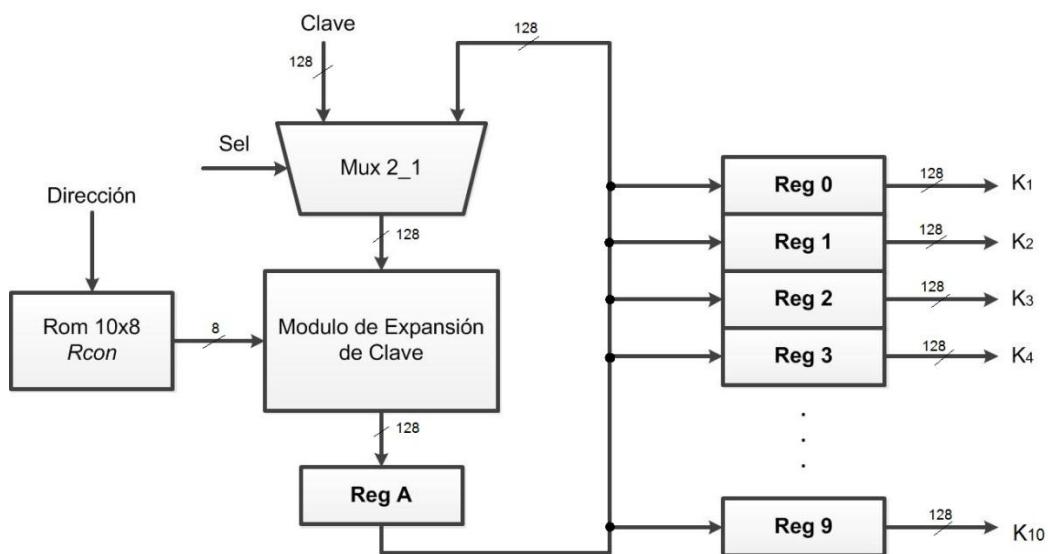


Figura 3.26. Ruta de datos de la unidad de subclaves iterativa.

Inicialmente la unidad de control activa las señales necesarias para que la clave inicial k_0 ingrese al módulo de expansión de clave y el resultado se almacene en el registro Reg A y en el registro Reg0. Una vez calculada la primera subclave, esta ingresa de nuevo al módulo de expansión de clave de ronda para calcular la siguiente subclave de ronda y así sucesivamente hasta calcular las diez subclaves.

La unidad de subclaves hace los cálculos de manera iterativa, así que para cada ciclo de reloj el valor del registro Reg A se almacena en un registro Reg i y se ingresa al módulo de expansión de clave. También se lee la posición de memoria ROM de acuerdo con el número de ronda, para que el resultado quede disponible en el siguiente ciclo de reloj y continuar así con la iteración.

La simulación funcional de la unidad de subclaves usando los vectores de prueba dados por [1] se muestra en la figura 3.27.

La clave a expandir es la siguiente:

Clave = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

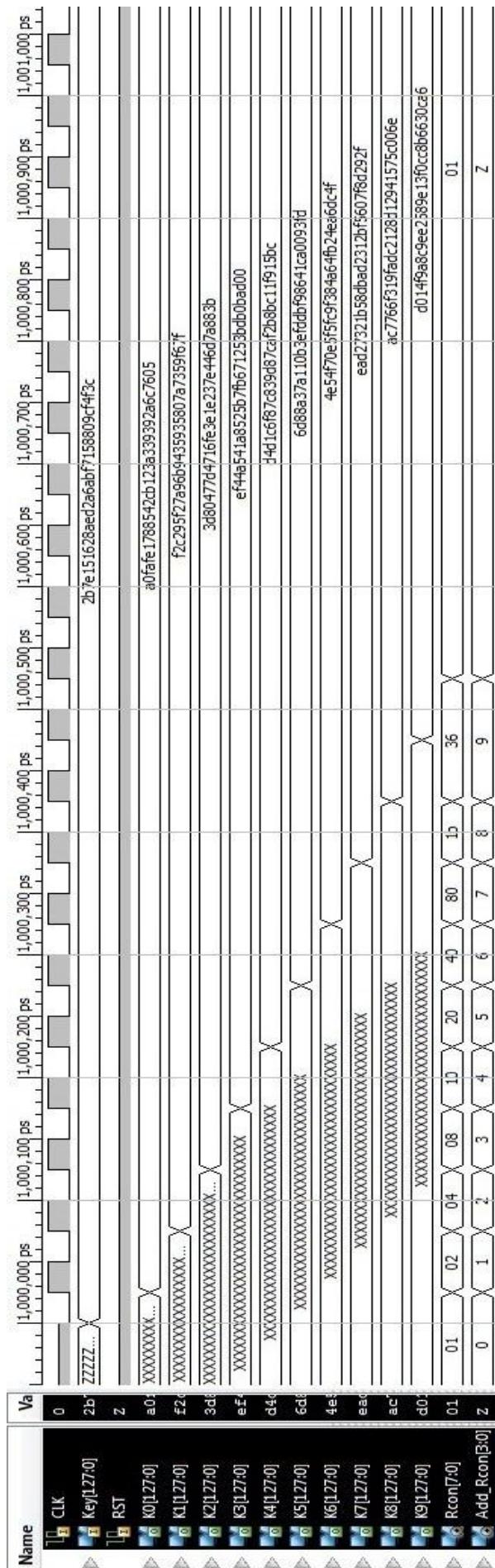


Figura 3.27. Resultados de simulación funcional de la unidad de subclaves iterativa.

La figura 3.27 muestra que por cada flanco de reloj se lee una posición de la memoria que almacena las constantes $Rcon$ y a la vez una nueva subclave es generada; por lo tanto, se requieren diez ciclos de reloj para generar las diez subclaves necesarias en el proceso de encriptación o desencriptación.

3.10.2. Unidad de subclaves en cascada

Esta unidad de subclaves utiliza diez circuitos de expansión de clave de ronda (figura 3.24) conectados en cascada de tal manera que cada módulo de expansión de clave de ronda calcula una subclave. La figura 3.28 ilustra la unidad de subclaves en cascada:

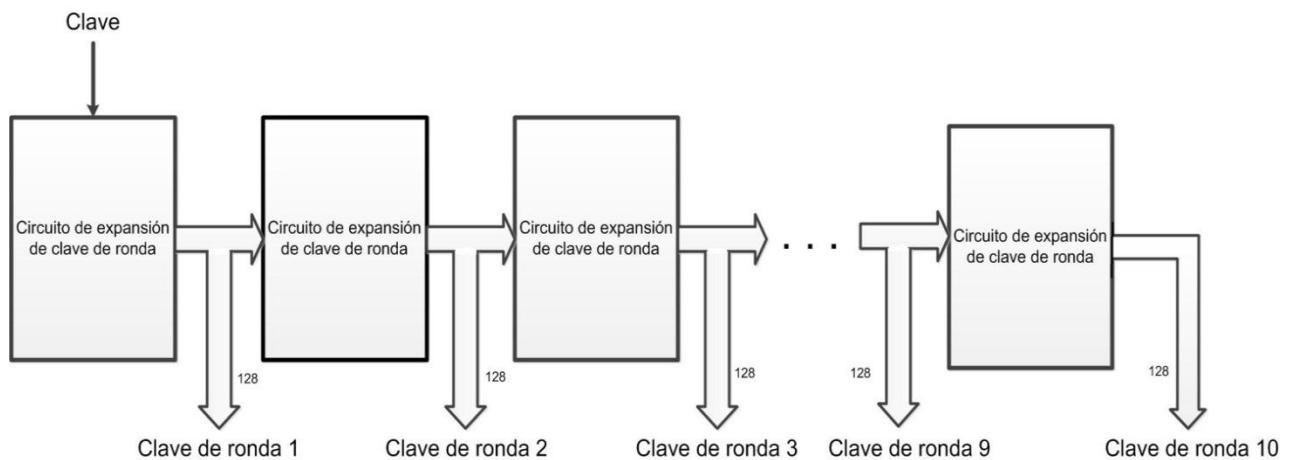


Figura 3.28. Unidad de subclaves en cascada.

En esta unidad de subclaves, cada circuito de expansión de clave de ronda utiliza cuatro memorias *ROM* de $256*8$ (*S-Box*) y el conjunto de puertas XOR para llevar a cabo las operaciones entre las columnas de la matriz de estado, es decir que en total se utilizan cuarenta memorias *ROM* de $256*8$ para realizar la transformación *subword()*.

Debido a que cada circuito de expansión de clave de ronda siempre tendrá asociado un valor constante de la tabla $Rcon$, la transformación $Rcon$ de un byte se puede llevar a cabo negando los bits en las posiciones que concuerden con las posiciones de bit de la constante $Rcon$ donde haya un uno lógico, por lo tanto cada circuito de expansión de clave de ronda incluye su propio circuito combinacional que realiza la transformación $Rcon$.

De otro lado, se puede notar que en este esquema no es necesario utilizar una señal de reloj debido a que no es necesario sincronizar señales, por lo tanto es un módulo mucho más rápido que la unidad de subclaves iterativa. Una vez la clave se ingresa a esta unidad, la primera subclave se genera en el primer módulo e inmediatamente el resultado ingresa al siguiente módulo para generar la segunda subclave y así sucesivamente hasta generar las diez subclaves.

La figura 3.29 muestra los resultados de la simulación para el siguiente vector de entrada:

Clave = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Name	Value	1,000 ns	1,200 ns	1,400 ns	1,600 ns	1,800 ns
key[127:0]	2b		2b7e15	1628aed2a6abf7158809	cf4f3c	
en	1					
rst	0					
roundkey0[127:0]	2b		2b7e15	1628aed2a6abf7158809	cf4f3c	
roundkey1[127:0]	a0			a0fafef88542cb123a339392a6c7605		
roundkey2[127:0]	f2			f2c295f27a96b9435935807a7359f67f		
roundkey3[127:0]	3d			3d80477d4716fe3e1e237e446d7a883b		
roundkey4[127:0]	ef			ef44a541a8525b7fb671253bdb0bad00		
roundkey5[127:0]	d4			d4d1cf87c839d87caf2b8bc11915bc		
roundkey6[127:0]	6d			6d88a37a110b3efddbf98641ca0093fd		
roundkey7[127:0]	4e			4e54f70e5f5fc9f384a64fb24ea6dc4f		
roundkey8[127:0]	ea			ead27321b58dbad2312bf5607f9d292f		
roundkey9[127:0]	ac			ac7766f319fadcc2128d12941575c006e		
roundkey10[127:0]	d0			d014f9a8c9ee2589e13f0cc8b6630ca6		

Figura 3.29. Resultados de simulación de la unidad de subclaves en cascada.

3.11. Bloque Ronda

El Bloque Ronda se usa en este caso para agrupar las cuatro funciones de transformación del algoritmo: *SubBytes*, *ShiftRows*, *MixColumns*, *AddRoundKey*, de tal manera que cada transformación se ejecute de manera sucesiva. El bloque Ronda debe ejecutarse $N_r - 1$ veces y su implementación en hardware se realiza como se aprecia en la figura 3.30.

El bloque Ronda no necesita señal de reloj, por lo que se obtiene una mejora en velocidad al no necesitar que se ejecute en un ciclo de reloj para poder continuar con el procesamiento.

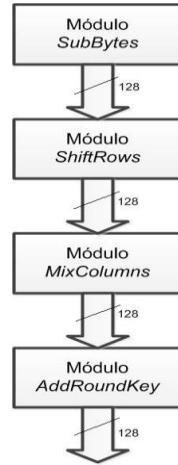


Figura 3.30. Bloque Ronda.

3.11.1. Bloque Ronda Final

El bloque Ronda Final es ligeramente diferente al bloque Ronda ya que no incluye el módulo *MixColumns* y su implementación en hardware se realiza conectando los módulos correspondientes como se aprecia en la figura 3.31.

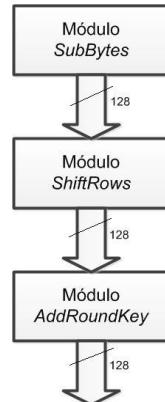


Figura 3.31. Bloque Ronda Final.

3.12. Bloque Ronda Inverso

El bloque Ronda Inverso agrupa las cuatro funciones de transformación inversa del algoritmo: *InvSubBytes*, *InvShiftRows*, *InvMixColumns*, *InvAddRoundKey*, de tal manera que se ejecute cada transformación una tras otra. El bloque Ronda Inverso debe ejecutarse $N_r - 1$ veces y su implementación en hardware se realiza al conectar los módulos desarrollados para cada transformación como se aprecia en la figura 3.32. El bloque Ronda Inverso tampoco utiliza señal de reloj.

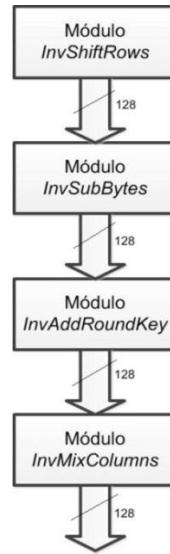


Figura 3.32. Bloque Ronda Inverso.

3.12.1. Bloque Ronda Final inverso

El bloque Ronda Final Inverso difiere del bloque Ronda Inverso en que no incluye el módulo *InvMixColumns*. Su implementación se realiza de acuerdo con la figura 3.33.

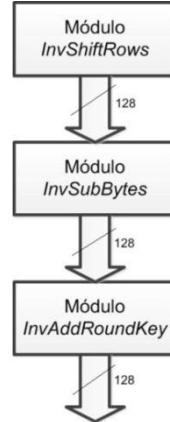


Figura 3.33. Bloque Ronda Final Inverso.

3.13. Proceso de encriptación en modo de operación ECB (*Electronic Codebook*)

De manera general, en las implementaciones hardware del algoritmo de *Rijndael* se distinguen dos partes fundamentales: la unidad de encriptación, que incluye los circuitos que implementan las funciones de transformación y la unidad de generación

de Subclaves. Ambas unidades interactúan mediante el módulo *AddRoundKey* en cada ronda de ejecución del algoritmo.

Para implementar el proceso de encriptación en modo *ECB* se ha utilizado una arquitectura *pipeline*, que requiere de nueve bloques de ronda, un bloque ronda final, diez registros de 128 bits, la unidad de subclaves y líneas de interconexión tal como se observa en la figura 3.34:

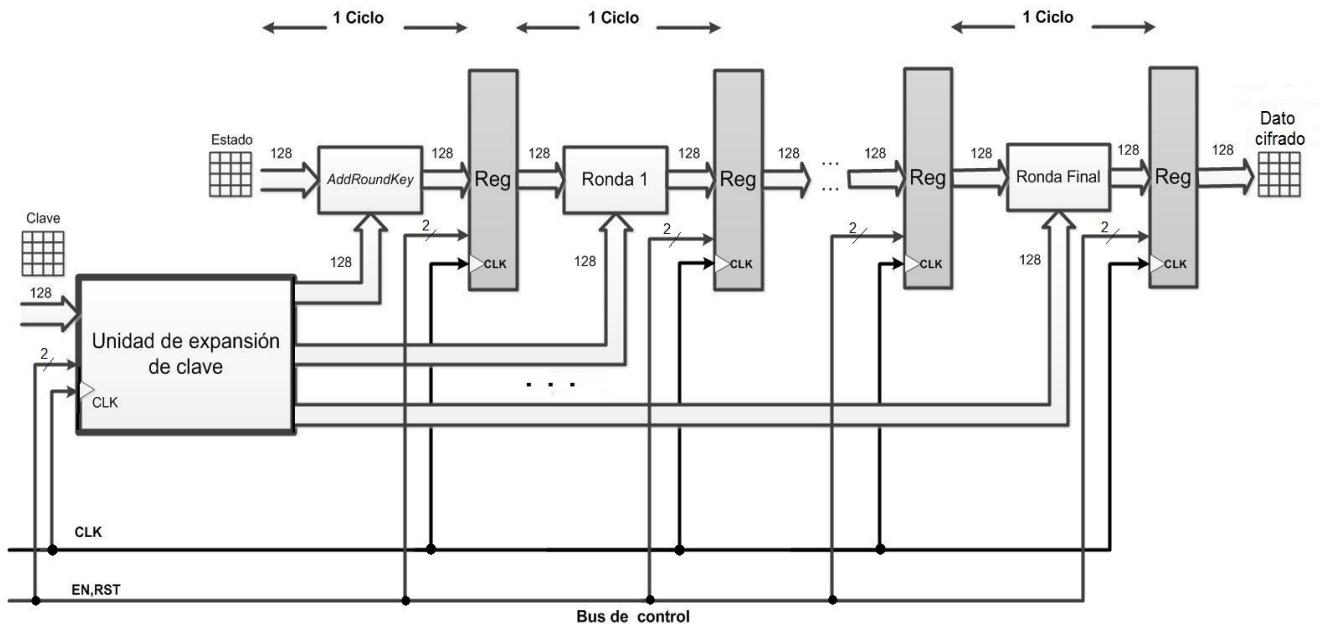


Figura 3.34. Arquitectura *pipeline* para el proceso de encriptación.

La organización mostrada en la figura 3.34, permite procesar un dato por cada ciclo de reloj una vez que el primer dato ha sido encriptado. [9], [11]

La unidad de subclaves y la unidad de encriptación están conectadas a un bus de control que incluye las señales de *Reset* y *Enable*.

En este circuito, los registros se encargan de almacenar los datos que van siendo ejecutados en cada ronda y ciclo de reloj.

Los valores utilizados como entrada de datos (*matriz de estado*) y clave de encriptación, cada uno de 16 bytes ($N_b = 4$ y $N_k = 4$), en formato hexadecimal son:

Texto Plano 1 = 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
Clave de encriptación 1 = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Texto Plano 2 = 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Clave de encriptación 2 = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

La figura 3.35 Muestra los resultados de simulación. Se aprecia que el circuito tarda exactamente diez ciclos para obtener el dato encriptado.

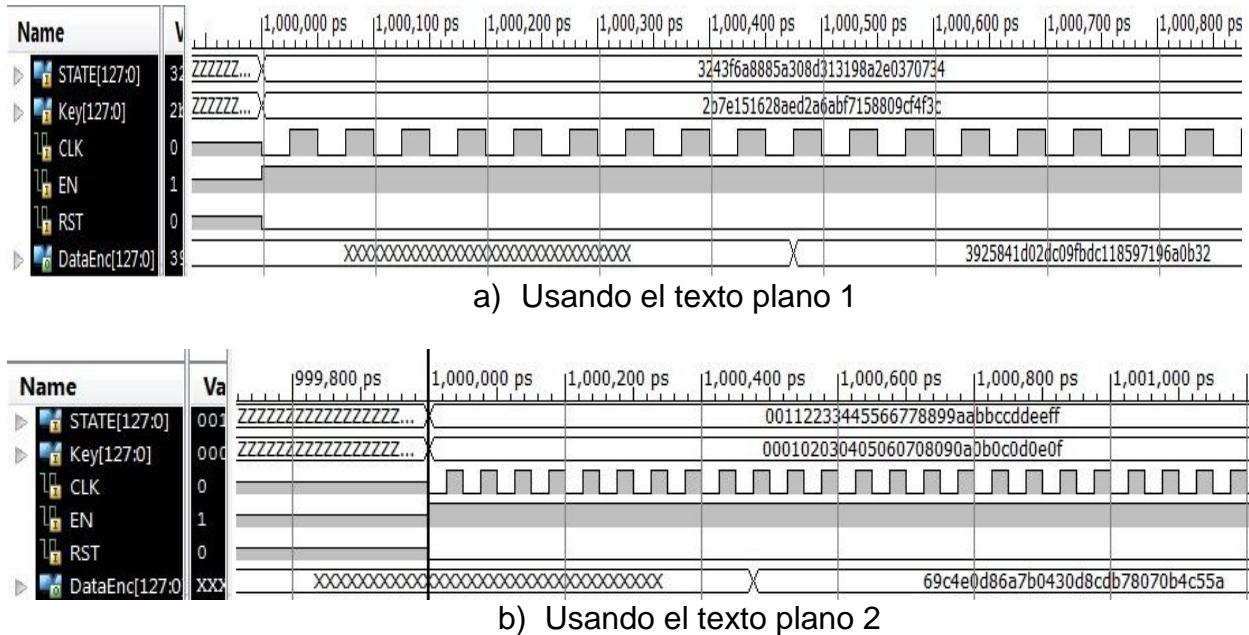


Figura 3.35. Resultados de simulación para el proceso de encriptación en modo *ECB*.

3.13.1. Sistema de encriptación en modo *ECB* para múltiples bloques

Para encriptar múltiples bloques de texto plano se puede utilizar un esquema como el que se muestra en la figura 3.36, donde los bloques de texto plano se almacenan en una memoria *ROM*. En este caso se almacenan ocho bloques de 128 bits. Las claves también se almacenan en una memoria *ROM*. La unidad de control se basa en una máquina de estados (*FSM*) y se encarga de controlar el flujo de datos entre las memorias *ROM*, el módulo de encriptación *AES_ECB_ENC* y la memoria *RAM*. La memoria *RAM* se encarga de almacenar los ocho datos encriptados que salen del módulo de encriptación *ECB* de tal manera que se puedan leer posteriormente.

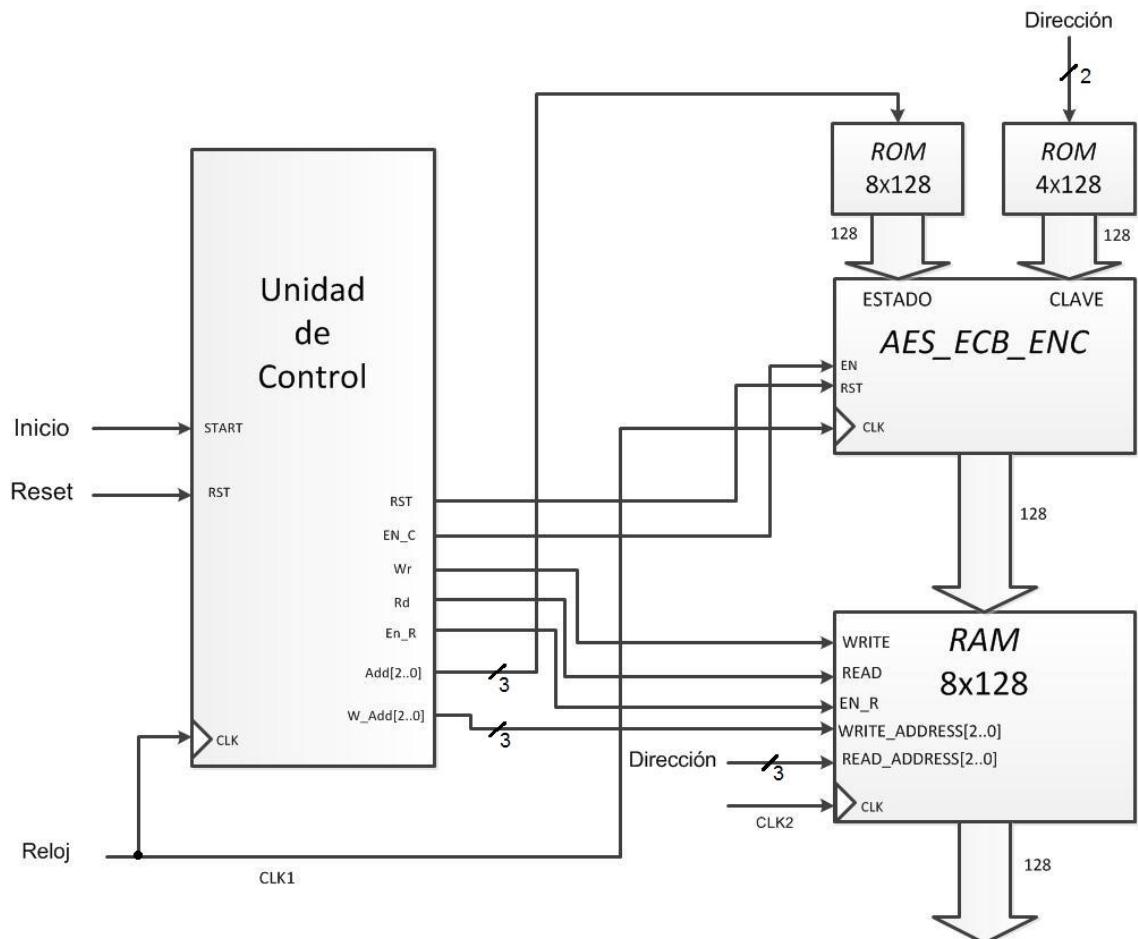


Figura 3.36. Esquema de cifrado para múltiples bloques en modo ECB.

Una vez la entrada de $START=1'$, la unidad de control inicia la lectura de la memoria *ROM* (puede ser reemplazada por una memoria *RAM*, aquí las *ROM* se usan para efectos de prototipaje), es decir que por cada ciclo de reloj un bloque de texto plano entra al módulo *AES_ECB_ENC*, por lo tanto el primer bloque encriptado se obtiene al décimo ciclo de reloj y se escribe en la memoria *RAM*, posteriormente por cada ciclo de reloj hay un dato encriptado que se escribe en la respectiva posición de memoria *RAM*.

Una vez se han escrito todos los datos encriptados en la memoria *RAM*, la unidad de control inhabilita todos los módulos de manera que no se procesen datos innecesarios y además hace posible leer los datos en la memoria *RAM* mediante el puerto *READ_ADDRESS*. Si la entrada de *RESET=1* la unidad de control vuelve al estado inicial y se vuelven a procesar todos los datos desde el principio sobrescribiendo la memoria *RAM*, esto hace posible que se pueda elegir de nuevo otra clave y dado el caso otros datos a encriptar para que sean ejecutados por el sistema de encriptación.

Para probar el funcionamiento del circuito de la figura 3.36, se escribe en las memorias *ROM* de texto plano y de clave la siguiente información para que sea encriptada en modo *ECB*:

Texto plano:

“Estos son bloques de texto plano utilizados para probar la implementación en hardware del algoritmo de Rijndael para 128 bits”

Clave:

“Ian_Carlo_Guzman”

Este texto plano ocupa un tamaño en memoria de 125 bytes, por lo tanto deben agregarse tres bytes de *padding* (agregación de ceros de manera que el tamaño del bloque se complete) al final de todo el bloque de tal manera que el tamaño total sea de 128 bytes (1024 bits):

Texto plano:

**45 73 74 6f 73 20 73 6f 6e 20 62 6c 6f 71 75 65
73 20 64 65 20 74 65 78 74 6f 20 70 6c 61 6e 6f
20 75 74 69 6c 69 7a 61 64 6f 73 20 70 61 72 61
20 70 72 6f 62 61 72 20 6c 61 20 69 6d 70 6c 65
6d 65 6e 74 61 63 69 6f 6e 20 65 6e 20 68 61 72
64 77 61 72 65 20 64 65 6c 20 61 6c 67 6f 72 69
74 6d 6f 20 64 65 20 52 69 6e 6a 64 61 65 6c 20
70 61 72 61 20 31 32 38 20 62 69 74 73 00 00 00**

Clave:

49 61 6e 5f 43 61 72 6c 6f 5f 47 75 7a 6d 61 6e

Los resultados de simulación se muestran en las figuras 3.37, 3.38 y 3.39, donde la figura 3.37 muestra el proceso de lectura, encriptación y escritura que se realiza entre la unidad de control, las memorias y el módulo de encriptación, mientras que las figuras 3.38 y 3.39 muestran los datos finalmente escritos en cada posición de la memoria RAM (desde la posición 0 hasta la posición 7) en formato hexadecimal y ASCII respectivamente.

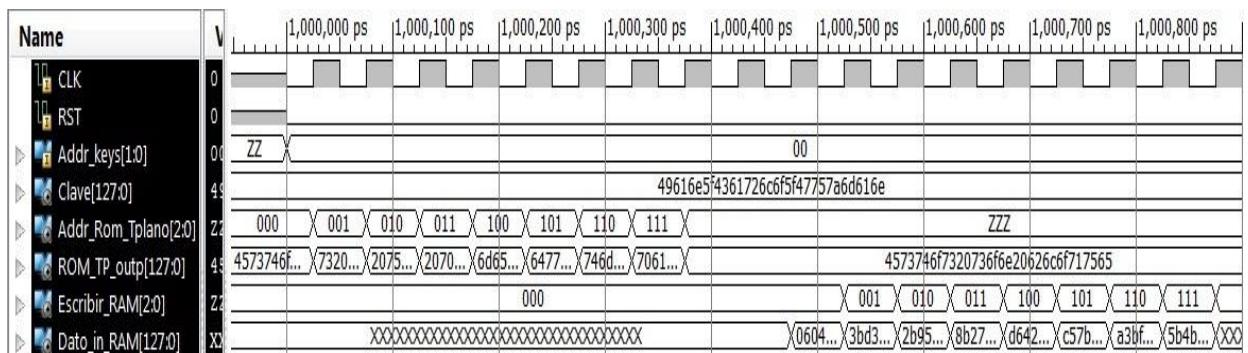


Figura 3.37. Proceso de encriptación, lectura y escritura entre la unidad de control, las memorias y el módulo de encriptación.

En la figura 3.37 se puede notar que encriptar los ocho datos guardados en la memoria *ROM* y escribirlos en la memoria *RAM* toma 18 ciclos de reloj, donde el primer dato se encripta y se escribe en *RAM* en el décimo ciclo de reloj y los restantes siete datos se encriptan y se almacenan en *RAM* a razón de un ciclo de reloj. Finalmente en el último ciclo de reloj se habilita la memoria *RAM* para que se puedan leer los datos escritos y se inhabilita el circuito de encriptación para que no se procesen más datos.

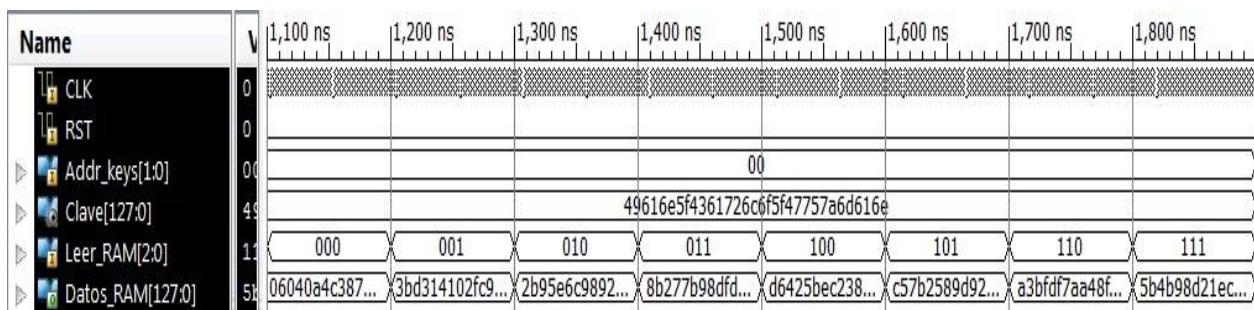


Figura 3.38. Datos encriptados en formato hexadecimal escritos en memoria *RAM*.

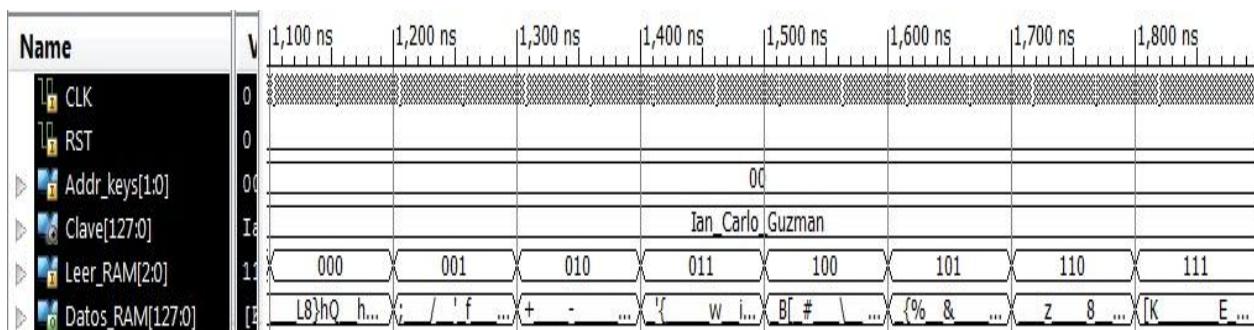


Figura 3.39. Datos encriptados en formato ASCII/ escritos en memoria *RAM*.

De acuerdo a los resultados de simulación, los datos encriptados vistos en formato ASCII/ son un conjunto de caracteres incoherentes e incomprendibles para cualquier lector. Los datos se han organizado en la tabla 3.2, en la cual se puede apreciar cada dato encriptado para cada posición de memoria en formato hexadecimal:

Tabla 3.2. Tabla que contiene los datos encriptados en modo *ECB*, escritos en memoria *RAM*.

Posición en memoria <i>RAM</i> (HEX)	Dato encriptado (HEX)
0	06 04 0a 4c 38 7d 68 51 c6 a7 68 a0 a6 61 a9 f0
1	3b d3 14 10 2f c9 f0 27 ea 66 1b 0d b5 98 79 8c
2	2b 95 e6 c9 89 2d 91 94 84 18 b9 a8 43 59 9d fd
3	8b 27 7b 98 df d9 a9 99 77 bb 96 69 da 7b 03 7e
4	d6 42 5b ec 23 8a be d9 5c 02 d4 20 03 3e 69 7d
5	c5 7b 25 89 d9 26 d1 d9 ee 0d 09 61 5f 8f ea e7
6	a3 bf df 7a a4 8f 8a e4 38 a0 09 d4 bd c0 b3 ae
7	5b 4b 98 d2 1e cc 97 b5 9c 45 0b 79 08 95 ec 05

3.14. Proceso de desencriptación en modo de operación ECB (Electronic Codebook)

De manera similar al proceso de encriptación en modo *ECB*, el proceso de desencriptación también se implementa con una arquitectura *pipeline* la cual requiere nueve bloques ronda inversos, un bloque ronda final inverso, diez registros de 128 bits, una unidad de subclaves y líneas de interconexión como se aprecia en la figura 3.40.

A diferencia de la organización *pipeline* para el proceso de encriptación, se puede notar en la figura 3.40, que la última subclave generada en la unidad de expansión de subclaves se usa como entrada a la primera transformación *AddRoundKey* y la primera subclave generada se usa como entrada a la ronda inversa final.

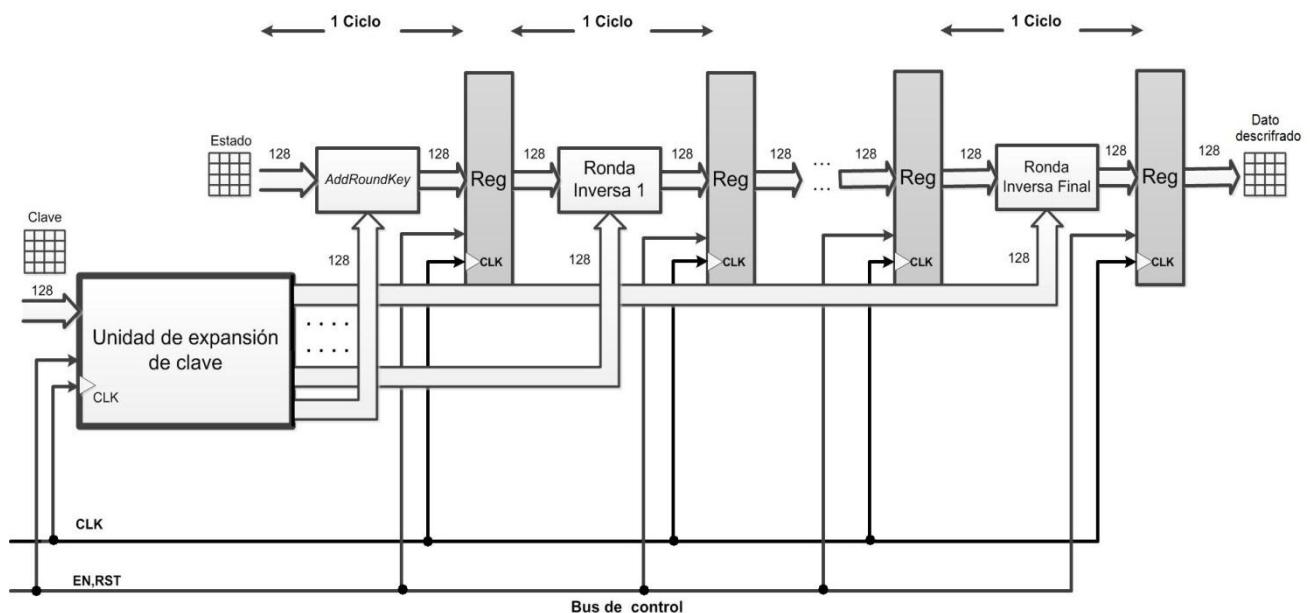


Figura 3.40. Organización *pipeline* para el proceso de desencriptación.

Esta estructura funcional permite procesar un dato por cada ciclo de reloj una vez que el primer dato ha sido desencriptado.

La unidad de Subclaves y la unidad de desencriptación se controlan mediante las señales *Reset* y *Enable*. En este circuito los registros se encargan de almacenar los datos que van siendo ejecutados en cada una de las rondas. Y también controlan el paso de datos acompañados de las señales *Reset* y *Enable*.

Los siguientes son los valores utilizados como entrada de datos (*matriz de estado*) y clave de desencriptación, cada uno de 16 bytes ($N_b = 4$ y $N_k = 4$) , en formato hexadecimal:

Texto cifrado 1 = 39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32
Clave desencriptación 1 = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Texto cifrado 2 = 69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
Clave desencriptación 2 = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

La figura 3.41 Muestra los resultados de simulación y se puede apreciar que el circuito tarda exactamente diez ciclos de reloj para obtener el dato desencriptado.



Figura 3.41. Resultados de simulación para el proceso de desencriptación en modo *ECB*.

3.14.1. Sistema de desencriptación en modo *ECB* para múltiples bloques

Para desencriptar múltiples bloques de texto plano se utilizó el esquema mostrado en la figura 3.42, donde los bloques de texto cifrado se almacenan en una memoria *ROM*, en este caso se almacenan ocho bloques de 128 bits, las claves también se almacenan en una memoria *ROM*. La unidad de control se basa en una máquina de estados y se encarga de controlar el flujo de datos entre las memorias *ROM*, el módulo de encriptación *AES_ECB_DEC* y la memoria *RAM*. La memoria *RAM* almacena los datos desencriptados que salen del módulo de desencriptación *ECB* de tal manera que se puedan leer posteriormente.

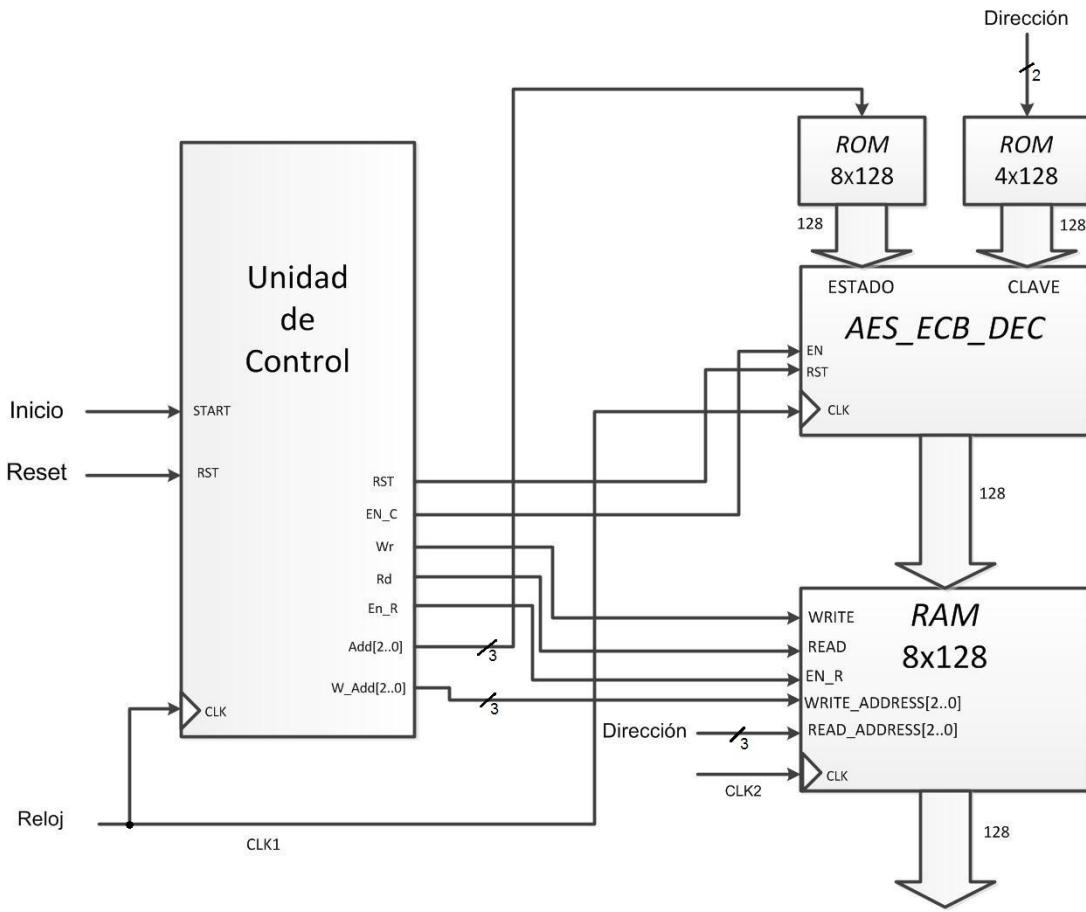


Figura 3.42. Esquema de descifrado para múltiples bloques en modo *ECB*.

Una vez la entrada de *START*=’1’, la unidad de control empieza a leer la memoria *ROM* para que por cada ciclo de reloj un bloque de texto cifrado entre al módulo *AES_ECB_DEC*, por lo tanto el primer bloque desencriptado se obtiene en el décimo ciclo de reloj y es escrito en la memoria *RAM*. Posteriormente por cada ciclo de reloj hay un dato desencriptado que se almacena en la respectiva posición de memoria *RAM*.

Una vez se han escrito todos los datos desencriptados en la memoria *RAM*, la unidad de control inhabilita todos los módulos de manera que no se procesen datos innecesarios y además hace posible leer los datos en la memoria *RAM* mediante el puerto *READ_ADDRESS*. Si la entrada de *RESET*=1 la unidad de control vuelve al estado inicial y se calculan todos los datos desde el principio para sobrescribir la memoria *RAM*, esto posibilita la elección de una nueva clave y dado el caso, otros datos a desencriptar para que sean procesados por el sistema.

Para probar el funcionamiento del circuito de la figura 3.42, se escribe en las memorias *ROM* que almacenan el texto cifrado y las claves la siguiente información de tal manera que sea desencriptada en modo *ECB*:

Texto Cifrado:

```
06 04 0a 4c 38 7d 68 51 c6 a7 68 a0 a6 61 a9 f0
3b d3 14 10 2f c9 f0 27 ea 66 1b 0d b5 98 79 8c
2b 95 e6 c9 89 2d 91 94 84 18 b9 a8 43 59 9d fd
8b 27 7b 98 df d9 a9 99 77 bb 96 69 da 7b 03 7e
d6 42 5b ec 23 8a be d9 5c 02 d4 20 03 3e 69 7d
c5 7b 25 89 d9 26 d1 d9 ee 0d 09 61 5f 8f ea e7
a3 bf df 7a a4 8f 8a e4 38 a0 09 d4 bd c0 b3 ae
5b 4b 98 d2 1e cc 97 b5 9c 45 0b 79 08 95 ec 05
```

Clave:

```
49 61 6e 5f 43 61 72 6c 6f 5f 47 75 7a 6d 61 6e
```

Los resultados de simulación se muestran en las figuras 3.43, 3.44 y 3.45, donde las figuras 3.43 y 3.44 muestran el proceso de lectura, desencriptación y escritura que se realiza entre la unidad de control, las memorias y el módulo de desencriptación, en formato ASCII y hexadecimal respectivamente, por otra parte la figura 3.45 muestra los datos desencriptados ya escritos en cada posición de memoria (desde la posición 0 hasta la posición 7).

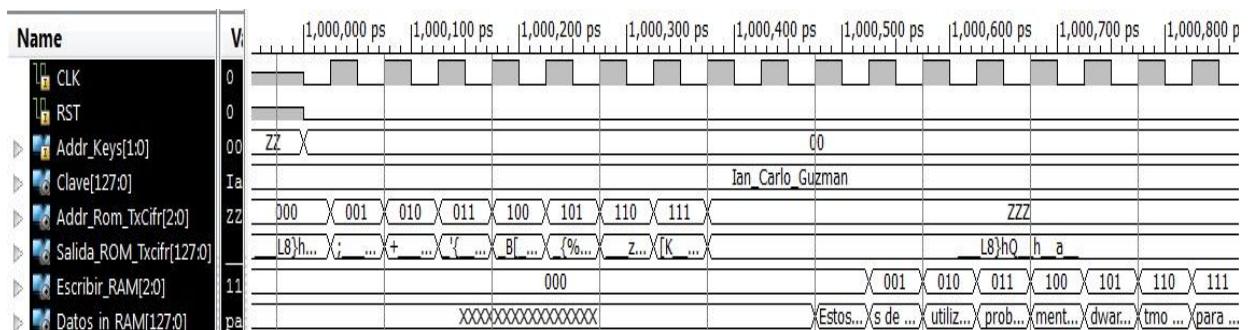


Figura 3.43. Proceso de desencriptación, lectura y escritura entre la unidad de control, las memorias y el módulo de desencriptación en formato ASCII.

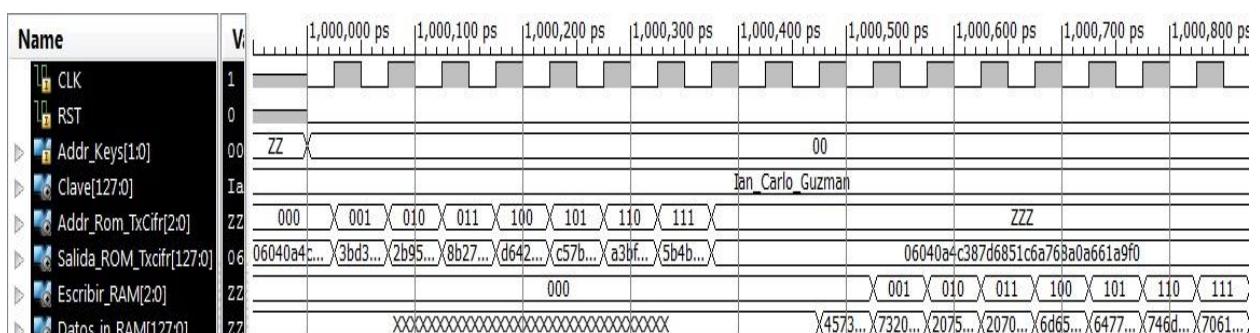


Figura 3.44. Proceso de desencriptación, lectura y escritura entre la unidad de control, las memorias y el módulo de desencriptación en formato hexadecimal.

De las figuras 3.43 y 3.44 se puede notar que desencriptar los ocho datos almacenados en la memoria *ROM* y escribirlos en la memoria *RAM* toma 18 ciclos de reloj, donde el primer dato se desencripta y se escribe en memoria *RAM* en el décimo ciclo de reloj y los restantes siete datos se desencriptan y se escriben en la memoria *RAM* a razón de un ciclo de reloj. Finalmente en el último ciclo de reloj se habilita la memoria *RAM* para que se puedan leer los datos escritos y se inhabilita el circuito de desencriptación para que no se procesen más datos.

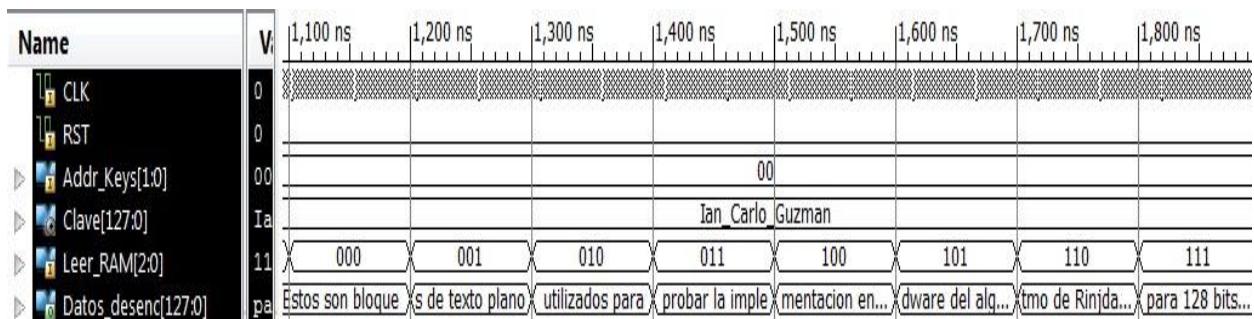


Figura 3.45. Lectura de Datos desencriptados escritos en memoria *RAM*.

De acuerdo a los resultados de simulación los datos desencriptados se han organizado en la tabla 3.3 en la cual se puede apreciar cada dato desencriptado para cada posición de memoria en formato *ASCII*:

Tabla 3.3. Tabla que contiene los datos desencriptados en modo *ECB*, escritos en memoria *RAM*.

Posición en memoria <i>RAM</i> (HEX)	Dato desencriptado (<i>ASCII</i>)
0	Estos son bloques
1	de texto plano
2	utilizados para
3	probar la imple
4	mentación en har
5	dware del algori
6	tmo de Rinjdael
7	para 128 bits

3.15. Proceso de encriptación y desencriptación en modo de operación *CTR* (*Counter*)

El modo *ECB* se usa para encriptar los datos de forma directa dada una clave secreta, esto significa que si un mismo dato se encripta más de una vez los datos encriptados equivalentes serán los mismos siempre. Por el contrario, el modo contador encripta datos de manera que si un mismo dato se encripta más de una vez, los datos encriptados equivalentes siempre serán diferentes, por lo tanto el modo contador presenta un nivel de seguridad más alto. No obstante, se debe cumplir que cada bloque de contador sea diferente para cada bloque de texto plano.

3.15.1. Módulo de bloque de contador

Este módulo proporciona los bloques de contador que ingresan al bloque de encriptación *ECB*. Cada bloque tiene una longitud de 128 bits distribuidos en dos bloques de 64 bits cada uno. Los 64 bits más significativos se conocen como bloque de *string* único y los 64 bits menos significativos se utilizan como contador el cual se incrementa en cada ciclo de reloj. El *String* único es un valor impredecible que se mantiene fijo y el contador es un valor que se incrementa en uno por cada ciclo de reloj.

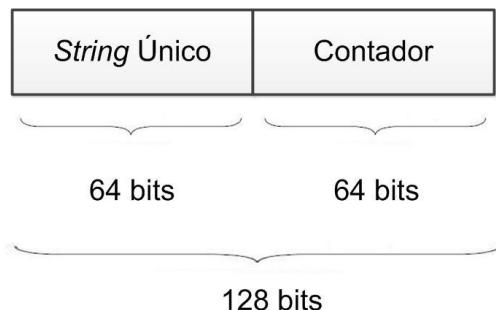


Figura 3.46. Distribución del bloque de contador.

De la figura 3.46 se puede notar que son posibles 2^{64} bloques diferentes en modo contador, por lo tanto si se encriptan n bloques de 128 bits se debe cumplir la condición que establece que establece $n \leq 2^{64}$, de lo contrario se utilizarían bloques repetidos, lo que puede afectar la protección criptográfica.

Suponiendo una frecuencia de reloj de $100MHz$ para un contador de 64 bits se tendría un periodo de $10ns$ lo cual significa que al contador le tardaría $(10ns)(2^{64})=5849.42$ años para generar todos los valores posibles. Por lo tanto, el aumento o la disminución del tiempo que se tarda en encriptar n bloques en modo contador depende de la frecuencia de reloj y también de la presencia de un bloque de encriptación en paralelo. Así que se puede deducir que cumplir la condición $n \leq 2^{64}$ no es un problema relevante para una frecuencia de reloj de $100MHz$ con uno ó dos bloques de encriptación en modo contador en paralelo.

El módulo de bloque de contador se implementa en hardware utilizando registros y un contador ascendente de 64 bits, de tal manera que en un registro se almacene el *string* Único y en otro registro se almacene la cuenta del contador por cada ciclo de reloj. La figura 3.47 ilustra el diagrama de bloques para implementar el módulo que genera los bloques contador:

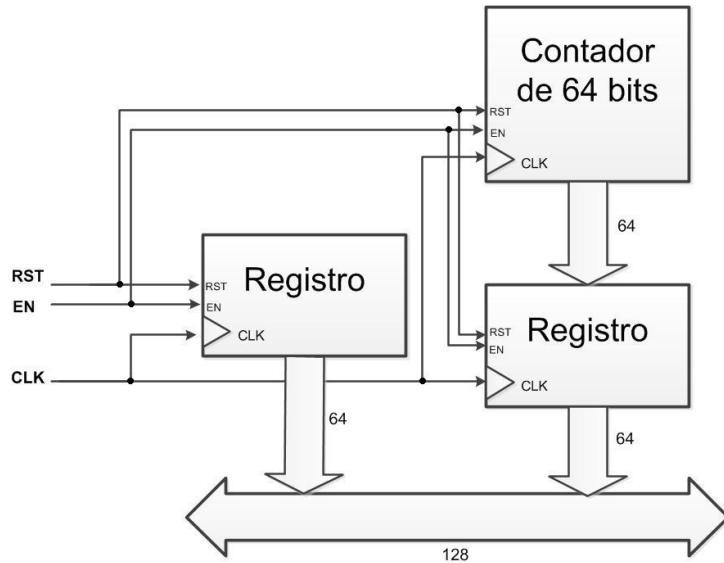


Figura 3.47. Diagrama de circuito del módulo de bloque contador.

La figura 3.48 muestra los resultados de simulación para el módulo de bloque de contador cuando el bloque de contador utiliza los siguientes valores en formato hexadecimal:

String Único: **f0 f1 f2 f3 f4 f5 f6 f7**
 Valor inicial del contador: **f8 f9 fa fb fc fd fe ff**

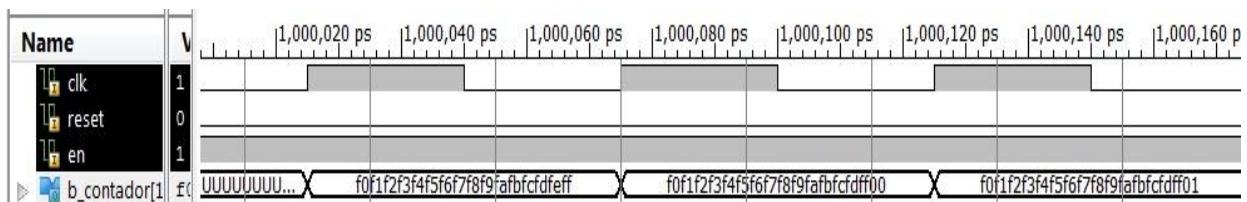


Figura 3.48. Resultados de simulación del módulo contador.

Como se puede apreciar, por cada ciclo de reloj se genera un nuevo bloque de contador. El *string* único se mantiene fijo y el contador incrementa en uno la cuenta de los 64 bits menos significativos.

3.15.2. Proceso de encriptación y desencriptación

El proceso de encriptación y desencriptación en modo *CTR* se puede implementar en hardware utilizando un módulo contador, un bloque de encriptación *ECB* y un bloque de compuertas XOR para 128 bits de dos entradas como se puede apreciar en la figura 3.49. Por cada ciclo de reloj se genera un nuevo valor de contador que luego ingresa al bloque de encriptación *ECB*. Una vez el bloque contador ha sido encriptado se realiza una operación XOR con el texto plano ó texto cifrado y se obtiene el texto cifrado ó descifrado.

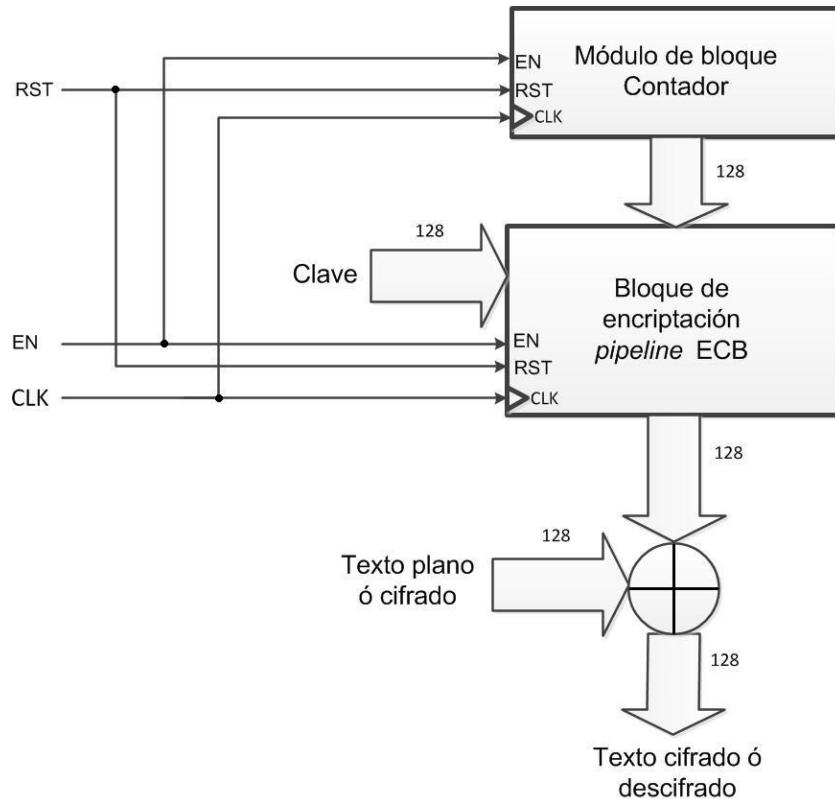


Figura 3.49. Diagrama de bloques de circuito para el proceso de encriptación y desencriptación en modo *CTR*.

La figura 3.50 muestra resultados de simulación para el proceso de encriptación en modo contador cuando se utilizan los vectores de prueba sugeridos en el estándar [2]:

Texto plano: **6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a**
 Clave: **2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c**

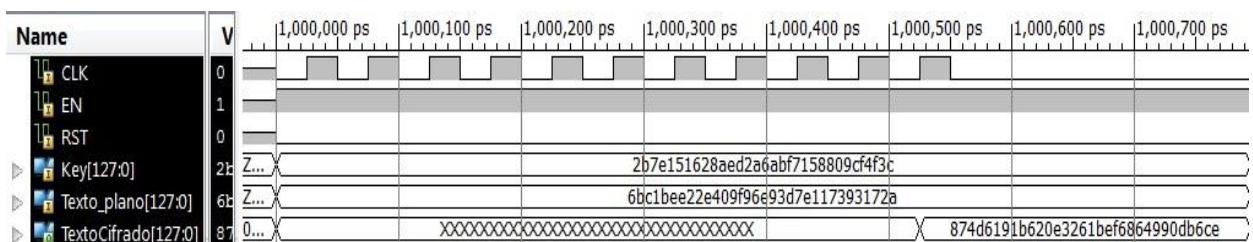


Figura 3.50. Resultados de simulación para el proceso de encriptación en modo contador.

De la figura 3.50 se puede apreciar que el dato se encripta en el ciclo de reloj número once, debido a que el primer ciclo de reloj se utiliza para que el módulo de bloque contador ponga el primer bloque contador a la entrada del módulo de encriptación *ECB*, los restantes diez ciclos de reloj constituyen el tiempo que tarda el bloque de encriptación en procesar el dato.

3.15.3 Sistema de encriptación en modo CTR para múltiples bloques

Para encriptar ó desencriptar múltiples bloques de texto plano se puede utilizar un esquema como el que se muestra en la figura 3.51, donde los bloques de texto plano se almacenan en una memoria *ROM*, en este caso se almacenan ocho bloques de 128 bits. Las claves también se almacenan en una memoria *ROM*. La unidad de control está basada en una máquina de estados *FSM* y se encarga de controlar el flujo de datos entre las memorias *ROM*, el contador, el módulo de encriptación *AES_ECB_ENC pipeline* y la memoria *RAM*. La memoria *RAM* se encarga de almacenar los ocho datos encriptados de tal manera que se puedan leer posteriormente.

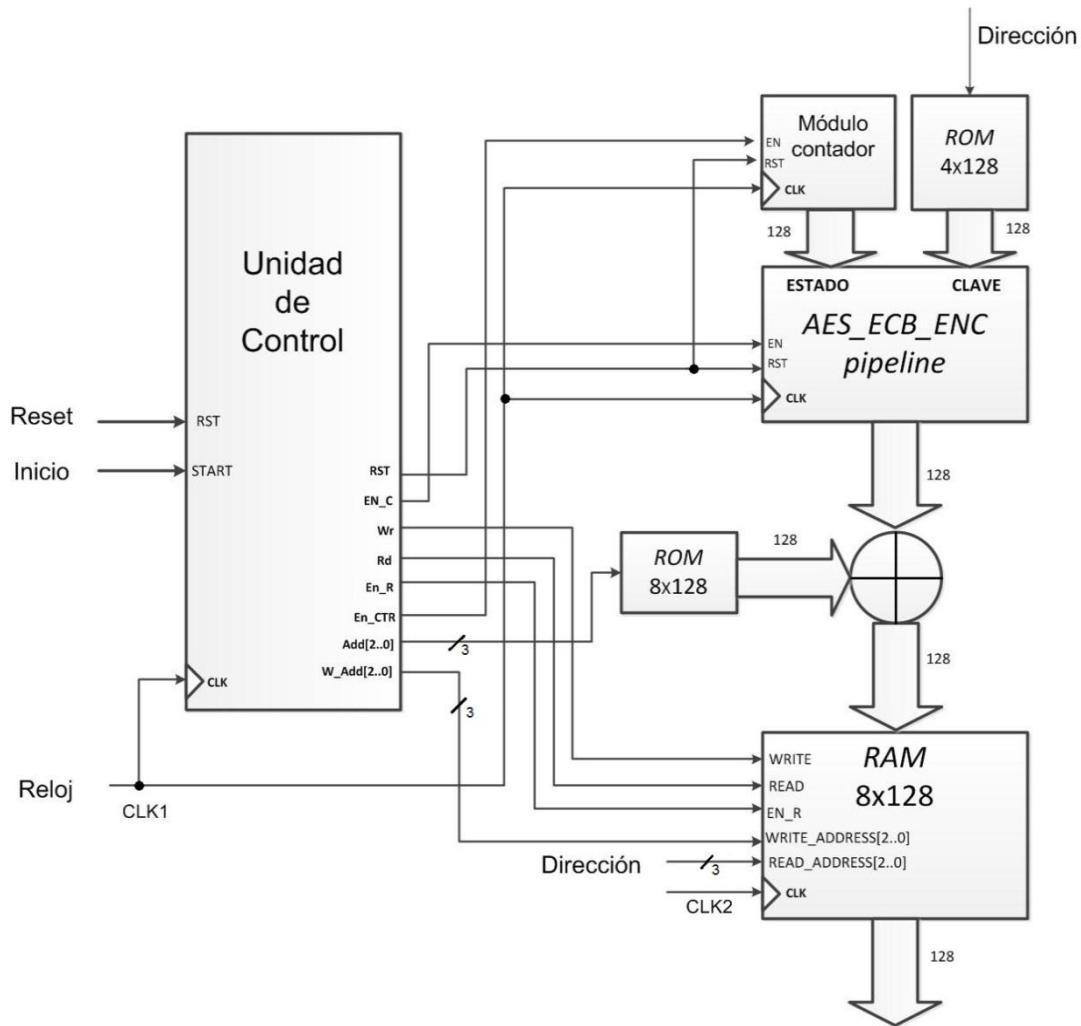


Figura 3.51. Esquema de cifrado y descifrado para múltiples bloques en modo *CTR*.

Una vez se habilita el inicio ($START=1'$), la unidad de control habilita tanto al contador como a la unidad de encriptación *AES_ECB_ENC* dando así inicio al proceso de encriptación del bloque contador. En el primer ciclo de reloj el bloque contador inicial queda disponible en el bus de entrada al módulo de encriptación *AES_ECB_ENC* y se obtiene el primer contador encriptado luego del onceavo ciclo de reloj para realizar la operación XOR con el primer bloque de texto plano y luego

ser almacenado en la memoria *RAM*. Por cada ciclo de reloj hay un nuevo dato encriptado que se almacena en la respectiva posición de memoria *RAM*. Una vez se han escrito todos los datos encriptados en la memoria *RAM*, la unidad de control inhabilita los módulos para que no se procesen datos innecesarios y para posibilitar la lectura de datos en la memoria *RAM* mediante el puerto *READ_ADDRESS*. Si la entrada de *RESET=1*, la unidad de control inicia de nuevo el proceso de encriptación, haciendo posible el procesamiento de nuevos datos.

Para probar el funcionamiento del circuito de la figura 3.51, se escribe en las respectivas memorias *ROM* tanto el texto plano como la clave. Los resultados de simulación se muestran en la figura 3.52, en la que se puede apreciar el proceso de lectura, encriptación y escritura que se realiza entre la unidad de control, el contador, las memorias y el módulo de encriptación *AES_ECB_ENC*. La figura 3.53 muestra los datos encriptados ya escritos en cada posición de memoria (desde la posición 0 hasta la posición 7).

Texto plano:

```
6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
ae 2d 8a 57 1e 03 ac 9c 9e b7 6f ac 45 af 8e 51
30 c8 1c 46 a3 5c e4 11 e5 fb c1 19 1a 0a 52 ef
f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10
6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
ae 2d 8a 57 1e 03 ac 9c 9e b7 6f ac 45 af 8e 51
30 c8 1c 46 a3 5c e4 11 e5 fb c1 19 1a 0a 52 ef
f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10
```

Clave:

```
2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
```

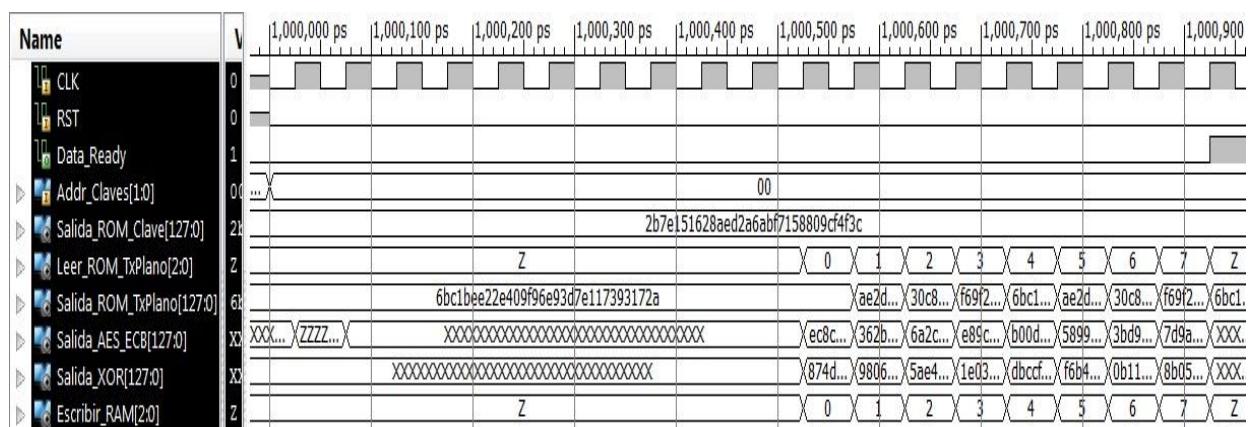


Figura 3.52. Proceso de encriptación, lectura y escritura entre la unidad de control, las memorias y el módulo de encriptación.

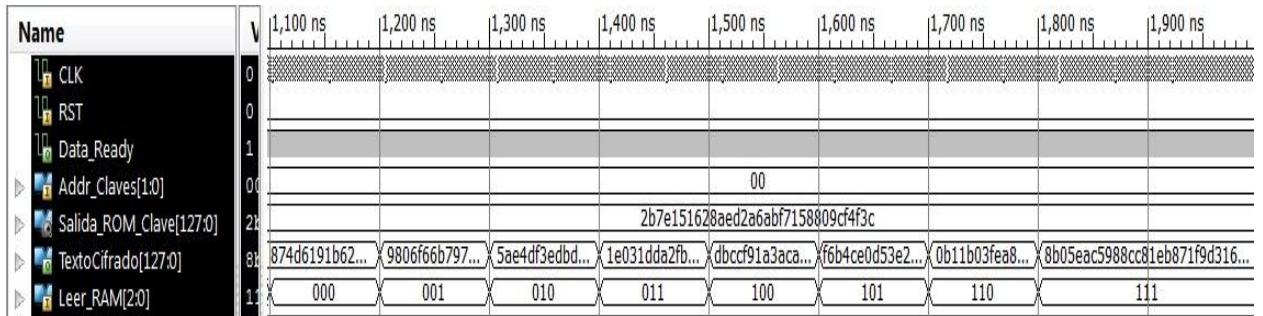


Figura 3.53. Datos encriptados en formato hexadecimal escritos en memoria *RAM*.

Los datos encriptados escritos en cada posición de la memoria *RAM* se han organizado en la tabla 3.4. Los resultados son idénticos a los obtenidos en el estándar [2]. También se puede notar que los datos encriptados en las posiciones 0 y 4, 1 y 5, 2 y 6, 3 y 7 son diferentes a pesar de que sus textos planos equivalentes son los mismos, siendo esta la razón por la cual el modo *CTR* se diferencia del modo *ECB*.

Tabla 3.4. Resultados de encriptación en modo contador escritos en memoria *RAM*.

RAM (HEX)	Texto Cifrado (HEX) (Escrito en memoria <i>RAM</i>)	Texto plano equivalente (HEX)
0	87 4d 61 91 b6 20 e3 26 1b ef 68 64 99 0d b6 ce	6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
1	98 06 f6 6b 79 70 fd ff 86 17 18 7b b9 ff fd ff	ae 2d 8a 57 1e 03 ac 9c 9e b7 6f ac 45 af 8e 51
2	5a e4 df 3e db d5 d3 5e 5b 4f 09 02 0d b0 3e ab	30 c8 1c 46 a3 5c e4 11 e5 fb c1 19 1a 0a 52 ef
3	1e 03 1d da 2f be 03 d1 79 21 70 a0 f3 00 9c ee	f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10
4	db cc f9 1a 3a ca 0e 98 19 55 4e 86 e3 d8 b2 28	6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
5	f6 b4 ce 0d 53 e2 ad 69 8d 7d be 34 38 26 67 4a	ae 2d 8a 57 1e 03 ac 9c 9e b7 6f ac 45 af 8e 51
6	0b 11 b0 3f ea 82 cf e8 80 92 6d 21 59 f2 20 ad	30 c8 1c 46 a3 5c e4 11 e5 fb c1 19 1a 0a 52 ef
7	8b 05 ea c5 98 8c c8 1e b8 71 f9 d3 16 e9 a0 a1	f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10

3.15.4. Sistema de desencriptación en modo CTR para múltiples bloques

El proceso de desencriptación en modo contador es igual al proceso de encriptación. Así que, en vez de ingresar el texto plano al circuito se ingresa el texto cifrado, por lo tanto no es necesario usar un bloque de desencriptación en modo *ECB*. Sin embargo, se debe tener en cuenta que los bloques contador utilizados para desencriptar algún texto cifrado deben ser los mismos que se utilizaron para cifrar el texto plano.

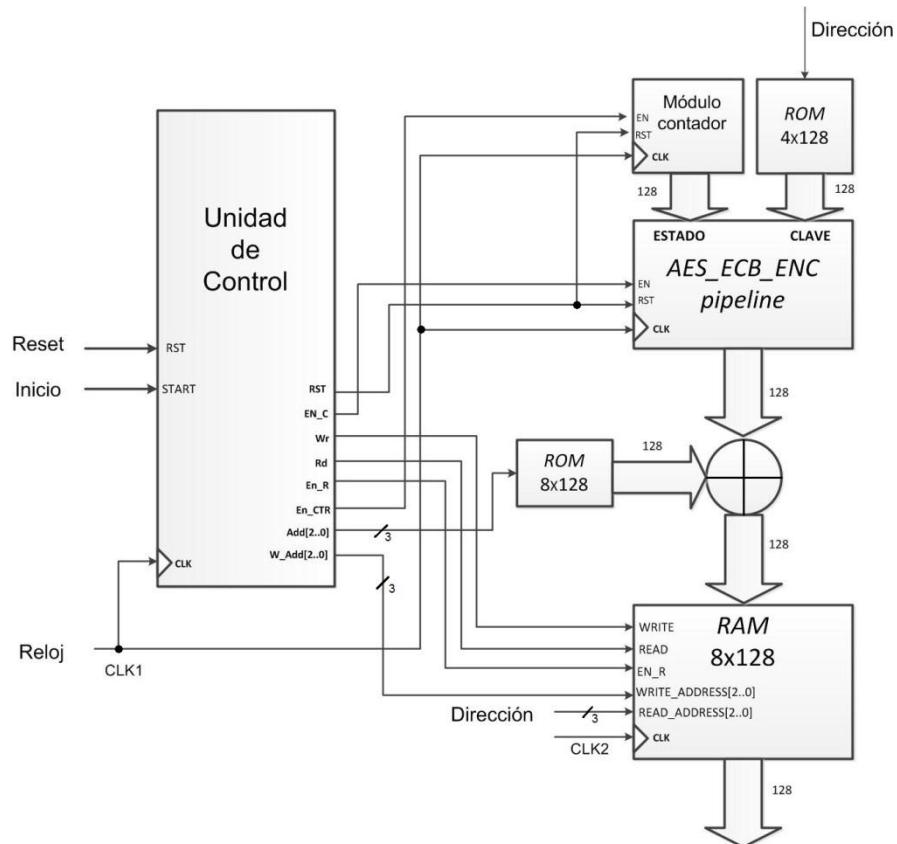


Figura 3.54. Esquema de cifrado/descrifrado para múltiples bloques en modo CTR.

El siguiente texto (texto cifrado) y clave, se utilizan para probar el funcionamiento del sistema en modo *CTR*.

Texto Cifrado:

```
87 4d 61 91 b6 20 e3 26 1b ef 68 64 99 0d b6 ce
98 06 f6 6b 79 70 fd ff 86 17 18 7b b9 ff fd ff
5a e4 df 3e db d5 d3 5e 5b 4f 09 02 0d b0 3e ab
1e 03 1d da 2f be 03 d1 79 21 70 a0 f3 00 9c ee
db cc f9 1a 3a ca 0e 98 19 55 4e 86 e3 d8 b2 28
f6 b4 ce 0d 53 e2 ad 69 8d 7d be 34 38 26 67 4a
0b 11 b0 3f ea 82 cf e8 80 92 6d 21 59 f2 20 ad
8b 05 ea c5 98 8c c8 1e b8 71 f9 d3 16 e9 a0 a1
```

Clave:

2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Los resultados de simulación se muestran en las figuras 3.55 y 3.56. La figura 3.55 muestra el proceso de lectura, desencriptación y escritura que se realiza entre la unidad de control, el contador, las memorias y el módulo de encriptación AES_ECB_ENC. La figura 3.56 muestra los datos desencriptados ya escritos en cada posición de memoria RAM (desde la posición 0 hasta la posición 7).

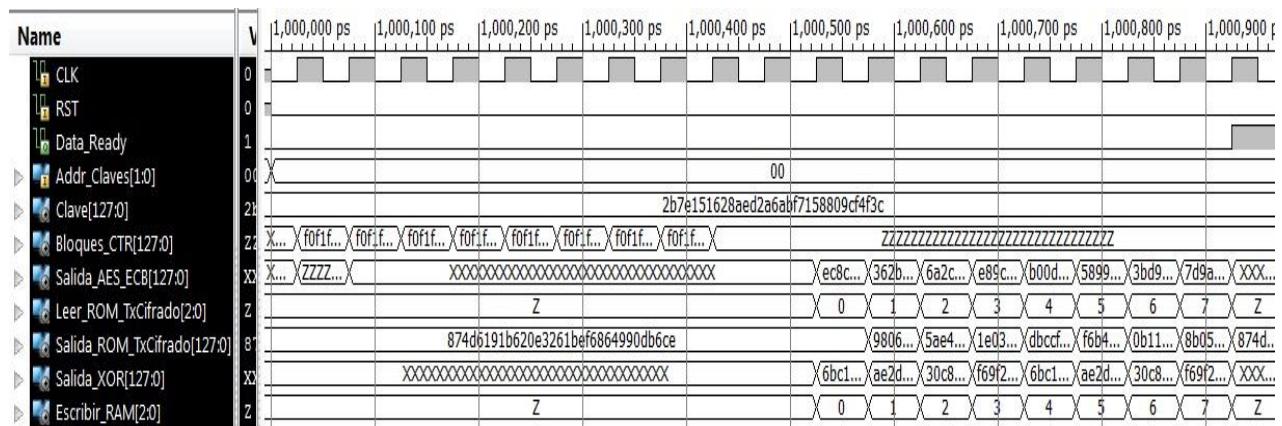


Figura 3.55. Proceso de desencriptación, lectura y escritura entre la unidad de control, las memorias y el módulo de encriptación.

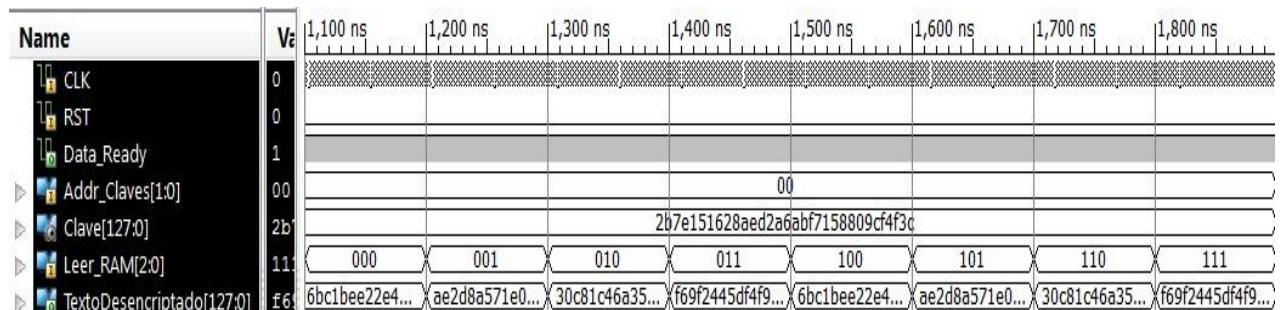


Figura 3.56. Lectura de Datos desencriptados escritos en memoria RAM.

De acuerdo a los resultados de simulación, los datos se han organizado en la tabla 3.5 en la cual se puede apreciar cada dato desencriptado para cada posición de memoria en formato hexadecimal. Se puede verificar que los datos desencriptados se corresponden con el texto plano original.

Tabla 3.5. Resultados de desencriptación en modo contador escritos en memoria *RAM*.

RAM (HEX)	Texto descifrado (HEX) (Escrito en <i>RAM</i>)	Texto cifrado equivalente (HEX)
0	6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a	87 4d 61 91 b6 20 e3 26 1b ef 68 64 99 0d b6 ce
1	ae 2d 8a 57 1e 03 ac 9c 9e b7 6f ac 45 af 8e 51	98 06 f6 6b 79 70 fd ff 86 17 18 7b b9 ff fd ff
2	30 c8 1c 46 a3 5c e4 11 e5 fb c1 19 1a 0a 52 ef	5a e4 df 3e db d5 d3 5e 5b 4f 09 02 0d b0 3e ab
3	f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10	1e 03 1d da 2f be 03 d1 79 21 70 a0 f3 00 9c ee
4	6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a	db cc f9 1a 3a ca 0e 98 19 55 4e 86 e3 d8 b2 28
5	ae 2d 8a 57 1e 03 ac 9c 9e b7 6f ac 45 af 8e 51	f6 b4 ce 0d 53 e2 ad 69 8d 7d be 34 38 26 67 4a
6	30 c8 1c 46 a3 5c e4 11 e5 fb c1 19 1a 0a 52 ef	0b 11 b0 3f ea 82 cf e8 80 92 6d 21 59 f2 20 ad
7	f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10	8b 05 ea c5 98 8c c8 1e b8 71 f9 d3 16 e9 a0 a1

CAPITULO 4: RESULTADOS Y ANALISIS DE IMPLEMENTACIÓN

En este capítulo se presentan los reportes de síntesis dados por el software *Xilinx ISE 13.1 [5]* para cada una de las organizaciones hardware diseñadas en el capítulo 3. Se analizan y comparan características como área, velocidad y consumo potencia según los datos reportados. Además se muestran los resultados de implementación utilizando la *FPGA XC5VLX110T [13]* de la familia *Virtex 5* de *Xilinx* y se visualizan los datos sobre la *FPGA* con la ayuda de la herramienta de verificación de hardware *ChipScope pro 13.1 [14]*. Las estimaciones de consumo de potencia para cada uno de los circuitos se obtuvieron gracias a la herramienta de análisis y diseño *Xilinx PlanAhead 13.1 [15]*.

4.1. Transformaciones

La tabla 4.1 muestra los recursos de *FPGA* reportados para la implementación de cada uno de los módulos que realizan las transformaciones, además se muestra el tiempo de retardo introducido por las rutas y por las puertas lógicas en cada uno de los módulos.

Tabla 4.1. Uso de recursos en *FPGA* y tiempos de retardo reportados para cada módulo de transformación.

Transformación	Recursos de <i>FPGA</i>		Retardo (ns)		Retardo máximo (ns)
	<i>Slice LUT's</i>	<i>Slice Registers</i>	Rutas	Lógica	
SubBytes/ InvSubBytes Tablas	512	0	1.294	3.299	4.593
SubBytes Operación matematica	1058	0	3.286	7.473	11.299
InvSubBytes Operación matematica	1024	0	3.826	6.932	10.758
ShiftRows e InvShiftRows	0	0	0.286	2.838	3.124
MixColumns	155	0	1.490	3.010	4.500
InvMixColumns	370	0	2.187	3.010	5.197
AddRoundKey	128	0	0.772	2.924	3.696
Ronda	688	0	2.651	3.685	6.336
Ronda final	640	0	1.706	3.385	5.091
Ronda inversa	940	0	3.068	3.771	6.839
Ronda inversa final	640	0	1.706	3.385	5.091

De la tabla 4.1. se puede notar que:

- Las transformaciones *SubBytes* e *InvSubBytes* basadas en el modelo matemático requieren alrededor del doble de recursos de *FPGA* para su implementación comparado con el modelo basado en tablas, además el modelo basado en tablas es mucho más rápido que el modelo basado en la operación matemática. [11] y [12] confirman que el modelo basado en tablas es más rápido que el modelo basado en la operación matemática.
- La transformación *InvMixColumns* requiere más recursos de *FPGA* que la transformación *MixColumns* debido a que la primera requiere implementar más funciones de multiplicación, además es un poco más lenta.
- Las transformaciones *ShiftRows* e *InvShiftRows* no requieren de *slice LUT's* ó de *slice registers* debido a que son simples líneas de interconexión, por lo tanto estas transformaciones solo requieren conmutar las conexiones de la salida de la transformación *SubBytes* a la entrada de la transformación *MixColumns*.
- Los recursos y retardos de la ronda normal final y de la ronda inversa final son iguales debido a que en ninguno de estos se implementa la transformación de *MixColumns* ó *InvMixColumns* cuyas diferencias en cuanto a recursos y retardos son considerables.
- Los recursos totales de ronda no se pueden calcular como la suma de los recursos totales de cada transformación, debido a que al unir los módulos de transformación se reorganizan los elementos lógicos de manera que el bloque ronda se ve como una unidad.
- El tiempo total de retardo de ronda no se calcula como la suma de retardos totales de cada transformación, ya que al unir los módulos de transformación, estos se reorganizan de tal manera que el bloque ronda se ve como una unidad con un solo retardo.

4.2. Unidad de subclaves

La tabla 4.2 muestra los recursos de *FPGA* y retardos reportados para la implementación de la unidad de subclaves iterativa y en cascada.

Tabla 4.2. Uso de recursos en *FPGA* y tiempos de retardo reportados para la unidad de subclaves.

Unidad de subclaves	Recursos de <i>FPGA</i>		Retardo (ns)	Fmáx (MHz)
	<i>Slice LUT's</i>	<i>Slice Registers</i>		
Iterativa	534	264	50 (200Mhz) (10 ciclos de reloj)	864.155
Cascada	3391	0	26.005 (máx.)	—
			Lógica 8.665 (33.3%)	
			Ruta 17.340 (66.7%)	

De la tabla 4.2 se resalta que la unidad de subclaves paralela requiere mucho más recursos de *FPGA* para su implementación que la unidad de subclaves iterativa, sin embargo se puede notar que la unidad de subclaves paralela es mucho más rápida generando las subclaves que la iterativa. Dado que la implementación se realiza utilizando una frecuencia de reloj de 200Mhz, la unidad de subclaves iterativa debe esperar 10 ciclos de reloj (50ns) para dejar todas las subclaves listas, mientras que la unidad de subclaves en cascada deja todas las subclaves listas con un retardo máximo de tan solo 26.005ns sin depender de una frecuencia de reloj. Por esta razón se eligió integrar la unidad de subclaves paralela en la implementación de cada uno de los circuitos de encriptación y desencriptación propuestos en el capítulo 3.

4.3. Proceso de encriptación en modo *ECB*

La tabla 4.3 muestra los recursos de *FPGA* reportados, frecuencia máxima, *Throughput* y retardos para la implementación del circuito de encriptación *pipeline* para AES-128 en modo de configuración *ECB*.

Tabla 4.3. Uso de recursos en *FPGA*, frecuencia máxima, *throughput* y tiempo de procesamiento para el circuito de encriptación *pipeline* en modo *ECB* de AES-128.

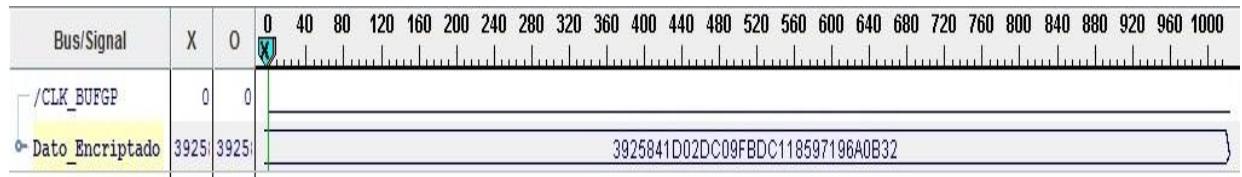
Recursos de <i>FPGA</i>						Tiempo Proc. (ns)	Fmáx (Mhz)	<i>Throughput</i> (Gbits/s)
<i>Slice LUT's</i>		<i>Slice Registers</i>		Pines I/O				
Valor	Uso disp. (%)	Valor	Uso disp. (%)	Valor	Uso disp. (%)	50 (10 ciclos de reloj)	272.59	25.6
11359	16	1289	1	387	60			

Para realizar la implementación en *FPGA*, los vectores de prueba dados por [1] se escribieron en memorias *ROM* de tal manera que se pudiera suministrar la clave y matriz de estado al circuito de encriptación *pipeline* en modo *ECB*. Los siguientes son los valores escritos en memoria *ROM* utilizados como entrada de datos (*matriz de estado*) y clave de encriptación, cada uno de 16 bytes ($N_b = 4$ y $N_k = 4$), en formato hexadecimal:

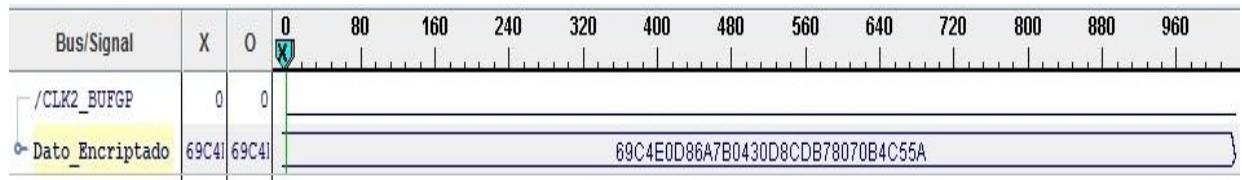
Texto Plano 1 = 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
Clave de encriptación 1 = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Texto Plano 2 = 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Clave de encriptación 2 = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

Los resultados se pueden observar con la ayuda de la herramienta *Chipscope pro 13.1* en la figura 4.1.



a) Usando el texto plano 1



b) Usando el texto plano 2

Figura 4.1. Resultados de encriptación en modo *ECB* obtenidos con *Chipscope pro*.

La estimación de consumo de potencia para la implementación se obtuvo con la ayuda de la herramienta para análisis de diseño *Xilinx planahead 13.1* y se muestra en la figura 4.2. La potencia total consumida es de 1572mW, los bloques I/O consumen 138mW, lo cual representa el 9% de la potencia total. El núcleo dinámico (lógica y reloj) consume 242mW, lo cual representa solo el 15% de la potencia total. De este 15% el 12% (29mW) lo consume el reloj y el 88% (213mW) la lógica, el restante 76%(1192mW) del total de consumo de potencia corresponde al consumo del dispositivo en estado estático.

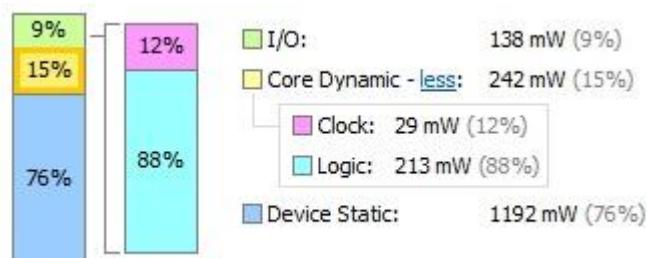


Figura 4.2. Estimación de consumo de potencia para la implementación del circuito de encriptación en modo *ECB*.

4.3.1. Sistema de encriptación en modo *ECB* para múltiples bloques

La tabla 4.4 muestra los recursos de *FPGA* reportados, frecuencia máxima, *Throughput* y retardos para la implementación del circuito de encriptación *pipeline* en modo *ECB* para encriptar múltiples bloques de datos.

Tabla 4.4. Uso de recursos en *FPGA*, frecuencia máxima, *throughput* y tiempo de procesamiento para el circuito de encriptación *pipeline* en modo *ECB* para múltiples bloques de 128 bits.

Recursos de FPGA								Tiempo Proc. (ns)	Fmáx (Mhz)	<i>Throughput</i> (Gbits/s)
Slice LUT's		Slice Registers		Pines I/O		Bloques RAM				
Valor	Uso disp. (%)	Valor	Uso disp. (%)	Valor	Uso disp. (%)	Valor	Uso disp. (%)			
11619	16	2554	3	136	21	2	1	190 (19 ciclos de reloj)	225.746	12.8

El texto plano utilizado para realizar la implementación en *FPGA* del circuito de encriptación *pipeline* en modo *ECB* para 8 bloques se muestra como sigue en formato ASCII y hexadecimal respectivamente:

Texto plano:

ASCII:

“Estos son bloques de texto plano utilizados para probar la implementación en hardware del algoritmo de Rijndael para 128 bits”

Clave: “Ian_Carlo_Guzman”

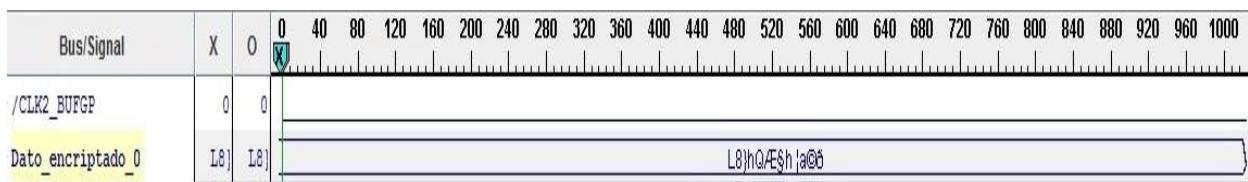
HEX:

**45 73 74 6f 73 20 73 6f 6e 20 62 6c 6f 71 75 65
73 20 64 65 20 74 65 78 74 6f 20 70 6c 61 6e 6f
20 75 74 69 6c 69 7a 61 64 6f 73 20 70 61 72 61
20 70 72 6f 62 61 72 20 6c 61 20 69 6d 70 6c 65
6d 65 6e 74 61 63 69 6f 6e 20 65 6e 20 68 61 72
64 77 61 72 65 20 64 65 6c 20 61 6c 67 6f 72 69
74 6d 6f 20 64 65 20 52 69 6e 6a 64 61 65 6c 20
70 61 72 61 20 31 32 38 20 62 69 74 73 00 00 00**

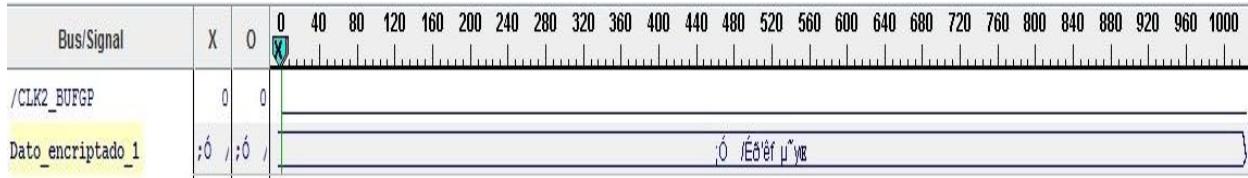
Clave:

49 61 6e 5f 43 61 72 6c 6f 5f 47 75 7a 6d 61 6e

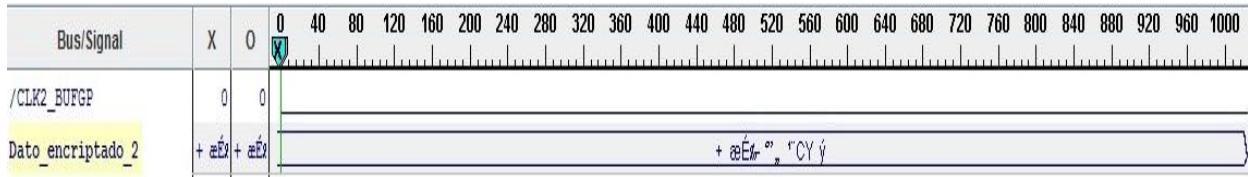
Los resultados se observan con la ayuda de la herramienta *ChipScope pro 13.1*. Como se puede apreciar en la figura 4.3, cada dato encriptado se muestra en formato ASCII como un conjunto de caracteres incoherentes e incomprendibles para algún lector, además se puede observar que los resultados son los mismos obtenidos en simulación.



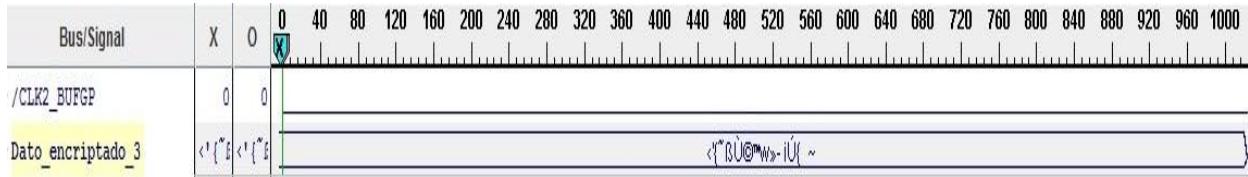
(a) Posición 0.



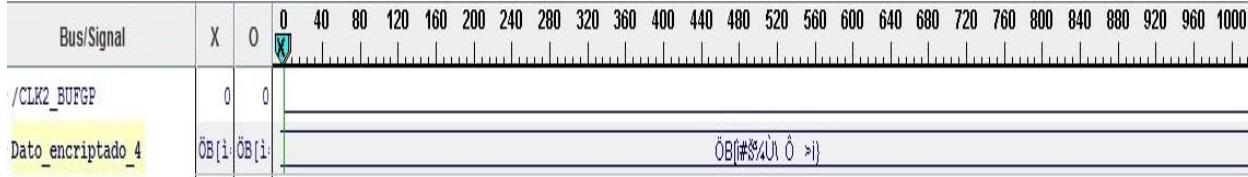
(b) Posición 1.



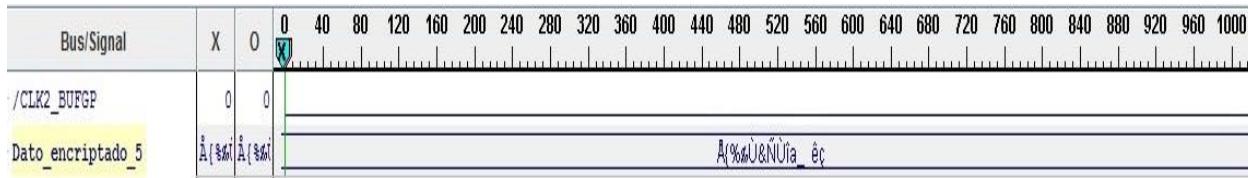
(c) Posición 2.



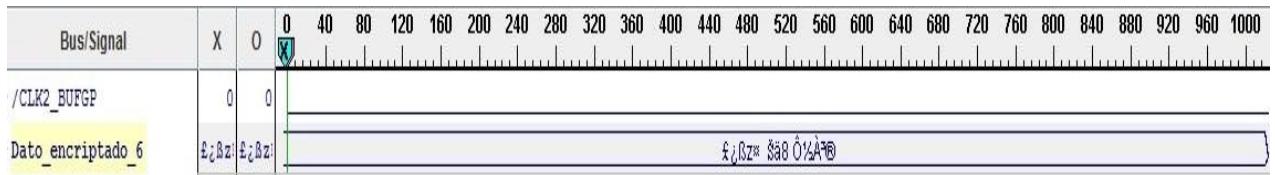
(d) Posición 3.



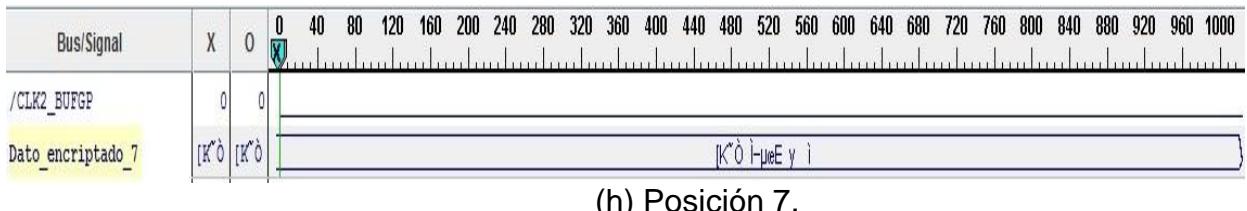
(e) Posición 4.



(f) Posición 5.



(g) Posición 6.



(h) Posición 7.

Figura 4.3. Datos encriptados en modo *ECB* leídos en cada posición de la memoria RAM sobre la *FPGA* usando *Chipscope pro*.

La estimación de consumo de potencia para esta implementación se muestra en la figura 4.4. Se puede observar que la potencia total consumida es de 1612mW donde los bloques de entrada/salida consumen 133mW, lo cual representa el 8% de la potencia total consumida. El núcleo dinámico (lógica, bloques de memoria y reloj) consume 286mW, lo cual representa el 18% de la potencia total, distribuido con el 21% (61mW) para el reloj, 74% (211mW) para la lógica y 5%(14mW) para los bloques de memoria. El restante 74%(1193mW) del total es consumido por el dispositivo en estado estático.

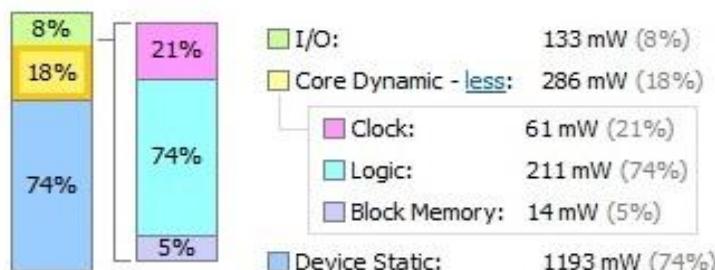


Figura 4.4. Estimación de consumo de potencia para la implementación del circuito de encriptación para multiples bloques en modo *ECB*.

4.4. Proceso de desencriptación en modo *ECB*

La tabla 4.5 muestra los recursos de *FPGA* reportados, frecuencia máxima, *Throughput* y retardos para la implementación del circuito de desencriptación *pipeline* para AES-128 en modo *ECB*.

Tabla 4.5. Uso de recursos en *FPGA*, frecuencia máxima, *Throughput* y tiempo de procesamiento para el circuito de desencriptación *pipeline* en modo *ECB* de AES-128.

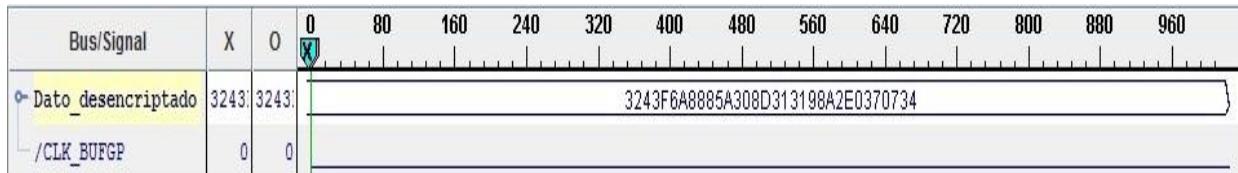
Recursos de <i>FPGA</i>						Tiempo Proc. (ns)	Fmáx (Mhz)	<i>Throughput</i> (Gbits/s)
Slice LUT's		Slice Registers		Pines I/O				
Valor	Uso total dispositivo (%)	Valor	Uso total dispositivo (%)	Valor	Uso total dispositivo (%)	50 (10 ciclos de reloj)	199.486	25.6
13952	20	1289	1	387	60			

Para realizar la implementación en *FPGA*, los vectores de prueba se escribieron en memorias *ROM* de tal manera que se pudiera suministrar la clave y matriz de estado. Los siguientes son los valores escritos en memoria *ROM* utilizados como entrada de datos (*matriz de estado*) y clave de desencriptación, cada uno de 16 bytes ($N_b = 4$ y $N_k = 4$), en formato hexadecimal:

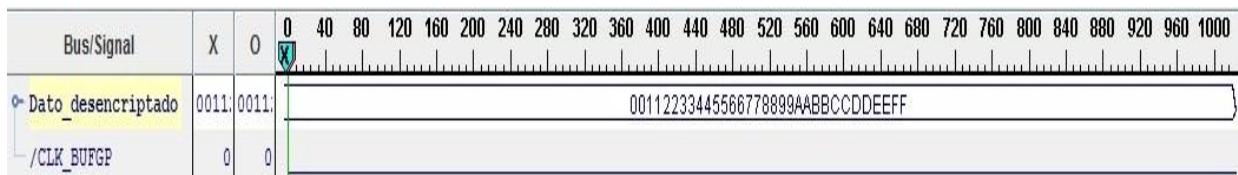
Texto cifrado 1 = 39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32
Clave desencriptación 1 = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Texto cifrado 2 = 69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
Clave desencriptación 2 = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

Los resultados se pueden observar con la ayuda de la herramienta *Chipscope pro 13.1* en la figura 4.5 y se puede comprobar que los resultados son los mismos obtenidos en simulación.



b) Usando el texto cifrado 1.



b) Usando el texto cifrado 2.

Figura 4.5. Resultados de desencriptación en modo *ECB* obtenidos con *Chipscope pro*.

La estimación de consumo de potencia para la implementación se obtuvo con la ayuda de la herramienta para análisis de diseño *Xilinx planahead 13.1* y se muestra en la figura 4.6, en la que se puede notar que la potencia total consumida es de 1613mW donde los pines de entrada/salida consumen 138mW, lo cual representa el 9% de la potencia total consumida. El núcleo dinámico (lógica y reloj) consume 282mW, lo cual representa el 17% de la potencia total consumida, distribuido con el 10% (29mW) para el reloj y el 90% (253mW) para la lógica, el restante 74%(1193mW) del total de consumo de potencia lo consume el dispositivo en estado estático.

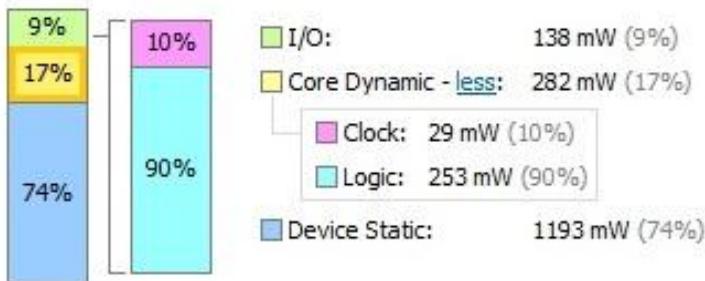


Figura 4.6. Estimación de consumo de potencia para la implementación del circuito de desencriptación en modo *ECB*.

4.4.1. Sistema de desencriptación en modo *ECB* para múltiples bloques

La tabla 4.6 muestra los recursos de *FPGA* reportados, frecuencia máxima, *Throughput* y retardos para la implementación del circuito de desencriptación *pipeline* en modo *ECB* para desencriptar múltiples bloques.

Tabla 4.6. Uso de recursos en *FPGA*, frecuencia máxima, *Throughput* y tiempo de procesamiento para el circuito de desencriptación *pipeline* en modo *ECB* para múltiples bloques.

Recursos de <i>FPGA</i>								Tiempo Proc. (ns)	Fmáx (Mhz)	<i>Throughput</i> (Gbits/s)
Slice LUT's		Slice Registers		Pines I/O		Bloques RAM				
Valor	Uso disp. (%)	Valor	Uso disp. (%)	Valor	Uso disp. (%)	Valor	Uso disp. (%)	190 (19 ciclos de reloj)	179.513	12.8
14051	20	2555	3	136	21	2	1			

El texto cifrado y clave para realizar la implementación se muestra a continuación en formato hexadecimal:

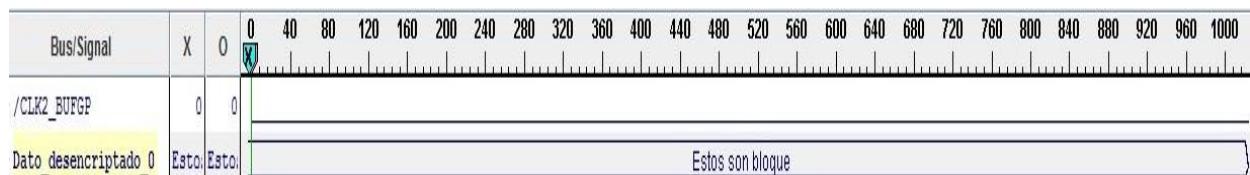
Texto Cifrado:

```
06 04 0a 4c 38 7d 68 51 c6 a7 68 a0 a6 61 a9 f0
3b d3 14 10 2f c9 f0 27 ea 66 1b 0d b5 98 79 8c
2b 95 e6 c9 89 2d 91 94 84 18 b9 a8 43 59 9d fd
8b 27 7b 98 df d9 a9 99 77 bb 96 69 da 7b 03 7e
d6 42 5b ec 23 8a be d9 5c 02 d4 20 03 3e 69 7d
c5 7b 25 89 d9 26 d1 d9 ee 0d 09 61 5f 8f ea e7
a3 bf df 7a a4 8f 8a e4 38 a0 09 d4 bd c0 b3 ae
5b 4b 98 d2 1e cc 97 b5 9c 45 0b 79 08 95 ec 05
```

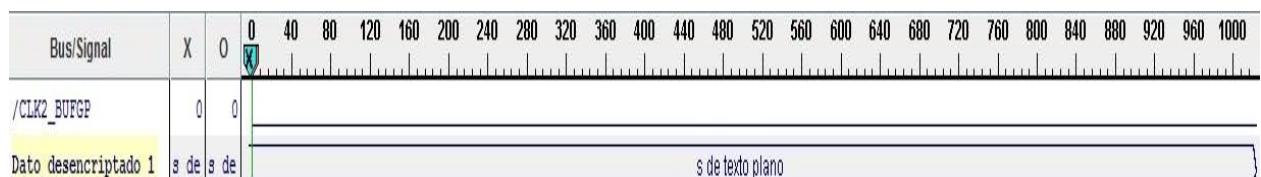
Clave:

49 61 6e 5f 43 61 72 6c 6f 5f 47 75 7a 6d 61 6e

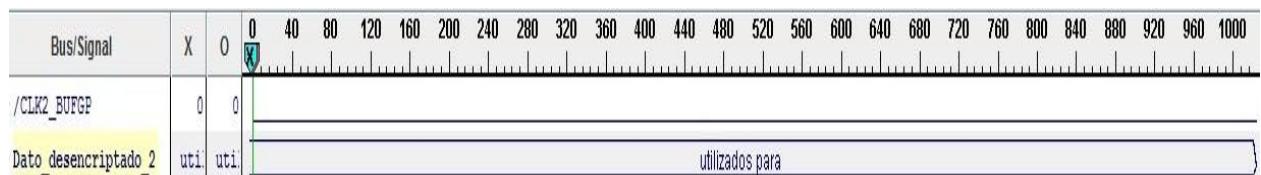
Los resultados se observan con la ayuda de la herramienta *ChipScope pro 13.1* en la figura 4.7. Como se puede apreciar cada dato desencriptado se muestra en formato ASCII. También se puede observar que los resultados son los mismos obtenidos en simulación.



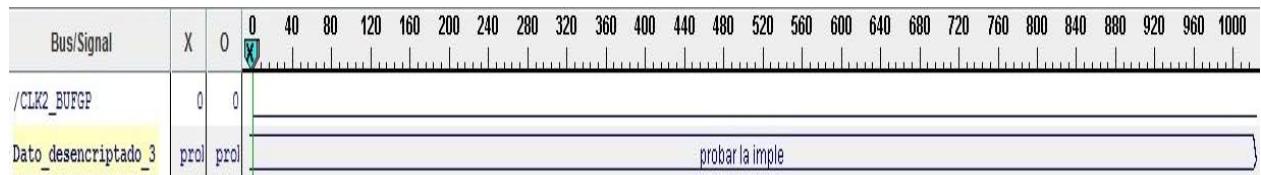
(a) Posición 0.



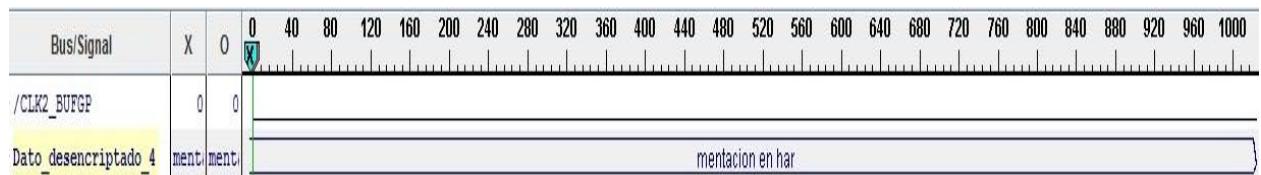
(b) Posición 1.



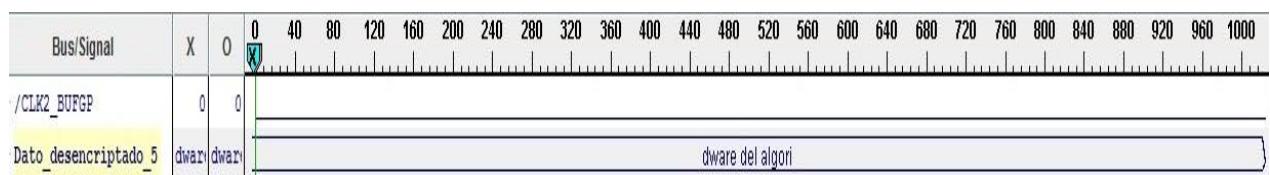
(c) Posición 2.



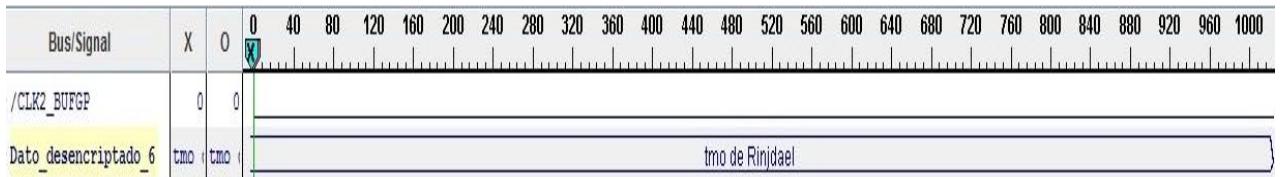
(d) Posición 3.



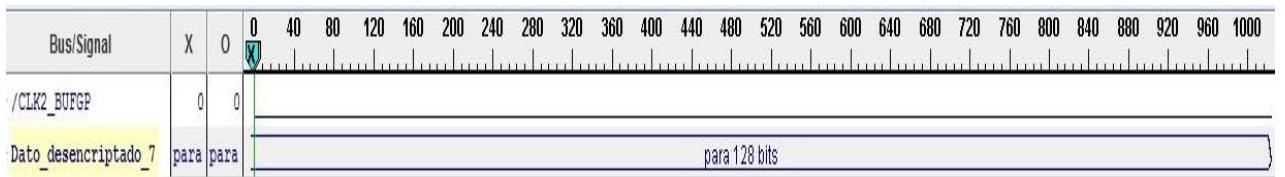
(e) Posición 4.



(f) Posición 5.



(g) Posición 6.



(a) Posición 7.

Figura 4.7. Datos desencriptados en modo *ECB* leídos en cada posición de la memoria *RAM* sobre la *FPGA* usando *Chipscope pro*.

La estimación de consumo de potencia para la implementación se obtuvo con la ayuda de la herramienta para análisis de diseño *Xilinx planahead 13.1* y se muestra en la figura 4.8, en la que se puede notar que la potencia total consumida es de 1627 mW donde los pines de entrada/salida consumen 133mW, lo cual representa el 8% de la potencia total consumida. El núcleo dinámico (lógica, bloques de memoria y reloj) consume 301mW, lo cual representa el 18% de la potencia total consumida, distribuido con el 20% (61mW) para el reloj, el 75% (226mW) para la lógica y el 5%(14mW) para los bloques de memoria. El restante 74%(1193mW) del total de consumo de potencia lo consume el dispositivo en estado estático.

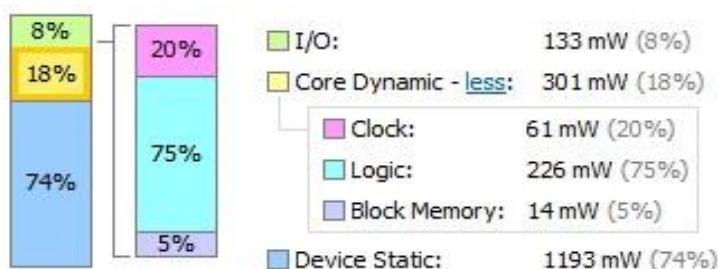


Figura 4.8. Estimación de consumo de potencia para la implementación del circuito de desencriptación en modo *ECB* para multiples bloques.

4.5. Proceso de Encriptación en modo *CTR*

La tabla 4.7 muestra los recursos de *FPGA* reportados, frecuencia máxima, *Throughput* y retardos para la implementación del circuito de encriptación/desencriptación en modo *CTR*.

Tabla 4.7. Uso de recursos en *FPGA*, frecuencia máxima, *throughput* y tiempo de procesamiento para el circuito de encriptación/desencriptación *pipeline* en modo *Counter* (*CTR*).

Recursos de <i>FPGA</i>						Tiempo Proc. (ns)	Fmáx (Mhz)	<i>Throughput</i> (Gbits/s)
<i>Slice LUT's</i>		<i>Slice Registers</i>		Pines I/O				
Valor	Uso disp. (%)	Valor	Uso disp. (%)	Valor	Uso disp. (%)			
11677	16	1484	2	387	60	55 (11 ciclos de reloj)	272.59	12.8

La estimación de consumo de potencia para la implementación se obtuvo con la ayuda de la herramienta para análisis de diseño *Xilinx planahead 13.1* y se muestra en la figura 4.9, en la que se puede notar que la potencia total consumida es de 1649 mW donde los pines de entrada/salida consumen 209mW, lo cual representa el 13% de la potencia total consumida. El núcleo dinámico (lógica, bloques de memoria y reloj) consume 246mW, lo cual representa el 15% de la potencia total consumida, distribuido con el 13% (33mW) para el reloj y el 87% (214mW) para la lógica. El restante 72%(1194mW) del total de consumo de potencia lo consume el dispositivo en estado estático.

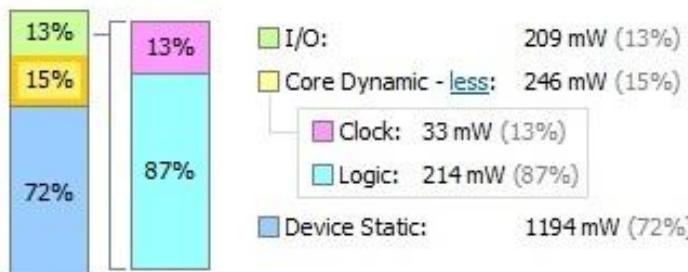


Figura 4.9. Estimación de consumo de potencia para la implementación del circuito de encriptación/desencriptación en modo *CTR*.

La verificación de la implementación con los vectores de prueba se muestra en la siguiente sección (sección 4.5.1) en la figura 4.10.

4.5.1 Sistema de encriptación en modo *CTR* para múltiples bloques

La tabla 4.8 muestra los recursos de *FPGA* reportados, frecuencia máxima, *Throughput* y retardos para la implementación del circuito de encriptación/desencriptación *pipeline* en modo *CTR* para encriptar múltiples bloques.

Tabla 4.8. Uso de recursos en *FPGA*, frecuencia máxima, *throughput* y tiempo de procesamiento para el circuito de encriptación/desencriptación *pipeline* en modo *Counter* (*CTR*) para múltiples bloques.

Recursos de FPGA								Tiempo Proc. (ns)	Fmáx (Mhz)	<i>Throughput</i> (Gbits/s)
Slice LUT's		Slice Registers		Pines I/O		Bloques RAM				
Valor	Uso disp. (%)	Valor	Uso disp. (%)	Valor	Uso disp. (%)	Valor	Uso disp. (%)			
11769	17	2720	3	135	21	2	1	210 (21 ciclos de reloj)	269.219	12.8

El texto plano y clave para realizar la implementación se muestra a continuación en formato hexadecimal:

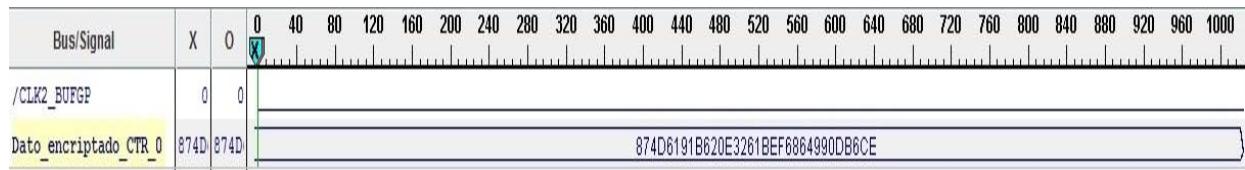
Texto Plano:

```
6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
ae 2d 8a 57 1e 03 ac 9c 9e b7 6f ac 45 af 8e 51
30 c8 1c 46 a3 5c e4 11 e5 fb c1 19 1a 0a 52 ef
f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10
6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
ae 2d 8a 57 1e 03 ac 9c 9e b7 6f ac 45 af 8e 51
30 c8 1c 46 a3 5c e4 11 e5 fb c1 19 1a 0a 52 ef
f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10
```

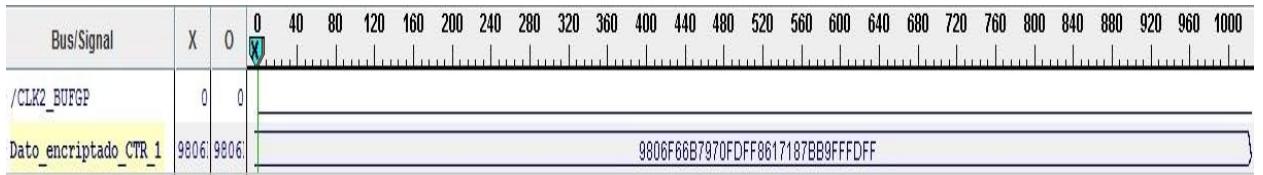
Clave:

```
2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
```

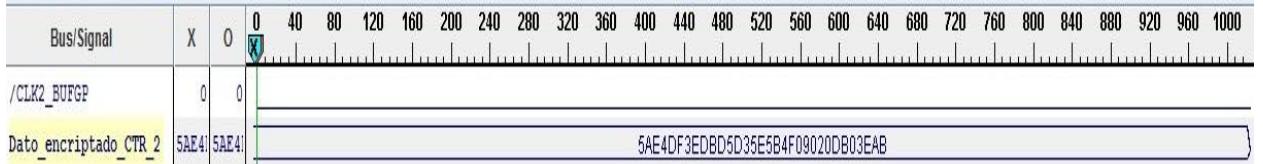
Los resultados se observan con la ayuda de la herramienta *Chipscope pro 13.1* en la figura 4.10. Se puede observar que los resultados son los mismos obtenidos en simulación.



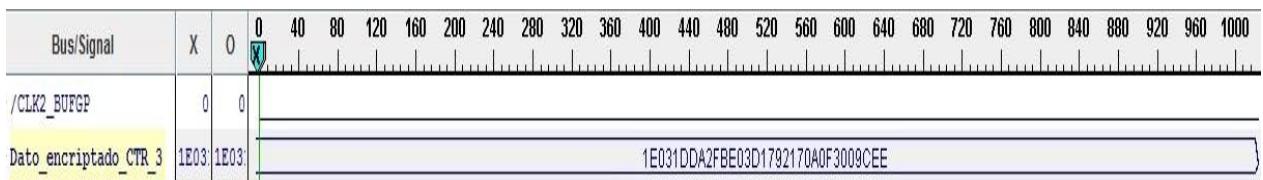
(a) Posición 0.



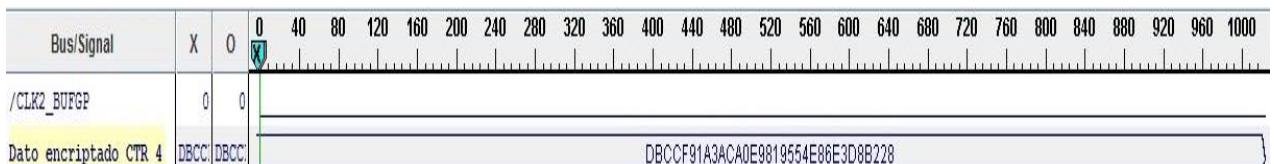
(b) Posición 1.



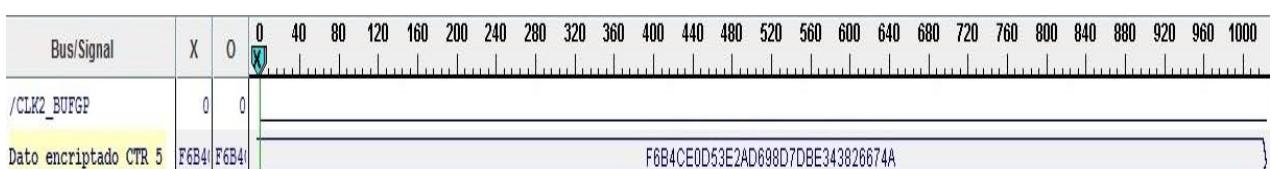
(c) Posición 2.



(d) Posición 3.



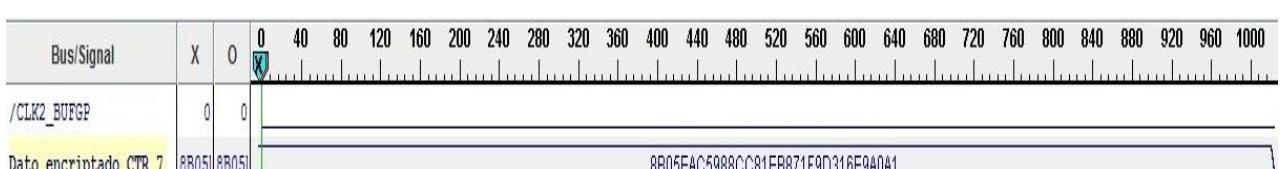
(e) Posición 4.



(f) Posición 5.



(g) Posición 6.



(f) Posición 7.

Figura 4.10. Datos encriptados en modo CTR leídos en cada posición de la memoria RAM sobre la FPGA usando *ChipScope pro*.

La estimación de consumo de potencia para la implementación se obtuvo con la ayuda de la herramienta para análisis de diseño *Xilinx planahead13.1* y se muestra en la figura 4.11, en la que se puede notar que la potencia total consumida es de 1608 mW donde los pines de entrada/salida consumen 133mW, lo cual representa el 8% de la potencia total consumida. El núcleo dinámico (lógica, bloques de memoria y reloj) consume 282 mW, lo cual representa el 18% de la potencia total consumida, distribuido con el 23% (64 mW) para el reloj, el 72% (203mW) para la lógica y el 5% (14mW) para los bloques de memoria, el restante 74%(1193mW) del total de consumo de potencia lo consume el dispositivo en estado estático.

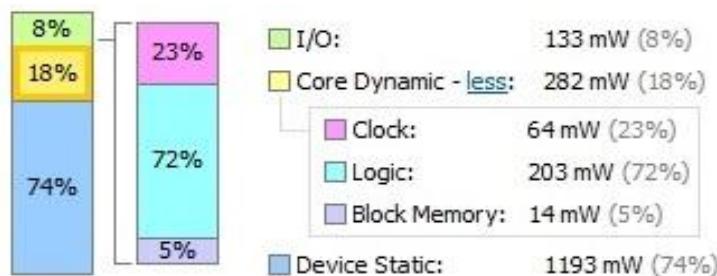


Figura 4.11. Estimación de consumo de potencia para la implementación del circuito de encriptación/desencriptación para multiples bloques en modo *CTR*.

4.6. Proceso de desencriptación en modo de operación *CTR* (*Counter*)

El circuito implementado en *FPGA* para llevar a cabo el proceso de desencriptación en modo *CTR* es el mismo implementado para llevar a cabo el proceso de encriptación, así que en vez de ingresar texto plano se ingresa el texto cifrado.

El texto cifrado y clave para realizar la implementación se muestra a continuación en formato hexadecimal y los resultados se observan con la ayuda de la herramienta *Chipscope pro 13.1* en la figura 4.7.

Texto cifrado:

```
87 4d 61 91 b6 20 e3 26 1b ef 68 64 99 0d b6 ce
98 06 f6 6b 79 70 fd ff 86 17 18 7b b9 ff fd ff
5a e4 df 3e db d5 d3 5e 5b 4f 09 02 0d b0 3e ab
1e 03 1d da 2f be 03 d1 79 21 70 a0 f3 00 9c ee
db cc f9 1a 3a ca 0e 98 19 55 4e 86 e3 d8 b2 28
f6 b4 ce 0d 53 e2 ad 69 8d 7d be 34 38 26 67 4a
0b 11 b0 3f ea 82 cf e8 80 92 6d 21 59 f2 20 ad
8b 05 ea c5 98 8c c8 1e b8 71 f9 d3 16 e9 a0 a1
```

Clave:

```
2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
```

Bus/Signal	X	0	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720	760	800	840	880	920	960	1000
/CLK2_BUFGP		0	0																									
Dato_desencriptado_CTR_0	6BC1	6BC1																										

(a) Posición 0.

Bus/Signal	X	0	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720	760	800	840	880	920	960	1000
/CLK2_BUFGP		0	0																									
Dato_desencriptado_CTR_1	AE2D	AE2D																										

(b) Posición 1.

Bus/Signal	X	0	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720	760	800	840	880	920	960	1000
/CLK2_BUFGP		0	0																									
Dato_desencriptado_CTR_2	30C8	30C8																										

(c) Posición 2.

Bus/Signal	X	0	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720	760	800	840	880	920	960	1000
/CLK2_BUFGP		0	0																									
Dato_desencriptado_CTR_3	F69F	F69F																										

(d) Posición 3.

Bus/Signal	X	0	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720	760	800	840	880	920	960	1000
/CLK2_BUFGP		0	0																									
Dato_desencriptado_CTR_4	6BC1	6BC1																										

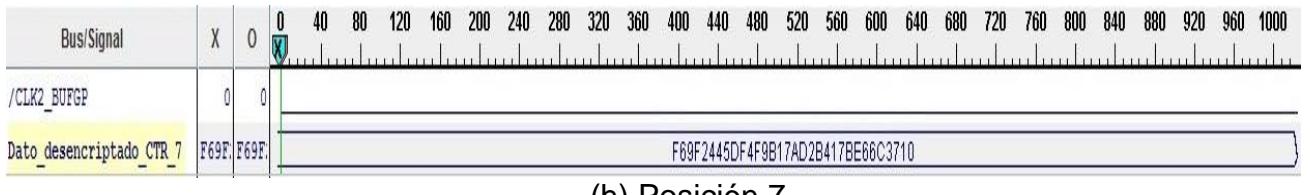
(e) Posición 4.

Bus/Signal	X	0	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720	760	800	840	880	920	960	1000
/CLK2_BUFGP		0	0																									
Dato_desencriptado_CTR_5	AE2D	AE2D																										

(f) Posición 5.

Bus/Signal	X	0	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720	760	800	840	880	920	960	1000
/CLK2_BUFGP		0	0																									
Dato_desencriptado_CTR_6	30C8	30C8																										

(g) Posición 6.



(h) Posición 7.

Figura 4.12. Datos desencriptados en modo *CTR* leídos en cada posición de la memoria *RAM* sobre la *FPGA* usando *Chipscope pro*.

4.7. Análisis de resultados de los modos de operación.

La tabla 4.9 muestra los datos reportados y los resultados obtenidos en cada uno de los modos de operación teniendo en cuenta el área, la frecuencia máxima, el *throughput* y el consumo de potencia.

Tabla 4.9. Área, frecuencia máxima, *throughput* y consumo de potencia para la encriptación y desencriptación en los modos *ECB* y *CTR*.

Modo	Área			Tiempo Proc. (ns)	Fmáx (Mhz)	<i>Throughput</i> (Gbits/s)	Consumo de potencia (mW)			
	<i>Slice LUT's</i>	<i>Slice Reg.</i>	Pines I/O				I/O	Reloj y lógica	Disp. estático	Total
Enc. <i>ECB</i>	11359	1289	387	100	272.59	25.6	138	242	1192	1572
Des. <i>ECB</i>	13952	1289	387	100	199.48	25.6	138	282	1193	1613
Enc/D ec <i>CTR</i>	11677	1484	387	110	272.59	25.6	209	246	1194	1649

De la tabla 4.9 se puede notar que el circuito de desencriptación en modo *ECB* requiere de más área que el circuito de encriptación en modo *ECB*, por lo tanto su lógica requiere de un mayor consumo de potencia, además se puede notar que el retardo para obtener el primer dato a la salida es el mismo en la encriptación y desencriptación en modo *ECB*. También se puede apreciar que el circuito de encriptación/desencriptación en modo *CTR* requiere de más área que el circuito de encriptación *ECB* pero de menos área de la que requiere el circuito de desencriptación *ECB*, no obstante el circuito de encriptación/desencriptación en modo *CTR* requiere de un mayor consumo de potencia debido al alto consumo de los pines de entrada/salida. De otro lado se puede notar que el *throughput* de cada uno de los circuitos es el mismo debido a que trabajan a la misma frecuencia de reloj (200Mhz) y que el retardo del circuito de encriptación/desencriptación en modo *CTR* es ligeramente mayor en obtener el primer dato encriptado o desencriptado comparado con el modo *ECB*. Finalmente se puede observar que el desempeño en frecuencia para el proceso de encriptación en los modos *ECB* y *CTR* son iguales pero el proceso de desencriptación en modo *ECB* presenta el desempeño en frecuencia más bajo.

4.7.1 Sistema de encriptación y desencriptación en los modos *ECB* y *CTR* para múltiples bloques.

Tabla 4.10. Área, frecuencia máxima, *throughput* y consumo de potencia para la encriptación y desencriptación en los modos *ECB* y *CTR* para múltiples bloques.

Modo	Área				Tiempo Proc. (ns)	Fmáx (Mhz)	<i>Throughput</i> (Gbits/s)	Consumo potencia (mW)
	<i>Slice LUT's</i>	<i>Slice Registers</i>	Pines I/O	BRAM				
Enc. <i>ECB</i>	11619	2554	136	2	190	225.74	12.8	1612
Des. <i>ECB</i>	14051	2555	136	2	190	179.51	12.8	1627
Enc/Dec <i>CTR</i>	11769	2720	135	2	210	269.22	12.8	1608

De la tabla 4.10 se puede notar que el circuito de encriptación/desencriptación para 8 bloques de texto plano o cifrado en modo contador (*CTR*) utiliza más área que el circuito de encriptación en modo *ECB*, pero menos área que el circuito de desencriptación en modo *ECB*, no obstante en el modo *CTR* se utilizan más registros que en el modo *ECB* debido a que se utiliza un contador de 64 bits y otros registros para formar el bloque de contador.

También se puede apreciar que el retardo es ligeramente mayor en el modo *CTR* con respecto al modo *ECB* y que el *throughput* es el mismo para todos los casos debido que se utilizó la misma frecuencia de reloj (100Mhz) en los dos modos de operación. Con respecto a la estimación de consumo de potencia se puede notar que el circuito de desencriptación en modo *ECB* es el que mayor consumo presenta debido a todos los recursos que utiliza, por el contrario el circuito de encriptación/desencriptación en modo *CTR* es el que menos consumo requiere. Finalmente, se puede observar que los desempeños en frecuencia para los sistemas de encriptacion y desencriptacion en modo *CTR* son mucho más altos con respecto a los sistemas de encriptación y desencriptación en modo *ECB*.

4.8. Estado del arte del algoritmo de *Rijndael* para sistemas reprogramables

Hay una gran variedad de arquitecturas para realizar la implementación en hardware del algoritmo de Rijndael en sistemas reprogramables, sin embargo, la mayoría de estas se pueden clasificar de acuerdo a dos arquitecturas básicas [6], [21]:

- **Arquitectura iterativa (Rolling Architecture):** En esta arquitectura se utiliza un modelo de realimentación donde los datos se transforman de forma iterativa sobre el bloque ronda. Este esquema utiliza poca área, pero un tiempo de procesamiento largo, por lo tanto el *throughput* es bajo.
- **Arquitectura no iterativa (Unrolling Architecture):** En esta arquitectura se conectan registros *pipeline* entre las rondas, por lo tanto se obtiene un *throughput* alto, pero el área utilizada aumenta considerablemente.

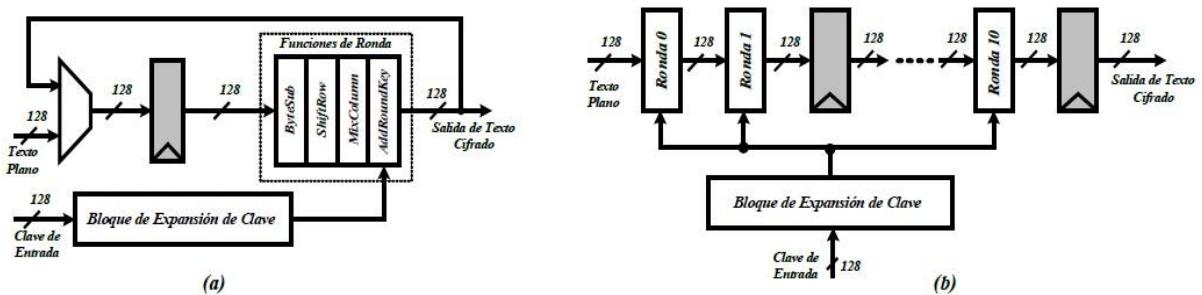


Figura 4.13. Arquitecturas de *Rijndael* para 128 bits: a) *Rolling*; b) *Unrolling*; [6],[21]

4.8.1. Arquitectura *pipeline* de ronda interna y ronda externa

Esta arquitectura utiliza registros entre cada una de las transformaciones y entre cada ronda, de manera que el tiempo de procesamiento entre cada ronda son cuatro ciclos de reloj, esto implica que se pueden utilizar frecuencias altas de reloj por que el tiempo de procesamiento entre cada transformación es corto, sin embargo, el número de ciclos de reloj a utilizar aumenta ya que se deben ejecutar 41 ciclos de reloj.

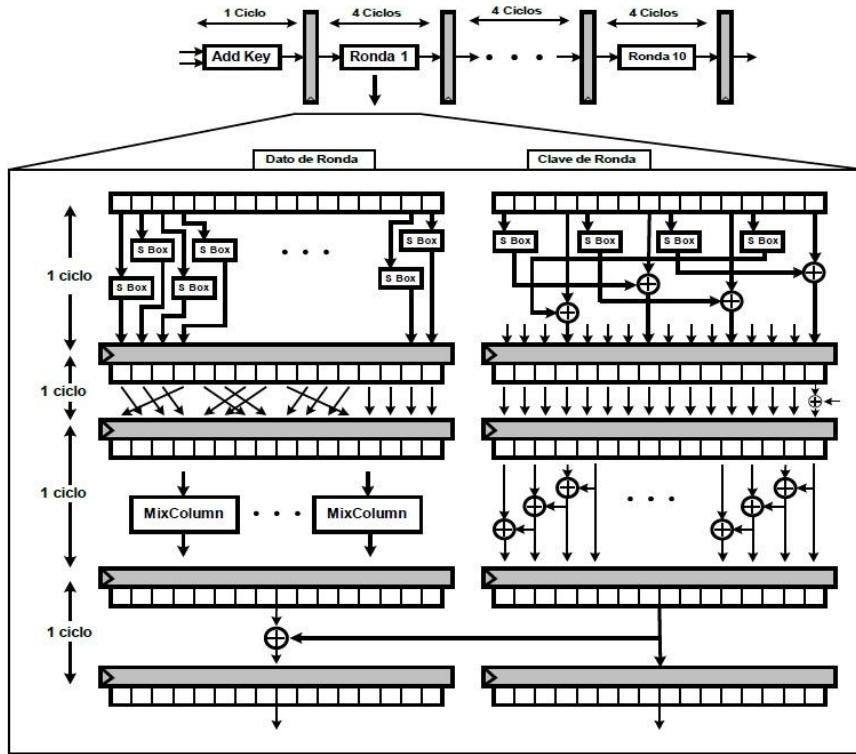


Figura 4.14. Arquitectura *pipeline* de ronda interna y de ronda externa ó *Subpipeline*. [6], [20]

4.8.2. Arquitectura *pipeline* de ronda externa

En esta arquitectura se eliminan los registros entre cada transformación, por lo tanto el tiempo de procesamiento para cada ronda es de un ciclo de reloj, así que se puede obtener el primer dato encriptado en diez ciclos de reloj.

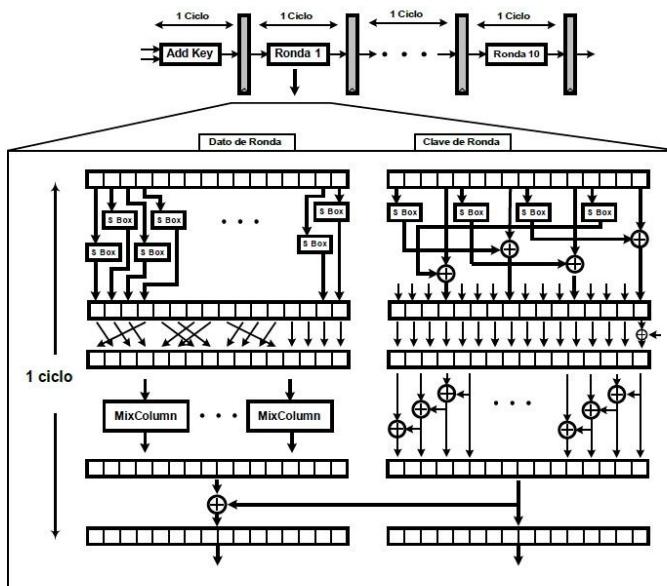


Figura 4.15. Arquitectura *pipeline* de ronda externa.[6], [20]

4.8.3. Arquitectura *pipeline* multironda

Esta arquitectura está diseñada para obtener una mayor optimización de área. Sobre un mismo *datapath* se implementan dos rondas que incluyen el cálculo de la subclave en paralelo. Para las diez rondas hay un total de cinco etapas *pipeline* y cada etapa toma dos ciclos de reloj, lo cual corresponde a dos rondas del algoritmo. Por tanto, se genera una salida cada dos ciclos.

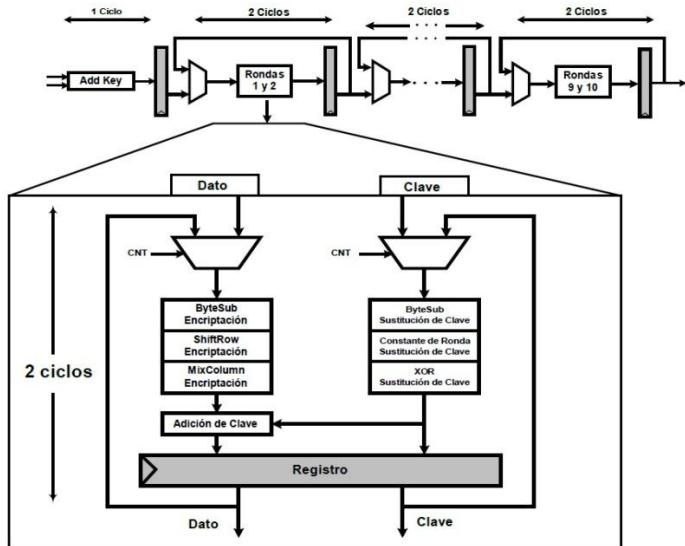


Figura 4.16. Arquitectura *pipeline* de multironda.[6], [20]

4.8.4. Arquitectura *pipeline* de campo compuesto

En [25], [26] se presenta una arquitectura alterna para la implementación de AES. Se trata de una arquitectura que utiliza técnicas *pipeline* de ronda-interna y ronda-externa. Esta arquitectura, mostrada en la figura 4.17, se implementa con base en el uso de un numero óptimo de registros *pipeline* en la fase de Sustitución de Byte y explota, además, la implementación hardware de las Sbox propuesta por Rijmen, quien es uno de los autores del algoritmo.

En la fase de sustitución de byte, la entrada se considera como un elemento en $GF(2^8)$. Primero se calcula el inverso multiplicativo en $GF(2^8)$ y luego se aplica una transformación afín sobre $GF(2)$ [27]. Los valores de sustitución se pueden almacenar en un bloque RAM o se pueden calcular en forma directa para luego implementarlos en un circuito lógico. Ya que el paso de mayor costo en la implementación del algoritmo AES lo constituye la fase de sustitución de byte (Sbox), Rijmen sugirió utilizar un algoritmo que calcula esta fase usando operaciones en $GF(2^4)$.

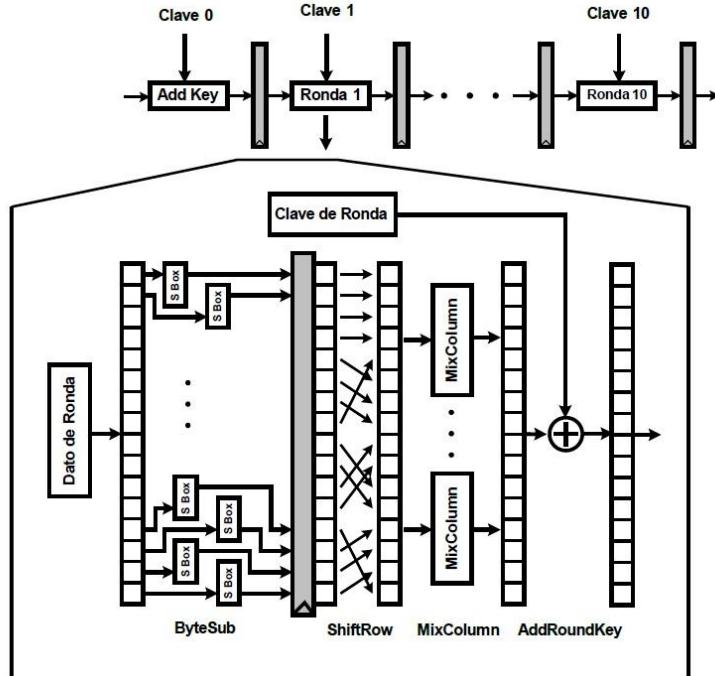


Figura 4.17. Arquitectura *pipeline* de campo compuesto. [6],[25], [26]

La figura 4.18 muestra la arquitectura de la fase de sustitución de byte con la entrada mapeada en elementos de $GF(2^4)$ y donde también se usan operaciones en $GF(2^4)$. Esta constituye la forma más eficiente de implementación de las tablas Sbox [27]. Debido a los retardos de esta arquitectura los diseños más eficientes son aquellos con tres y seis etapas *pipeline*. Las líneas punteadas constituyen registros *pipeline* para sustitución de byte de tres etapas y las líneas llenas son los registros para Sbox de seis etapas. Lo anterior sugiere que la implementación óptima *pipeline* tiene un total de cuatro a siete etapas *pipeline* para cada ronda.

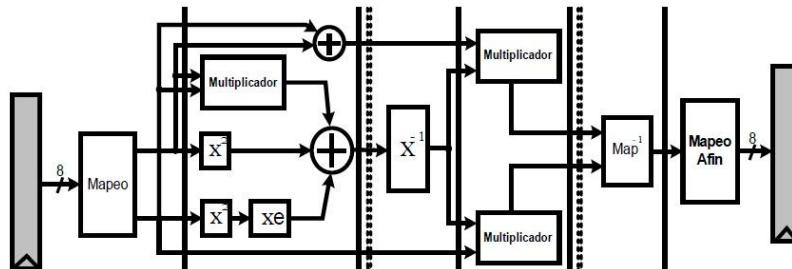


Figura 4.18. *pipeline* de fase de sustitución de byte de campo compuesto. [26]

4.8.5. Reportes de implementaciones en *FPGA* para el algoritmo de *Rijndael* .

La tabla 4.11 muestra una serie de publicaciones importantes en las cuales se implementa el algoritmo de *Rijndael* para sistemas reprogramables. El desempeño

en frecuencia más alto ha sido reportado por [18] con un *throughput* de 73.737 Gbits/s. En las últimas posiciones de la tabla se pueden encontrar reportes con valores altos de *throughput*.

Tabla 4.11. Implementaciones en hardware reprogramable de arquitectura *pipeline* para el algoritmo de *Rijndael*.

Ref.	CLB's	BRAM's	LUT's	Frec (Mhz)	Throu. (Gbits/s)	Comentarios
[18]			22994	576	73.73	- Diseño con xc5vlx85 - Throughput mas alto reportado - Alto desempeño en frecuencia
[19]		10	780	115	1.3	-Sbox en LUT -Pipeline de ronda interna -Subclaves generadas en paralelo -Diseño con Spartan II-5
[29]		10	770	155	1.75	-Sbox en LUT -Pipeline de ronda interna -Subclaves generadas en paralelo -Diseño en virtex E8
[30]	12600				12.2	-Estudio comparativo con otras implementaciones AES -Pipeline de ronda interna y externa -Diseño en Virtex XCV-1000
[31]	2507			32	0.41	-Arquitectura "Full Unrolled" -Diseño con Virtex
[31]	2057		8	99	1.26	- Utiliza una ronda con pipeline - Diseño con Virtex
[31]	12600	80		95	12.6	- Arquitectura Unrolled con pipeline - Diseño con Virtex
[24]	10750			139.1	17.8	- Sbox implementada en forma combinatoria - Arquitectura Full pipeline, ronda interna y externa -Diseño con Virtex II, XC2V2000-5
[24]	11719			129.2	16.54	- Sbox implementada en forma combinatoria - Arquitectura Full pipeline - Diseño con Virtex II, XCV1000-8
[32]	2222	100		54.35	7.0	-Una ronda por ciclo de reloj -Arquitectura pipeline -Hardware para 10 rondas de encriptación -Sbox en ROM (LUT)
[33]		20		2.5	7.6	-Sbox en ROM -Arquitectura pipeline - Hardware para 10 rondas de encriptación - Diseño Virtex II, XC2V1500

[34]	4325	1		75	0.73	<ul style="list-style-type: none"> - Arquitectura pipeline de ronda externa - Claves generadas y almacenadas en memoria antes de encriptación - Diseño en Virtex II, XC2V1000-4
[21]			12847 (*4 LUT)	82	10.49	<ul style="list-style-type: none"> -Arquitectura PPR (<i>Pipeline Partial Rolling</i>) -Diseño con Altera Stratix, 1S20C5
[35]	2052			135.7	1.57	<ul style="list-style-type: none"> -Arquitectura <i>Subpipeline</i> -Diseño con Virtex II
[36]	8141			77.69	4.73	<ul style="list-style-type: none"> -Arquitectura <i>Subpipeline</i> -Subclaves generadas en paralelo -Diseño con Virtex II
[26]	5177	84	8285	168.3	21.54	<ul style="list-style-type: none"> -Arquitectura pipeline de ronda interna y externa -Diseño con Virtex II pro.
[37]	163	3		71.5	0.20	-Diseño con Spartan 3 XCV3S50-4
[37]	146	3		123	0.35	- Diseño con virtex II XC2V40
[38]			17984		22.01	-Diseño con xc3s2000
[39]			11720		26.47	-Diseño con xcv1000
[40]			18855		28.5	-Diseño con xc3s200
[41]			86086		32.0	-Diseño con xc4vlx60

CAPITULO 5

CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se presentan las conclusiones y observaciones obtenidas en el desarrollo de este trabajo, además se proponen posibles trabajos futuros basados en el tema objeto de proyecto.

5.1. CONCLUSIONES

5.1.1. Se diseñó e implementó, en hardware reprogramable, el estándar *AES* (*Advanced Encryption Standard*) para tamaño de datos y de clave de 128 bits para los procesos de encriptación y desencriptación en los modos de configuración de entrada no realimentados *ECB* (*Electronic CodeBook*) y *CTR* (*Counter*) utilizando una organización segmentada (*pipeline*) para cada diseño.

5.1.2. Las comparaciones realizadas referentes a la implementación de los modos de operación permitieron determinar que el circuito de encriptación/desencriptación en modo *CTR* es una opción que presenta ventajas considerables con respecto al modo *ECB*, como por ejemplo el uso de un mismo circuito para realizar los procesos de encriptación y desencriptación permitiendo reducir el costo de área.

5.1.3. El circuito de encriptación/desencriptación en modo *CTR* presenta un nivel más alto de seguridad que en el modo *ECB* debido a que los bloques de texto cifrado siempre son diferentes, siempre y cuando se utilicen bloques de contador diferentes para cada bloque de texto plano. Finalmente, el desempeño en frecuencia para el modo *CTR* es mucho mejor con respecto al modo *ECB* y el consumo de potencia en la implementación de cada uno de los modos no presenta grandes diferencias.

5.1.4. Las comparaciones realizadas permitieron determinar que las transformaciones *SubBytes* e *InvSubBytes* basadas en tablas utilizan menos área y son más rápidas que las transformaciones basadas en el modelo matemático, por lo que el modelo basado en tablas es más aconsejable en implementaciones con lógica reprogramable basadas en FPGAs.

5.1.5. Las comparaciones realizadas permitieron determinar que la unidad de subclaves en cascada es más rápida que la unidad de subclaves iterativa. Sin embargo la unidad de subclaves en cascada utiliza más área.

5.1.6. En el modo *ECB* el circuito de desencriptación presenta un costo de área mayor que el circuito de encriptación. Lo anterior se debe a que la transformación *InvMixColumns* realiza más multiplicaciones en $GF(2^8)$ que la transformación *MixColumns*. Sin embargo este hecho no fue una limitante para que el circuito de desencriptación pudiera trabajar a la misma frecuencia de reloj que el circuito de encriptación, consecuentemente las velocidades de procesamiento de datos fueron

iguales tanto para el proceso de encriptación como para el proceso de desencriptación.

5.1.7. En la implementación se utilizó una frecuencia de reloj de 200Mhz, por lo tanto se obtuvo un rendimiento (*throughput*) de 25.6 Gbits/s en los procesos de encriptación y desencriptación *pipeline* en modo *ECB* y *CTR*. Para el caso de los circuitos que procesan múltiples bloques se utilizó una frecuencia de reloj de 100Mhz, por lo tanto se obtuvo un rendimiento (*throughput*) de 12.8 Gbits/s.

5.1.8. El algoritmo implementado en una arquitectura *pipeline* presenta un rendimiento (*throughput*) mucho más alto que el de una arquitectura iterativa, pero el costo de área es mucho mayor.

5.1.9. Implementar el estándar *AES* en modo contador (*CTR*) constituyó el principal esfuerzo académico debido a que son pocas las referencias reportadas en la literatura relacionadas con implementaciones en *FPGA* en este modo de configuración. Algunos trabajos en los que se puede encontrar la implementación del estándar *AES* en modo contador son [12], [17] y [18].

5.2. OBSERVACIONES

5.2.1. Los diseños e implementaciones del estándar *AES-128* se realizaron utilizando el lenguaje de descripción de hardware *VHDL* y fueron simulados y sintetizados con el software de síntesis e implementación de sistemas digitales *Xilinx ISE 13.1*.

5.2.2. La implementación de todos los módulos circuitales y los sistemas completos se llevó a cabo utilizando la *FPGA XC5VLX110T* de la familia *Virtex 5* de *Xilinx* la cual viene integrada a la tarjeta de desarrollo *XUPV5-LX110T* de *Xilinx*. Los resultados de implementación se visualizaron gracias a la herramienta de verificación de hardware *ChipScope pro 13.1*.

5.2.3. El correcto funcionamiento de cada diseño fue verificado de acuerdo con los vectores de prueba suministrados en los estándares *FIPS 197* [1] y *SP800-38a* [2] del *National Institute of Standards and Technology (NIST)*.

5.2.4. Para añadir una interfaz de usuario con comunicación serial a la *FPGA* era necesaria la implementación de una *UART* para la recepción y transmisión de los datos. Se consideraron varios diseños de la *UART* descritos en *VHDL* y aunque en simulación se obtuvieron buenos resultados, no se obtuvieron buenos resultados en la implementación. Finalmente se decidió dejar este trabajo adicional como un trabajo futuro en el que la validación del diseño se realice no solo con cualquier tamaño de texto sino también con formatos gráficos.

5.2.5. En el desarrollo de este trabajo se entendió la importancia de la criptografía en las tecnologías de la información y las comunicaciones, además se aprendió y se exploró el funcionamiento en detalle de un algoritmo criptográfico de clave simétrica (estándar *AES-128*) y los modos en que este se puede utilizar. Los conocimientos adquiridos constituyen un enriquecimiento académico ya que estos no han sido vistos en el transcurso de la carrera, además la culminación de este trabajo brinda la oportunidad para realizar un artículo de difusión del tema.

5.3. TRABAJO FUTURO

5.3.1. Obtener el diseño del circuito integrado (*ASIC*) de la implementación realizada en modo contador (*CTR*) para que sea implementado y adaptado en un dispositivo de red tal como un enrutador ó un *switch*, tomando como referencia el *RFC* (*Request for comments*) 3686 [16].

5.3.2. Realizar el diseño y la implementación en *FPGA* del estándar *AES* en modos de operación realimentados (*CBC*, *CFB*, *OFB*), con el objetivo de comparar características tales como el área, velocidad y consumo de potencia con las obtenidas en los modos no realimentados (*pipeline*).

5.3.3. Realizar una interfaz de usuario y establecer la comunicación con el puerto serial con el objetivo de enviar y recibir datos desde y hacia la *FPGA*. En la interfaz se deberá escribir un texto plano o un texto cifrado que será enviado a la *FPGA*. Una vez se ha procesado el texto plano ó cifrado se debe enviar a la interfaz para que el resultado se pueda visualizar.

5.3.4. Realizar las distintas implementaciones del algoritmo de Rijndael utilizando técnicas asíncronas y estudiar arquitecturas y técnicas que permitan paralelizar el diseño de estas.

REFERENCIAS

- [1] *FIPS 197 (Federal information processing standards publication 197) announcing the advanced encryption standard (AES)*. Noviembre 26 del 2001. Disponible en <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
- [2] *Recommendation for Block cipher modes of operation, methods and Techniques, NIST special publication 800-38A 2001 Edition*. Disponible en <http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>
- [3] Redes de computadoras - Andrew S. Tanenbaum – 4ta edición, 2003.
- [4] *IEEE 1076, Behavioural languages, VHDL language reference manual, 2004*.
- [5] *Xilinx® ISE® Design suite 13 : Release notes guide ,UG631(v13.1), march 1 2011*. Disponible en http://www.xilinx.com/support/documentation/sw_manuals/xilinx_13_1/irn.pdf
- [6] Nieto Londoño Rubén Darío. Tesis *Ph.D: “Diseño e Implementación De Un Cripto-procesador Asíncrono De Bajo Consumo Basado En El Algoritmo De Rijndael”*, 2009. Universidad Del Valle.
- [7] Kaur, S., & Vig, R. (2007). *Efficient Implementation of AES Algorithm in FPGA Device. International Conference on Computational Intelligence and Multimedia Applications (ICCIMA 2007)*, 179–187. doi:10.1109/ICCIMA.2007.250
- [8] Lu, C., & Tseng, S. (n.d.). *Integrated design of AES (Advanced Encryption Standard) encrypter and decrypter. Proceedings IEEE International Conference on Application- Specific Systems, Architectures, and Processors*, 277–285. doi:10.1109/ASAP.2002.1030726
- [9] Jácome-calderón, G., Velasco-medina, J., López-hernández, J., Eisc, E., & Valle, U. (n.d.). *IMPLEMENTACIÓN EN HARDWARE DEL ALGORITMO RIJNDAEL*, 2003.
- [10] Cryptography and network security principles and practices, 4th ed, William Stallings, 2005.
- [11] Nalini, C., Anandmohan, P. ., Poornaiah, D. ., & Kulkarni, V. D. (2006). *An FPGA Based Performance Analysis of Pipelining and Unrolling of AES Algorithm*. *2006 International Conference on Advanced Computing and Communications*, 477–482. doi:10.1109/ADCOM.2006.4289939
- [12] Fu, Y., Hao, L., Zhang, X., & No, B. R. (2005). *Design of An Extremely High Performance Counter Mode AES Reconfigurable Processor*.

- [13] *Virtex-5 FPGA User Guide*, UG190(v5.4), March 16, 2012. Disponible en http://www.xilinx.com/support/documentation/user_guides/ug190.pdf
- [14] *Chipscope Pro 13.1 Software and Cores, User Guide*, UG029(v13.1) march 1, 2011. Disponible en http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/chipscope_pro_sw_cores_ug029.pdf
- [15] *Planahead software tutorial: Design analysis and foorplaning for performance*, UG676 (v13.1), march 1, 2011. Disponible en http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/PlanAhead_Tutorial_Design_Analysis_Floorplan.pdf
- [16] página web: <http://www.ietf.org/rfc/rfc3686.txt> (fecha de consulta: martes 28/05/2013)
- [17] Grabowski, J. S., & Youssef, A. (2007). *An FPGA Implementation of AES with Support for Counter and Feedback Modes*, (December).
- [18] Qu, S., Shou, G., Hu, Y., Guo, Z., & Qian, Z. (2009). *High Throughput, Pipelined Implementation of AES on FPGA*. *2009 International Symposium on Information Engineering and Electronic Commerce*, (x), 542–545. doi:10.1109/IIEC.2009.120
- [19] Weaver, N., Wawrzynek, J. "High Performance, Compact AES Implementations in Xilinx FPGAs". U. C. Berkeley BRASS group. 2002.
- [20] Hodjat, A., Verbauwhede I. "Speed-Area trade-off for 10 to 100 Gbits/s Throughput AES Processor". *IEEE, Thirty-seventh Asilomar Conference on Signals, Systems and Computers*, volume 2, P-p 2147-2150. Los Angeles, Ca., USA, 2003.
- [21] Qin, H., Sasao, T., Iguchi, Y. "An FPGA Design of AES Encryption Circuit with 128-bit Keys". *GLSVLSI`05*, Chicago, Illinois, USA. 2005
- [22] ECRYPT, Information Society. "State of Art in Hardware Architectures". *IST-2002-507932*. European Network of Excellence in Cryptology. Revision 1.0. 2005
- [23] Chodowiec, P., Khuon, P., & Gaj, K. "Fast Implementations of secret-key block ciphers using mixed inner-and outer-round pipelining". *Proceedings ACM/SIGDA, International Symposium on Field Programmable Gate Arrays, FPGA`01*, pages 94-102. Monterey, CA, USA. 2001
- [24] Jarvinen, K., Tommiska, M., & Skitta, J. "A fully Pipelined Memoryless 17.8 Gbits/s AES-128 Crypto". *FPGA`03*, ACM 1-58113-651-x/03/0002. Monterey California, USA. 2003.
- [25] Hodjat, A., Verbauwhede, I. "Minimun Area cost for a 30 to 70 Gbits/s AES Processor". Electrical Engineering Department, University of California, Los Angeles. 2004a

- [26] Hodjat, A., Verbauwheide, I. “A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA”. Electrical Engineering Department, California University, Los Angeles. 2004b
- [27] FIPS, *Federal Information Processing Standards*. (2001b). “Specification for the Advanced Encryption Standard (AES)”. Disponible en: <http://www.ipsec.pl/aes/dfips-AES.pdf>
- [28] Hodjat, A., Schaumont, Patrick, Verbauwheide, I. “Architectural Design Features of a Programmable High Throughput AES Coprocessor”. Electrical Engineering Department, University of California, Los Angeles. 2004
- [29] Weaver, N., Wawrynek, J. “A comparison of the AES candidates amenability to FPGA implementation”. In *Proceedings of the third Advanced Encryption Standard (AES) Candidate Conference*, New York, USA. 2000.
- [30] Gaj, K., Chodowiec, P. “Comparison of the hardware performance of the AES candidates using reconfigurable hardware”. In *Proceedings of the third Advanced Encryption Standard (AES) Candidate Conference*, New York, USA. 2000.
- [31] Chodowiec, P., Khuon, P., Gaj, K. “Fast Implementations of secret-key block ciphers using mixed inner-and outer-round pipelining”. *Proceedings ACM/SIGDA, International Symposium on Field Programmable Gate Arrays, FPGA'01*, pages 94-102. Monterey, CA, USA. 2001.
- [32] McLoone, M., McCanny, J.V. “High performance single-chip FPGA Rijndael algorithm implementations”. In C , . K. Ko , c, D. Naccache, and C. Paar, editors, *Proceedings of 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2162 in Lecture Notes in Computer Science, page 6576, Paris, France. Springer-Verlag. 2001.
- [33] Alam, M., Badawy W., Jullien G. “A novel pipelined threads architecture for AES encryption algorithm”. In M. Schulte, S. Bhattacharyya, N. Burgess, and R.
- [34] Chitu, C., Chien, D., Chien, C., Verbauwheide, I., Chang F. “A hardware implementation in FPGA of the Rijndael algorithm”. In *Proceedings of the 45th Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 507–510. 2002.
- [35] Li, H., Li, J. “A high Performance Sub-Pipelined Architecture for AES”. Department of Mathematics and Computer Science University of Lethbridge, Canada. *Proceedings of the 2005 International Conference on Computer Design (ICCD'05)*. IEEE. 2005.
- [36] Alexander, K., Karri, R., Minkin, I., Wu, K. Mishra, P., Li, X. “Towards 10-100 Gbps Cryptographic Architectures”. IBM Corporation, ECE Department, Polytechnic University, Brooklin, NY. 2002.
- [37] Rouvroy, G., Standaert, F., Quisquater, J., Legat, J. “Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael Very Well Suited for Small Embedded Applications”. UCL Crypto Group, Laboratoire de

Microelectronique, Universite Catholique de Louvain. *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)*. IEEE. 2004.

[38] FX. Standaert, G. Rouvoy, JJ. Quisquater, JD. Legat, "Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware:Improvements and Design Tradeoffs". In the proceedings of CHES 2003, Lecture Notes in Computer Science, vol 2779, pp. 334-350, Springer Verlag

[39] Nalini C. Iyer, Anandmohan P.V., Poornaiah D.V, and V.D. Kulkarni,"High Throughput, low cost, Fully Pipelined Architecture for AES Crypto Chip". Annual India Conference, 2006 New Delhi pp. 1 – 6 2006.

[40] M.R.M. Rizk, M. Morsy, "Optimized Area and Optimized Speed Hardware Implementations of AES on FPGA". International Design and Test Workshop, 2007 2nd Cairo pp. 207 - 217 Dec. 2007

[41] Chih-Peng Fan, Jun-Kui Hwang, "Implementations of High Throughput Sequential and Fully Pipelined AES Processors on FPGA". Proceedings of 2007 International Symposium on Intelligent Signal Processing and Communication Systems Nov.28-Dec.1, 2007 Xiamen, China